C Programming (6)

Scribe Notes for Lecture 7

By: Jacob Horner, Joseph Hernandez, Chris Iaquinto, Crystal Ho

Conditional Operations

#ifdef: enter is the option is passed (defined)

#ifndef: enter if the option is not passed (not defined)

Conditional directives help the preprocessor decide what text goes to the compiler or not. We, the programmer, can define some directives to be added only if another is defined. This is common for debugging code (see Example Below).

```
project.c:

#ifdef DEBUG
#include "mylib.h"
#define malloc mymalloc
#define free myfree
#endif
...
p = malloc (100);
```

In this example, the only way for the code after #ifdef but before #endif to be passed to the compiler is for the option DEBUG is passed as a directive from the command line. That is done in the form:

```
gcc -DDEBUG -o project project.c mylib.c
```

Another common use of conditional operators is once-only headers (can also be thought of the wrapper #ifndef). Basically this is used to prevent compilation errors in several situations. For example, if a header file is included more than once, this can cause errors or at the very least slow down your programs performance. We can use #ifndef so that we only define things that are not already defined. The code sample below is the example from class:

```
/* File foo.  */
        #ifndef FILE_FOO_SEEN
          #define FILE_FOO_SEEN

        ***the entire file***

        #endif /* !FILE_FOO_SEEN */
```

A Quick Note on Streams (more later)

In class we observed the following code on slide four.

```
void *mymalloc(int size);
void myfree(void *ptr);
mylib.c:
void *mymalloc (int size) {
        void *ret = malloc (size);
        fprintf (stderr, "Allocating: %d at %p\n", size, ret);
        return ret;
}
void myfree(void *ptr) {
        fprintf(stderr, "Freeing: %p\n", ptr);      free(ptr);
}
```

It is here that we see the first instance of a stream. The following code causes memory allocation values to be output to stderror (the standard error stream).

Linked Lists

A linked list is a data structure that contains a set of data. The data is held in structures called nodes. These nodes point to other nodes hence "linking" them through pointers. The first note is called the head and the last element of the list has a pointer which points to NULL. In iterating through the list, we generally keep a pointer pointing to the head as well as the current element of the linked list.

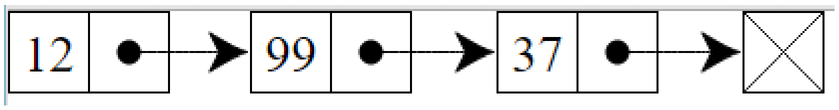
Linked lists are useful for many reasons. They are dynamic in size which means they can grow and shrink to allow more flexibility in the running of the program. They are also very fast to access. Linked lists have O(1) insert and deletion time.

Creating a Linked List

1. Create the basic block of the list often called a node. This is the code snipit from class where val is the value of the element and next is a pointer to the next node in the list

```
struct test_struct {
   // various data items making up the basic element
   int val;
   struct test_struct *next;
}
```

2. We now must create the elements that will be the head and current nodes of the list. This is done by creating a new instance of the node structure and assigning it values. REMEMBER: we must allocate memory for the new node.



```
struct test_struct *ptr =
(struct test_struct *) malloc (sizeof (struct test_struct);
```

3. After the structures are created, we assign them values, “val” gets the value of the node and “next” get the pointer value to the next node. At first the next node is just NULL. See the code

```
                    ptr->val = val;
                          ptr->next = NULL;
```

below for a complete example of creating a linked list from class.

```
struct test_struct *head = NULL;
struct test_struct *curr = NULL;
struct mystruct* create_list (int val) {
   printf ("\n creating list with headnode as [%d]\n",val);
   struct test_struct *ptr = (struct test_struct*) malloc                  (sizeof (struct test_struct));
   if (NULL == ptr) {
       printf ("\n Node creation failed \n");
       return NULL;
   }
   ptr->val = val;
   ptr->next = NULL;
   head = curr = ptr;
   return ptr;
}
```

Linked Lists Made Easier with Functions

Many of the operations we perform on linked lists can be coded once into a function so that we can just reuse that function to save on code space and prevent code repetition. These functions include but are not limited to adding elements, removing elements, and searching for an element.

