Lecture 09

C Structs and Linked Lists

In this lecture

- Structs in C
- → operator
- Structs within BMP files
- Reading Header information from BMP files
- Passing structs to functions
- Passing pointer to structs
- Array of structs and Array of struct pointers
- Concept of a linked list
- Types of Linked List
- Implementation
- Further readings
- Exercises
- Answers

Structs in C

Structs in C are used to package several data fields into one unit. Structs can be used to define types and can be used to define variables of that type. C structs are different from Java classes. Java Classes "encapsulates" both state (data fields) and behavior (methods) with these fields being public, private or protected. C structs can only declare public data fields and all fields are accessible using the dot (.) operator. For example, if we define a typedef for a struct that contain two fields, var1 and var2 as:

```
typedef struct {
    unsigned int var1;
    char* var2;
} node;
```

and a variable mynode of type node

```
node mynode;
```

fields within **mynode** variable can be accessed using,

```
mynode.var1 = 100;
mynode.var2 = malloc(20);    ================== (1)
```

The amount of memory required to hold the variable mynode is equivalent to **sizeof(unsigned int) + sizeof(char*)** . You can also write **sizeof(node)** to find the how many bytes are required to hold a variable of size node. Note that sizes of specific data types are dependent on the machine you may be using. For example, on andrew domain machines(unix.andrew.cmu.edu), an address variable is 4 bytes while in CS domain machines (linux.gp.cs.cmu.edu) an address variable is 8 bytes.

As with initializing variables as they are declared, structs can also be initialized as

node mynode = {100, malloc(20)};

➔ operator

There are two ways to access fields within a struct. Field within a struct can be accessed using the dot operator. However, if a pointer to the struct is given, then we can access fields within the struct using -> operator. For example,

node mynode;
node* ptr = &mynode;
ptr➔var1 = 100;
ptr➔var2 = malloc(20);

is equivalent to the code given above in (1) using the dot operator. The arrow operator will be used extensively in cases where a pointer to the struct is passed to a function instead of a copy.

Structs within BMP files

An interesting example of a struct type is header information stored in a bitmap (BMP) file. BMP, a format invented by Microsoft stores the image using a schema as follows. The first 14 bytes is reserved for information given by the following struct

```
typedef struct {
   unsigned short int type;  /* BMP type identifier */
   unsigned int size;        /* size of the file in bytes*/
   unsigned short int reserved1, reserved2;
   unsigned int offset;      /* starting address of the byte */
} HEADER;

The next 40 bytes are reserved for a structure as follows.
typedef struct {
   unsigned int size;               /* Header size in bytes      */
   int width,height;                /* Width and height in pixels */
   unsigned short int planes;       /* Number of color planes   */
   unsigned short int bits;         /* Bits per pixel            */
   unsigned int compression;        /* Compression type          */
   unsigned int imagesize;          /* Image size in bytes       */
   int xresolution,yresolution;     /* Pixels per meter          */
   unsigned int ncolors;           /* Number of colors         */
```

```
   unsigned int importantcolors;   /* Important colors         */
} INFOHEADER;
```

Suppose we are interested in extracting this information from a BMP file. First we need to read a block of 54 bytes using fread function as follows. Since BMP files are binary files (recall that there are two types of files, ASCII and Binary) and reading bytes from a BMP file needs to be done using fread (instead of fscanf for formatted data)

The prototype of the **fread** function is given by

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

The fread() function reads, into the array pointed to by ptr, up to nitems members whose size is specified by size in bytes, from the stream pointed to by stream. [source:open group]

Reading Header Information from BMP file

Suppose we would like to read header information from a BMP file. Let us first allocate a block of 54 bytes to hold the struct.

void* ptr = malloc(54);

Now we can read 54 bytes from a file stream that is opened.

FILE* infile = fopen("image.bmp", "r");
fread(ptr, 54, 1, infile);

Now suppose we need to find the width and height of the image from header information. We could look at the 54 bytes as follows.

To find the width, we need to offset a total of 18 bytes (14 + 4 bytes for the unsigned int). Hence we can extract information about width as follows.

int* tmp = ptr + 18;
printf("The width of the image is %d \n", *tmp);

Similarly all other information about the BMP file can be extracted and manipulated.

