CMSC 11

DYNAMIC DATA STRUCTURES

by MCCRISOSTOMO

Objectives:
At the end of this lesson, you should be able to:
a) Declare a pointer
b) Implement a pointer
c) Implement dynamic memory allocation

POINTER REFRESHER
A ***pointer*** is a variable that points to or references a memory location in which data is stored. This means that a pointer holds the memory address of another variable. In other words, it does not hold a value; instead, it holds the address of another variable. It "***points to***" that other variable by holding a copy of its address.

SOME TERMINOLOGIES

1. A ***"pointer"*** stores a reference to another variable sometimes known as its "pointee".

2. A ***"pointee"*** pointed to by the pointer. (Note: A pointer may be set to the value NULL which encodes that it does not currently refer to any pointee.).

3. The ***"dereference"*** operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been set to refer to a specific pointee. A pointer which does not have a pointee is "bad" (below) and should not be dereferenced.

4. A "***bad pointer"*** is a pointer which does not have an assigned pointee and should not be dereferenced. It sometimes causes crashes and and sometimes randomly corrupts the memory of the running program, causing a crash or incorrect manipulation later. Almost always, an uninitialized pointer or a bad pointer address is the cause of ***segmentation faults***.

DECLARING A POINTER
Syntax:
```
data_type *variableName;
```

Example:
You start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of the variable.

Referencing
```
main() {
int *ptr;
int sum;
sum=45;
ptr=&sum;
printf ("\n Sum is %d\n", sum);
printf ("\n The sum pointer is %d", *ptr);
}
```

POINTERS: COMMON BUGS

BUG #1 : Uninitialized Pointer
One of the easiest ways to create a pointer bug is to try to reference the value of a pointer even though the pointer is uninitialized and does not yet point to a valid address.
For example:

```
int *p;
*p = 12;
```

BUG #2: Invalid Pointer References
An invalid pointer reference occurs when a pointer's value is referenced even though the pointer doesn't point to a valid block.
For example:

```
int *p, *q;
p = q;
```

BUG #3: Zero Pointer Reference
A zero pointer reference occurs whenever a pointer pointing to zero is used in a statement that attempts to reference a block.
For example:
```
int *p;

p = 0;
*p = 12;
```

There is no address pointed to by p. Therefore, trying to read or write anything from or to that address is an invalid zero pointer reference. There are good, valid reasons to point a pointer to zero, dereferencing such a pointer, however, are invalid.

NOTE: All of these bugs are fatal to a program that contains them.

POINTERS: WHY? In programming we may come across situations where we may have to deal with data, which is dynamic in nature. The number of data items may change during the executions of a program. The number of customers in a queue can increase or decrease during the process at any time. When the list grows we need to allocate more memory space to accommodate additional data items. Such situations can be handled move easily by using dynamic techniques. Dynamic data items at run time, thus optimizing file usage of storage space.

DYNAMIC MEMORY ALLOCATION

- Memory allocations process
- Allocating a block of memory
- Releasing the used space

The following functions are used in C for purpose of memory management:

Function	**Task**
malloc	Allocates memory requests size of bytes and returns a pointer to the Ist byte of allocated space
free	Frees previously allocated space

Allocating a block of Memory

Syntax:
ptr = (cast-type *)malloc(byte-size);

Example:

```
typedef struct{
char name[20];
int age;
float scores[3];
}Info;

main(){
int *x;
Info *y;
char *z;

x = (int *)malloc(sizeof(int)*100);
```
➔ memory allocation for an array of integers

```
y = (Info *)malloc(sizeof(Info));
```
➔ memory allocation for the structure Info

```
z = (char *)malloc(sizeof(char));
```
➔ memory allocation for a single character

```
}
```

Releasing the used space

Syntax:
free(ptr);

Example:

We can delete the allocated memory above by using the free function:

```
main()
{

free(x);
free(y);
free(z);

}
```