When we add an element to the list, we must pass in the parameters we want assigned to that item. This will be dependent on how many data types are represented in your list node. In class our node only has one integer data type in addition to the pointer to the next node. The in class example showcased below has the added option of allowing your new element to append or prepend the list.

```
struct test_struct* add_to_list (int val, bool add_to_end) {
    if (NULL == head) {
        return (create_list(val));
    }
    struct test_struct *ptr = (struct test_struct*)
              malloc (sizeof (struct test_struct));
    if (NULL == ptr) {
        return NULL;
    }
    ptr->val = val;
    ptr->next = NULL;
```

```
if (add_to_end) {
   curr->next = ptr;
      curr = ptr;
         }
       else {
   ptr->next = head;
      head = ptr;
         }
   return ptr;
      }
```

Searching for an element in the list is also easily implemented with a function. To search the list, we must traverse the whole list either from head to the desired element (if it exists) of head to NULL (if the element being searched for is not in the list. Once again the code example from class showcases this below.

```
struct test_struct* search_in_list (int val, struct test_struct **prev){
   struct test_struct *ptr = head;
   struct test_struct *tmp = NULL;
   bool found = false;

   while(ptr != NULL) {
        if(ptr->val == val) {
            found = true;
            break;
        }
        else {
            tmp = ptr;
            ptr = ptr->next;
        }
   }
```

```
if(true == found) {
          if(prev)
          *prev = tmp;
        return ptr;
           }
         else {
        return NULL;
            }
          }
```

When an element is deleted from the list we have to be careful in writing the function that deletes it. There are multiple steps to deleting the node form the list. We must find the element and return it, then to delete the node, we reassign the pointers (essentially removing it from this list) and then we must deallocate the memory for the deleted node. If this is not done properly, it will be the source of a memory leak in the final program. Again the code from the in class example is below.

```
int delete_from_list (int val) {
    struct test_struct *prev = NULL;
    struct test_struct *del = NULL;

    del = search_in_list (val,&prev);
    if (del == NULL) {
        return -1;
    }
    else {
        if (prev != NULL)
            prev->next = del->next;
        if (del == curr) {
            curr = prev;
        }
        else if (del == head) {
            head = del->next;
        }
    }
```

```
free (del);
  del = NULL;

   return 0;
     }
```

Basic Input and Output in C

Inputs and outputs are pretty self-explanatory. Input is any way that you the user interact with the program from the outside (mouse, keyboard, microphone, etc…). Similarly, output is anyway that you get feedback from the program (terminal, monitor, speakers, etc…). In C programing, these are dealt with as streams. Later in the course we may discuss Network IO, but not at this time.

Streams

Input Output (IO) is usually handled in things known as streams. On a basic level a stream is just a series of objects which are accessed in sequential order (often bytes). We use pointers to iterate through these streams in many cases. There are many streams, but the ones we shall focus on are as follows:

stdout => standard output (usually for monitor)

stderr => standard error output

stdin => standard input (usually from keyboard)

Files can also be treated as streams, and while we are not talking about it right now, network sockets are as well.

Functions used to Operate with Streams

There are several functions that we use to work with streams:

puts => dumps stream passed to it straight to the terminal (nor formatting, just a line)

printf => like puts, but allows for the use of formatting

Those streams functions are default to deal with stdout, we can use other stream functions to operate on streams other than the stdout.

fputs => same as “puts” but into another specified stream such as stderr

fprintf => same as “printf” but into another specified stream such as stderr

Requesting inputs to the stream is just as easy an can be done with the following functions.

fgets => gets an input, stdin, often defined to be from the keyboard

scanf => same as fgets but for multiple inputs at the same time

File IO

There are two common ways to deal with the input and output of files. There is the C Standard library which uses functions such as fopen and fclose and deals with files as streams. There is also the POSIX stile which deals with file descriptions rather than just streams. POSIX uses the open and close functions as well.

To read or write from/to a file, the file must be opened. In POSIX, you open the file by passing the file path and access permissions that you wish to have during that access. The open function returns an integer; this is used as the file descriptor. Once on want to read or write to the now open file, you call either read or write with the file descriptor, a buffer, and the length of the data to be read or written. The functions actually return the number of bytes read or written for data integrity checking.

We also have other library functions which operate with File IO when we access the files as streams. They are detailed below from the examples in class:

> lseek(fd, numbytes, SEEK CUR);
>
> > => Seek numbytes from from current location.
> >
> > => Useful because it allows fast traversal through our file.
>
> sync();
>
> > => apply changes made to the buffer to the disk, calling sync ensure that your changes in the buffer are transferred to the disk itself.
> >
> > =>Not needed for the most part.
>
> FILE *ffd = fdopen(fd, "r");
>
> > => Construct a stream from file-descriptor.
>
> fprintf(ffd, "format", args);
>
> > => Write formatted text output to file.
>
> fscanf(ffd, "format", args);
>
> > => Read formatted input from the file.
>
> fclose(ffd);
>
> > => Close a stream
> >
> > => DO NOT FORGET TO CLOSE YOUR STREAMS