Passing Structures to Functions

Structures can be passed to functions as arguments. For example, consider the following function foo that takes a copy of a struct as an argument. We will be using struct INFOHEADER as defined above in BMP example.

```
int foo(INFOHEADER  info){
   …..

   return 0;
}
```

We define a variable **info** and pass that to function **foo.** A **copy of info** is made and placed on the runtime stack of foo.

```
INFOHEADER  info;
foo(info);
```

Note that the fields within the original variable **info** may not be affected by statements in foo.

When a structure is passed as **value** parameter, a copy of the struct is made on the runtime stack and information is used to perform the operations. If the function does not require permission to change the original variable info, this may be ok. However, a programmer needs to be aware that if the struct is large, then two much information may be copied to run time stack, making the program run slower. A programmer must be careful in a situation where the foo may be called multiple times.

Passing a Pointer to a Struct

To avoid copying large structs within the run time stack, we can pass the address of a struct variable (i.e. a pointer) to a function. There are two instances under which passing a pointer to a struct, instead of a copy of the struct may be advantages. A pointer allows efficient access to the struct (instead of making a copy on runtime stack) as well as the opportunity to directly manipulate the information within the original struct.

For example, if we need to change some information within a field of INFOHEADER, we can pass the address of an INFOHEADER variable to a function whose prototype is given by

```
int foo(INFOHEADER*  ptr){
   …..
   ptr → width = 720;
   return 0;
}
```

In the calling function we can do;

INFOHEADER info;
foo(&info);

Now the fields within the original variable info can be manipulated directly by the ptr.

ptr -> width = 720;

Compromising Security

One disadvantage of passing an address to a function is that we may be compromising the security of the variable. When a pointer to a struct is passed to a function, then the function can change the information within the struct, even if you did not intend to do so. But there is a way to retain the efficiency of passing by reference, while maintaining the security. One possibility is to define a function as follows.

```
int foo(const INFOHEADER*  ptr){
   …..

   return 0;
}
```

This does not allow any changes to the original content, but provides access to the fields within the struct directly from the ptr.

So ptr -> width = 100; would be illegal

while

int tmp = ptr -> width; would be legal

Array of Structs

Structs can be combined to form an array. Suppose we need to define a struct that will store positional information as well as color information of a point in 2D space. A struct 2Dpoint is then can be defined as follows.

```
typedef struct {
    unsigned char R,G,B;  // stores a value between 0-255 representing the color depth
    int x,y;
}  2Dpoint;
```

Then we can define an array of 100, 2D points as follows.

2Dpoint A[100];

The array can also be initialized as

2Dpoint A[100] = {{0,0,255,20,40}, {255,0,255,40,20}, ….};

It should be noted that one could ignore the inner braces as long as the list matches the amount and type of things to be initialized. So we could write

2Dpoint A[100] = {0,0,255,20,40,255,0,255,40,20, ….};

Array of Struct Pointers

In some applications, using an array of struct pointers may be useful. Therefore we can define an array of struct points as follows.

2Dpoint* A[100];

In this case each array element, A[i], is a pointer to a 2Dpoint. Access to the fields can be obtained using

A[i] ➔ R = 255; /* changes the color red to 255 */

Dynamically Allocated Lists

Concept of a linked list

Static arrays are structures whose size is fixed at compile time and therefore cannot be extended or reduced to fit the data set. A dynamic array can be extended by doubling the size but there is overhead associated with the operation of copying old data and freeing the memory associated with the old data structure. One potential problem of using arrays for storing data is that arrays require a contiguous block of memory which may not be available, if the requested contiguous block is too large. However the advantages of using arrays are that each element in the array can be accessed very efficiently using an index. However, for applications that can be better managed without using contiguous memory we define a concept called "linked lists".

A **linked list** is a collection of objects linked together by references from one object to another object. By convention these objects are named as **nodes**. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the next node. The last node points to a NULL reference to indicate the end of the list.

