Structures & Pointers

Program Persistent Data

Lecture 6

Review

- In C there are buffers required to work with files.
- Streams are declared using **`FILE *fp`**;
- These streams are required for each file that you work on.
- To open and use the stream, error check that the file exists then close when finished:

```
fp = fopen("write.txt","w");
if (fp == NULL)
    {printf("Can't open file.\n");}
fclose(fp);
```

Review – *text* file <stdlib.h>

Instruction	Meaning
`fgetc(fp)`	Read a char from file using stream.
`fputc(fp)`	Write a char to file using stream.
`fgets(string,size,fp)`	Read a string from file using stream. It reads a string of a specified size.
`fputs(string,fp)`	Write a string to file using stream.
`fprintf(fp,“Hi %s, you are %i”,s,a)`	Write the content to the file using stream.
`fscanf(fp,”%s %s %i”,a,b,&c)`	Read a formatted line from file using stream.

#include <string.h>

Name	Example	Meaning
1. Strlen()	`len=strlen(str);`	Get the length of a string
2. strcmp	`strcmp(str, "jane" )`	Compare 2 strings
3. strcpy	`strcpy(str, “jane”);`	Copy a strings to another
4. strcat	`strcat( str, “ Ferris” );`	Concatenate 2 strings
5. strstr	`Strstr(str, “jane”)`	Look for a substring in a string

Non text files

Name	Example	Meaning
fread	`fread(var, size, number, FILEpointer);`	Read from position in file
fwrite	`fwrite(var, size, number, FILEpointer);`	Write to position in file
fseek	`fseek(FILE, offset in byte, whence);` `SEEK_SET/ _CUR or _END`	Go to 1 of 3 places; start, current or end of file
ftell	`ftell(FilePointer);`	Where is the pointer now?
rewind	`rewind(FilePointer);`	Point to start if the file



Useful ctype functions

<ctype.h>

Function	?
isalnum()	Is it Aa-Zz or 0-9?
isalpha()	Is it Aa-Zz?
isascii()	Is it asci code0-127
iscntrl()	Is it ascii code 0-31 or 127?
isdigit	Is it from 0-9?
isgraph()	Is it everything BUT a space?
islower()	Is it a-z?
isprint()	Is it printable?
ispunct()	Is it punctuation char?
isspace()	Is it asci code 9-13 or 32?
isupper()	Is it A-Z?

Structs

- A **struct** in **C** is a data type declaration that groups variables of different types under one name and is in one block of memory.
- Struct types are declared for each element within a template.
- The dot operator is used to access the elements.

Review: accessing structs

```
struct employee {
      int id;
      int age;
      float salary; };
main()
{     FILE * fp;
      struct employee employee;
      fp=fopen("database.dat","rb");
      //move to the third positon
      fseek(fp,2*sizeof(employee),SEEK_SET);
      //Read the third employee
      fread(&employee,sizeof(employee),1,fp);
      //read (& needed!)
      //Display
      printf("ID: %i \n", employee.id);
      printf("Age: %i \n", employee.age);
      printf("Salary: %f \n", employee.salary);
      fclose(fp); }
```

Functions

- Functions adds further customised ‘functionality’ to c.
- Functions have prototypes (similar to declarations) coded external to main().
- And they must be accessed only via the method prototyped.
- Functions have: `type name (argument1, argument2 ..)`
  - `type`; returned value type (output).
  - `name`; unique name.
  - `arguments`; variables required (passed) to process the function.

Function examples

1. `void function (void)`

- This is a void function that appears to return and is passed no values. Called in main() by `function();`

2. `void function1(char ch)`

- This is a void function that is passed a char named ch.

3. `int function2(void)`

- This is an int function that is passed no values.

4. `int function3(int a)`

- This is an int function that is passed no values.

void function (void)

```
#include <stdio.h>

void function(void)
                    {
                    puts("soup");
                    }


main()
{
puts("For lunch I had ");
     function();
}
```

void function1(int a,int b)

```
/*From fresh2fresh.com swap function*/
#include<stdio.h>
void function1(int a, int b); {
       int tmp;
       tmp = a;
       a = b;
       b = tmp;
       printf(" \nvalues after swap m = %d\n and n = %d", a, b);
                     }
main()
{
    int m = 22, n = 44;

    printf(" values before swap  m = %d \n and n = %d",m, n);
    function1 (m, n);
    //function1 will swap and print the numbers
}
```

int function2(void)

```
/*Main is this form of function*/
int main(void)
{
int i;
for(i=1; i <= 5; i++)
     {
     printf("%d ", i*i);
     }
 return 0;
}
```

float function3(float a)

```
/*From fresh2fresh.com square function*/
#include<stdio.h>
float function3 ( float x )
      {     float p ;
            p = x * x ;
            return ( p ) ;     }
main( )
{
    float m, n ;
    printf ( "\nEnter some number for finding square
            \n");
    scanf ( "%f", &m ) ;
    n = function3 ( m ) ;
    printf ( "\nSquare of the given number %f is
%f",m,n );
}
```

