CMSC 11

DYNAMIC DATA STRUCTURES

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
d) And, implement a linked-list

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:

```
int *p;
char *string;
float *f;
double *d;
```

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.

IMPLEMENTING A POINTER

The following example codes show a typical use of a pointer:

Parameter Passing in Functions: Pass-by-Reference

```
main()
{
  int x = 1, y = 2;

  printf("x = %d and y = %d", x, y);


  printf("after executing swap fxn");
  swap(&x, &y);

  printf("x = %d and y = %d", x, y);

}

void swap(int *a, int *b)
{
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

CMSC 11

DYNAMIC DATA STRUCTURES

MCCRISOSTOMO

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:

```
int *x;

x = (int *)malloc(sizeof(int)*100);
```

CMSC 11

DYNAMIC DATA STRUCTURES

MCCRISOSTOMO

Releasing the used space

Syntax:

```
free(ptr);
```

Example:

```
main()
{
  int *x, i;

  x = (int *)malloc(sizeof(int)*100);

  for(i = 0; i < 100; i++)
    *(x+i) = 0;

  free(x);
}
```

Note:

DYNAMIC DATA STRUCTURE: LINKED LISTS

A ***linked list*** is called so because each of items in the list is a part of a structure, which is linked to the structure containing the next item. This type of list is called a linked list since it can be considered as a list whose order is given by links from one item to the next.

WHY LINKED-LIST?

Linked lists and arrays are similar since they both store collections of data.

Disadvantages of arrays are…

1) The size of the array is fixed.

2) Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough". Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than its size, the code breaks.

DECLARING A LINKED-LIST

To build a linked-list, we need a

- *Node* The data for the nodes which will make up the body of the list.

```
struct struct_tag_name {
    data_type_1 var_1;
    data_type_2 var_2;
    . . .
};
```

- *Node pointer* The pointer to nodes.

```
struct struct_tag_name {
    data_type_1 var_1;
    data_type_2 var_2;
    . . .
    struct struct_tag_name * next;
};
```

Example:

```
struct node {
    int data;
    struct node * next;
};
```

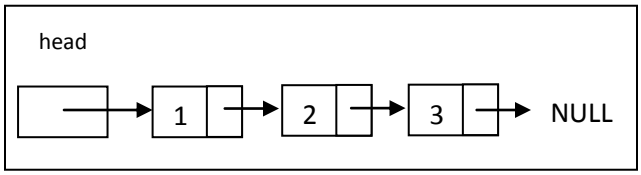
WHAT LINKED-LISTS LOOK LIKE

A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields:

1) *Data field* to store whatever element type the list holds, and a
2) *Next field* which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free().



buildOneTwoThree() Function

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node * next;
};


struct node* BuildOneTwoThree()
{
   struct node* head = NULL;
   struct node* second = NULL;
   struct node* third = NULL;

   head = malloc(sizeof(struct node)); /* allocate 3 nodes in the heap */
   second = malloc(sizeof(struct node));
   third = malloc(sizeof(struct node));

   head->data = 1; /* setup first node */
   head->next = second; /* note: pointer assignment rule */

   second->data = 2; /* setup second node */
   second->next = third;

   third->data = 3; /* setup third link */
   third->next = NULL;

   /* At this point, the linked list referenced by "head" */

   return head;
}

main()
{
       struct node *headPtr = NULL;

       headPtr = BuildOneTwoThree();
}
```

CMSC 11

DYNAMIC DATA STRUCTURES

MCCRISOSTOMO

LIST BUILDING

BuildOneTwoThree() is a fine as example of pointer manipulation code, but it's not a general mechanism to build lists. The best solution will be an independent function which adds a single new node to any list.

3-Step Link In operation

1) Allocate

```
struct node* newNode;
newNode = malloc(sizeof(struct node));
newNode->data = data_to_store;
```

2) Link next

```
newNode->next = head;
```

3) Link head

```
head = newNode;
```

3-Step Link In Code

```
struct node * linkTest(struct node *
head)
{
   struct node * newNode;
   newNode = (struct node
*)malloc(sizeof(struct node));
   scanf("%d", &newNode->data);

   newNode->next = head;

   head = newNode;

   return head;
}
```

ITERATE DOWN A LIST

A very frequent technique in linked list code is to iterate a pointer over all the nodes in a list. Traditionally, this is written as a while loop.

```
void viewAll(struct node * head)
{
   int count = 0;

   struct node* current = head;

   while (current != NULL) {
      printf("node %d: %d",count+1,
current->data);
      count++;
      current = current->next;
   }
}
```

- The *head* pointer is copied into a local variable *current* which then iterates down the list.
- Test for the end of the list with *current!=NULL*.
- Advance the pointer with *current=current->next*.

CMSC 11

DYNAMIC DATA STRUCTURES

MCCRISOSTOMO

SAMPLE PROGRAM

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
		char name[50];
		int stdNo;
		int exam[3];
		int telNo;

		struct node * next;
}student;


student * addStudent(student * head);
void viewAllStudent(student * head);

main()
{
	student * headPtr = NULL;
	int choice;

	do{
		printf("\n ====== MENU ====== ");
		printf("\n [1] - Add Student ");
		printf("\n [2] - Edit Student ");
		printf("\n [3] - Search Student ");
		printf("\n [4] - View All Student ");
		printf("\n [5] - Delete Student ");
		printf("\n [6] - Exit Student ");
		printf("\n Choice: ");

		scanf("%d", &choice);

		switch(choice)
		{
			case 1:
					headPtr = addStudent(headPtr);
					break;
			case 2:
					break;
			case 3:
					break;
			case 4:	viewAllStudent(headPtr);

					break;
			case 5:
					break;
			case 6:	printf("Good bye!");
					break;
			default:
					printf("Invalid input");
		}

	}while(choice != 6);


}
```

```
student * addStudent(student * head)
{
   struct node * newNode;
   newNode = (student *)malloc(sizeof(student));

   scanf("%s", newNode->name);
   scanf("%d", &newNode->stdNo);
   scanf("%d %d %d", &newNode->exam[0], &newNode->exam[1], &newNode->exam[2]);
   scanf("%d", &newNode->telNo);

   newNode->next = head;

   head = newNode;

   return head;
}

void viewAllStudent(student * head)
{
   int count = 0;

   student * current = head;

   while (current != NULL) {
        printf("\nstudent %d:",count+1);
         printf("\n Name: \t %s", current->name);
         printf("\n Student Number: \t %d", current->stdNo);
         printf("\n Exam 1: \t %d", current->exam[0]);
         printf("\n Exam 2: \t %d", current->exam[1]);
         printf("\n Exam 3: \t %d", current->exam[2]);
         printf("\n Tel #: \t %d", current->telNo);

        count++;
        current = current->next;
   }
}
```