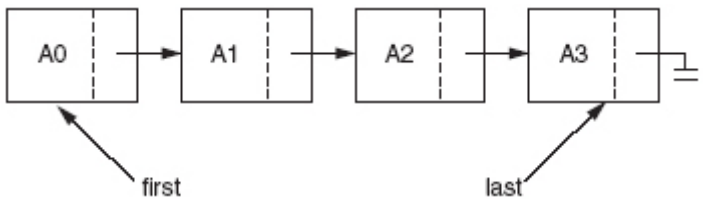


image source: Weiss Data Structures

The entry point into a linked list is always the first or head of the list. It should be noted that head is NOT a separate node, but a reference to the first Node in the list. If the list is empty, then the head has the value NULL. Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be contiguous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle and will be discussed in the next lesson. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

Types of Linked Lists

There are few different types of linked lists. A **singly linked list** as described above provides access to the list from the head node. Traversal is allowed only one way and there is no going back. A **doubly linked list** is a list that has two references, one to the next node and another to previous node. Doubly linked list also starts from head node, but provide access both ways. That is one can traverse forward or backward from any node. A **multilinked list** (see figures 1 & 2) is a more general linked list with multiple links from nodes. For examples, we can define a Node that has two references, age pointer and a name pointer. With this structure it is possible to maintain a single list, where if we follow the name pointer we can traverse the list in alphabetical order of names and if we traverse the age pointer, we can traverse the list sorted by ages. This type of node organization may be useful for maintaining a customer list in a bank where same list can be traversed in any order (name, age, or any other criteria) based on the need.

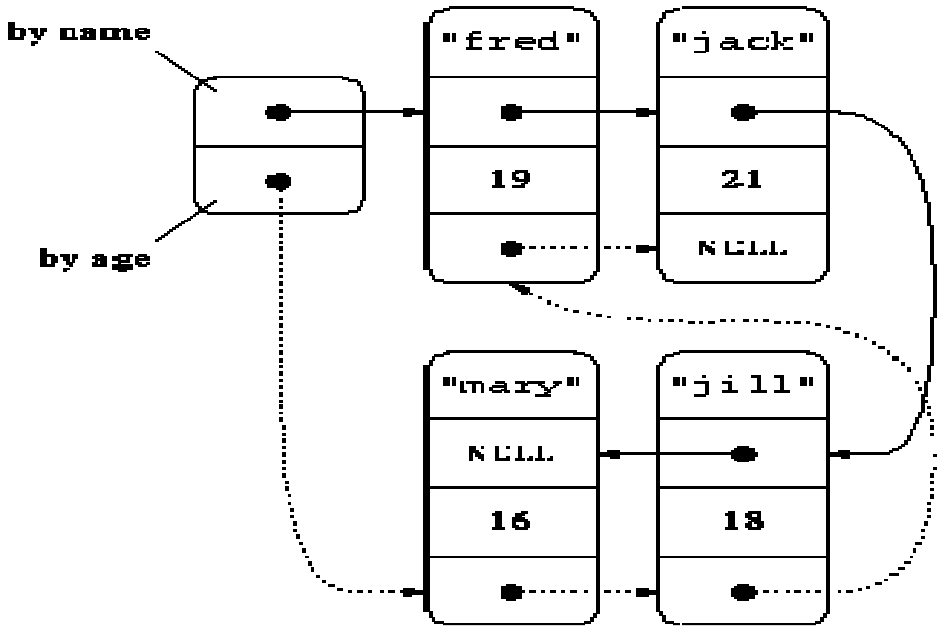


Figure 1 – Linked List with two pointers

Another example of multilinked list is a structure that represents a sparse matrix as shown below.