Pointers

- Pointers point to a variable that holds a memory location.
- A pointer can be any type, it must be declared and initialised before it is used.
  - If not there are lots of errors.
  - Use the asterisks to declare
- The address operator ‘&’ fetches the memory address.
- The indirection operator ‘*’ returns the value of the variable.

```
/*Returning from the struct*/
#include <stdio.h>
#include <string.h>
char *longer(char *s1, char *s2)
{       int len1,len2;
        len1 = strlen(s1);
        len2 = strlen(s2);
        if( len1 > len2 )
                return(s1);
        else
                return(s2);
}
main()
{       char *string1 = “This is text";
        char *string2 = “This is more text";
        char *result;

        result = longer(string1,string2);
        printf("String '%s' is longer.\n",result);
}
```

Accessing via the pointer

- Use the pointer to access the elements.
- Used the dot operator previously but 2 other methods that use pointers:

1. (*ptrName).title
2. ptrName->title

```
e.g. friend.age=55;
(*friendPtr).age=55;
 friendPtr ->age=55;
//all equivalent
```

```
#include <stdio.h>
#include <string.h>
struct student            {
            int id;
            char name[20];
            char grade [3];} record1 = {1, “Sean", “A++”};
main()
{      struct student record2, record3;
      printf("  Id : %d \n  Name : %s\n  Percentage : %f\n",
      record1.id, record1.name, record1.percentage);
record2.id=record1.id;
strcpy(record2.name, record1.name);
record2.percentage = record1.percentage;
      printf("  Id : %d \n  Name : %s\n  Percentage : %f\n",
      record2->id, record2->name, record2->percentage);
record3=record2;
      printf("  Id : %d \n  Name : %s\n  Percentage : %f\n",
      record3.id, record3.name, record3.percentage);
}
```

Pointers to structures

- Structure like arrays can be accessed using pointers.
- The struct pointer must first be declared.
- Address of Pointer variable can be obtained using ‘&’ operator

```
struct book *ptr, B;
//Single structure variable & Pointer of Structure type
ptr=&B;
```

```
/*Pass by value*/
#include <stdio.h>
#include <string.h>

struct student                  {
            int num;
            char name[20];
            char grade [3];           };

 void funcctionWrite(struct student record)
      {
            printf(" Id is: %d\n", record.num);
            printf(" Name is: %s \n", record.name);
            printf(" Percentage is: %s \n",record.grade);
      }
main()
{           struct student record;
            record.num=1;
            strcpy(record.name, "Jane");
            strcpy(record.char, "A+");

            functionWrite(record);
 }
```

```
/*Pass by address of the struct*/
#include <stdio.h>
#include <string.h>

struct student                     {
            int num;
            char name[20];
            char grade [3];}record;

 void funcctionWrite(struct student *record)
           {
            printf(" Id is: %d \n", record.num);
            printf(" Name is: %s \n", record.name);
            printf(" Percentage is: %s
            \n",record.char);
                                                  }
main()
 {          record.num=1;
            strcpy(record.name, "J");
            strcpy(record.grade, "A+");
 }
```

Passing FILE streams

```
void my_write(FILE *fp, char *str)
  { fprintf(fp, "%s", str); }
```

Function to open file

```
int fopenFile (FILE **FILE
pstream, const char *fn)
 { *pstream = fopen(fn, "r");
 return (*pstream != NULL) ? 0 : -
1; }
```

Arrays of structs

- The use of structs are further expanded through the use of arrays.
- Previously all we could do is a single record (instance).
- Commonly in DBs we require multiple records .
- Arrays store the same variable type so arrays of structs.

```
struct inventory {
      int partNo;
      float cost;
      float price; };
struct inventory DB[25]
//This will hold 25 records for inventory
DB[0].number=1234;
//each struct member element via [element#]
```

Writing to an array of structs

```
#include <stdlib.h>
#include <stdio.h>
struct employee     {
       int id;
       int age;
       float salary; };
main()
{      FILE * fp;
       struct employee employees[10];
       //employees is an array of 10 employee variable
       int i =0 ;
       fp=fopen("database.dat","wb");
       for (i=0;i<10;i++) //fill the array with content
       {employees[i].id=i+1;
        employees[i].age=i+20;
        employees[i].salary=10000*i;     }
       //write the array in one instruction
       fwrite(&employees,sizeof(employees),1,fp);
       fclose(fp);}
```

Reading from an array of structs

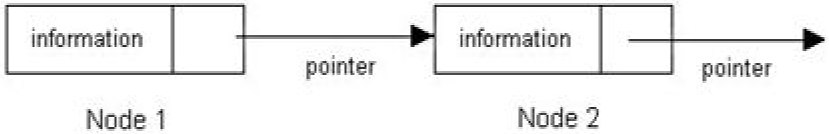
```
struct employee{
              int id;
              int age;
       float salary; };
main()
{      FILE * fp;
       struct employee employees[10];
       int i =0;
       fp=fopen("database.dat","rb");
       fread(&employees,sizeof(employee),1,fp);
       //Display all the employees
       for (i=0;i<10;i++)
       {      printf("== Employee Data ==\n");
              printf("ID: %i \n", employees[i].id);
              printf("Age: %i \n", employees[i].age);
              printf("Salary: %f \n",
employees[i].salary);}
       fclose(fp);}
```

Dynamic Data Structures

- Arrays and Structures are fixed size data structures.
- These are considered ‘concrete’ at execution time.
- Often we require more flexible structures that grow and shrink such as Queueing items for writing to files or to disk.
- Dynamic data structures are built from linked lists which are built from structs & pointers.

Flexible structures require ‘self reference’

- Self referential structures contain a pointer to a struct of the same type.
- The struct member that holds the pointer (ptr) is the link and links to the node of the other structure




```
struct node {
        int information;
        struct node *ptr; };
```

Linking the nodes

- Once the nodes and pointers are declared they need to be initialised and the nodes linked.
- This may be FIFO or reversed or created in any method you require.

```
struct node A = {1, NULL};
struct node B = {2, &A};
struct node C = {1, &B};
//the NULL pointer is used as it is
not linked
```

Printing the list in reverse

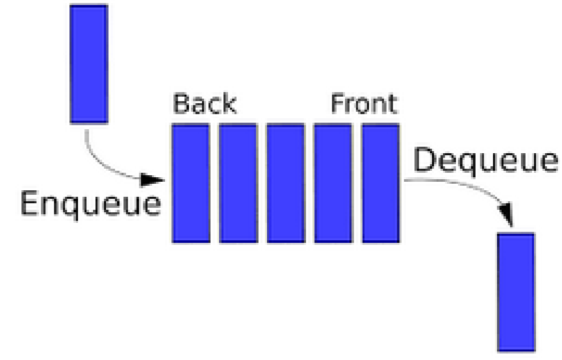
```
#include <stdio.h>
struct node {
                char name[10];
                struct node *next;
            };

void printLIST(struct node *current)
        { while (current!=NULL)
                {printf("%s \t", current->name);
                current=current->next;}}
main()
{       struct node a1={"Jane", NULL};
        struct node b1={"Sean", &a1};
        struct node c1={"Paul", &b1};
        struct node d1={"Mark", &c1};
printLIST(&d1);
}
```

Linked list applications

- A linked list allows for flexibility.
- You can insert or delete nodes anywhere in the linked model.
- Queue (FIFO)

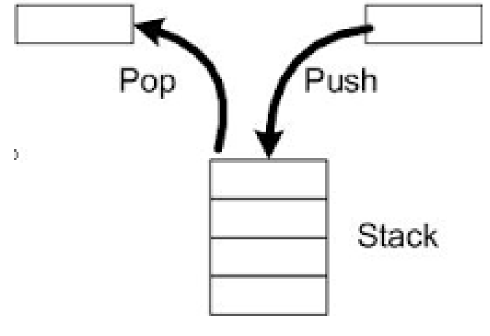
This is a First In First Out linked list



- Stack (LIFO)

This is a Last In First Out

linked list.



What do dynamic structures need?

- Linked lists are flexible they may grow and shrink.
- Lots of advantages over arrays.
- Dynamic structures need Memory.
- If the structures is growing we will need to provide memory to link new nodes.
- To allocate memory at runtime we need `malloc()`
- If shrinking we may release memory held.
- To release memory we **must** use `free()` - to avoid a memory leak.

`malloc()`

`pointer= malloc(size)`

- This function allocates a continuous block of memory and returns a pointer to the start of block assigned.
- If can't give the memory then a NULL pointer returned.
- `sizeof` is very useful with `malloc()`.
- `sizeof(struct node)`
- Important to use as the size of structs are determined not only by content but OS considerations.

Allocating new memory

- To allocate new memory for an additional node.
- First declare a pointer for the new memory location then call malloc to give the correct amount determined by sizeof:

```
struct node *newPTR;

..

newPTR= malloc(sizeof(struct
node));
```

Freeing memory

- To deallocate the memory to avoid holding excessive memory use `free()`.
- The free function will free the memory indicated to by the pointer.
- The pointer must first have been used by malloc.

```
struct node *newPTR;
..
newPTR= malloc(sizeof(struct node));
..
free(newPTR);
```

Linked Lists

- Linked lists are self referential flexible data structures.
- They are constructed from structs which contain a pointer to another struct of the same type.
- To provides additional memory to growing nodes you must provide memory from the heap.
- This must be managed efficiently using `malloc()` and `free()`.