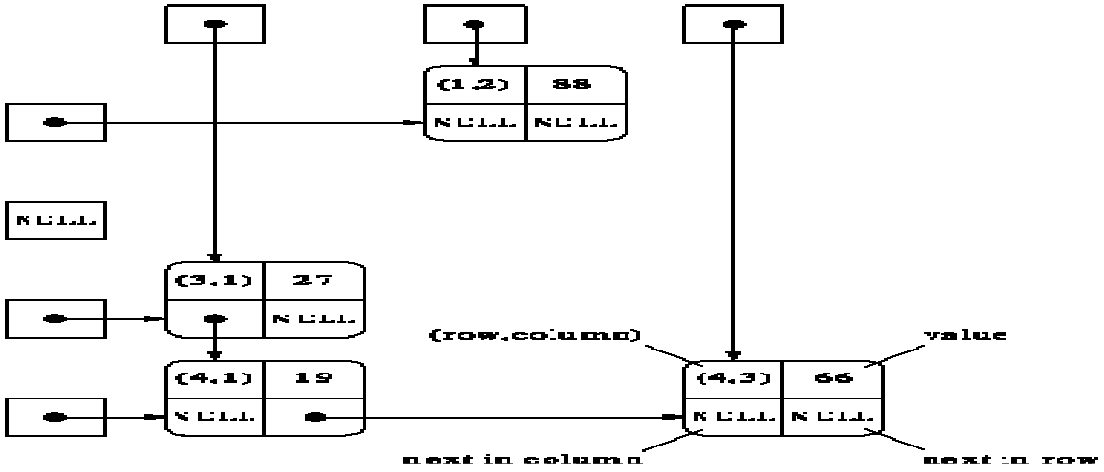


Figure 2 – A sparse matrix representation

Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

Implementation of a Linked List

Designing the Node

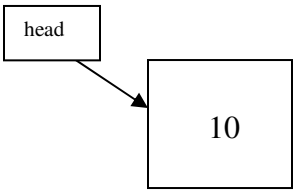
Linked list is a collection of linked nodes. A node is a struct with at least a data field and a reference to a node of the same type. A node is called a **self-referential** object, since it contains a pointer to a variable that refers to a variable of the same type. For example, a struct Node that contains an int data field and a pointer to another node can be defined as follows.

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

typedef struct Node node;

node* head = NULL;
```

Allocating memory for the first node

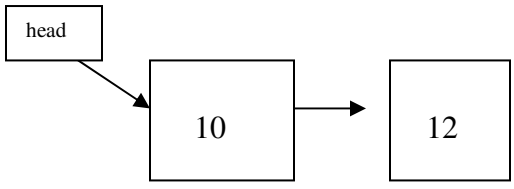
Memory must be allocated for one node and assigned to head as follows.

```
head = (node*) malloc(sizeof(node));

(*head).data = 10;

(*head).next = NULL;
```



Adding the second node and linking

```
node* nextnode = malloc(sizeof(node));
(*nextnode).data = 12;
(*nextnode).next = NULL;
(*head).next = nextnode;
```



Continuation of this process creates a linked list of nodes. The advantages of a linked list as compared to an array is that memory blocks are small and hence there is

more flexibility in managing memory required by the application. In the next lesson we will discuss some operations on linked lists as well as details about implementing a Doubly Linked List.

Further Readings

[1] K & R – chapter 6.1-6.4 – pages 127-138
[2] http://www.cs.cmu.edu/~thoffman/S09-15123/Chapter-4/Chapter-4.html#CHAP_4.1

Exercises

For all of the following exercises (where applicable) use the following definition of node.

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

[1] What would be returned if sizeof(node) is used (assume linux.andrew.cmu.edu)

[2] What is wrong with the following code?

```
node ptr;
ptr → data = 25;
ptr → next = NULL;
```

[3] The following code is supposed to insert a new node, after the first node. (assume there is at least one node). However, the code seems to throw a seg fault after sometime. What could be the reason?

```
node* newnode = malloc(sizeof(node));
first → next = newnode;
newnode → next = first → next;
```

[4] design a struct that can be used to implement a multilinked list as given in Figure 2. Include the proper fields to hold all the data.

Answers

For all of the following exercises (where applicable) use the following definition of node.

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

[1] What would be returned if sizeof(node) is used (assume linux.andrew.cmu.edu)

ANSWER: sizof(node) = sizeof(int) + sizeof(struct node*)
= 4 + 4 = 8

[2] What is wrong with the following code?

```
node ptr;
ptr → data = 25;
ptr → next = NULL;
```

ANSWER: Memory is not being allocated for ptr. Therefore any dereference to Ptr → data could cause a segmentation fault

[3] The following code is supposed to insert a new node, after the first node. (assume there is at least one node). However, the code seems to throw a seg fault after sometime. What could be the reason?

```
node* newnode = malloc(sizeof(node));
first → next = newnode;                  ---------- (2)
newnode → next = first → next;           ----------  (3)
```

ANSWER: The problem seems to be that this creates an infinite loop situation. We can fix the code by switching lines 2 and 3.

[4] design a struct that can be used to implement a multilinked list as given in Figure 2. Include the proper fields to hold all the data.

ANSWER:

```
typedef struct node {
        int row, col;
        double value;
        struct node* rowptr;
        struct node* colptr;
} node;
```