Lesson 8 POINTERS IN C

INTRODUCTION TO POINTERS

A pointer is a variable that stores an address of another variable. So the contents of a pointer will be an address in memory. The address in memory may be the address of some other variable such as an integer or double or string etc. Pointers provide a way of accessing a variable without refering to the variable directly. They can access a variable indirectly via the address of the variable. Pointers are used when passing actual values is not possible or needs to be avoided. Common usage of pointers include.

- returning a value from a function
- passing arrays and strings to functions
- using pointers to access/move arrays
- using linked lists and binary trees
- dynamic memory allocation

We have already used pointers in a previous lesson, namely a pointer constant. When passing an array down to a function, we actually passed the address of the array ie the array name. The address in this case was a pointer constant – recall the difference between a constant and a variable.

We shall now examine pointer variables. The difference between pointers constants and pointer variables is as follows. A pointer constant is an address but a pointer variable is a place to store an address.

POINTERS AND FUNCTION ARGUMENTS

We have already seen how arguments to functions in C are passed by value. This means that there is no direct way for the called function to change the value of a variable in the calling function. Type in the following code for an example of this.

```c
#include <stdio.h>

void change(int , int);

int main(void);
      {
       int x,y;
       x=4;y=2;
       change(x,y);
       printf(" x= %d  y=%d",x,y);
       return(0);
      }

void change(int a,int b)
      {
       int hold;
       hold = a;
       a= b;
       b= hold;
      }
```

In the code above, copies of x and y are swapped in the function change using a and b. The original variables x and y remain unchanged. Modify the code if necessary to convince yourself of this. We can change the values of x and y by passing their addresses to the calling function. This is done using the address of operator `&`. Thus instead of the function call *change(x,y)* we use *change(&x,&y)*. Now we are passing down two addresses to integers or `pointers to intergers' to the function change.

We must also make the following changes to the function *change* .

```c
void change(int *a,int *b)  /* arguments are now pointers to intergers */
```

```
    {
      int hold;
      hold  = *a;
      *a  = *b;
      *b = hold;
     }
```

The symbol `*' means "pointer type" in a definition statement and in other statements it means variable pointed to by. When applied to a pointer, it accesses the object the pointer points to. Thus the definition statement *hold = *a* sets *hold* equal to the contents at the address that *a points to.*

In the statement *int *a*, this tells the compiler to set aside space in memory to hold an address of an integer variable ie. the variable *hold* now equals the contents of the address that *a* points to. This is the contents of the address of *x*, which is 4. So now *hold* is equal to 4.

By a similar argument the contents of the address that *a* points to is equal to the contents of the address that *b* points to ie contents of *x* is equal to the contents of *y*. Finally, the last assignment statement sets the contents of *b* equal to *hold*. As an exercise, you should make a sketch diagram with two boxes representing the two functions and arrows indicating what exactly is going on.

Using pointers inside a function

When using pointer variables in the above content two rules should be remembered.

- ***ptr** is the contents of **ptr**
- **&var** is the address of **var**

Consider the following code which demonstrates the use of pointers

```
#include <stdio.h>
int main(void)
      {
        int x=10,y=12;
        int *ptrx, *ptry;

        ptrx = &x;       /* block b */
        ptry = &y;

        *ptrx = ++(*ptrx); /* block c */
        *ptry=  --(*ptry);
        printf(" x is now %d and y is now %d \n",x,y);
        return(0);
       }
```

Note the 2 assignment statements in block b, the pointer variables are set equal to the address of the variables *x* and *y* . In block c, the contents of ptrx and ptry are increased / decreased by one. Run the above program and make sure that you understand what is going on.

POINTERS and ARRAYS

In C an array name is defined to be a constant pointer to the beginning of the array. In order to access the elements in an array using pointers the following notation can be used. Consider the array definition

```
int hold[5] = {30,45,60,75,90};
```

Since the array name is itself a pointer, then variable *hold* is basically an address. The contents of hold is given by **hold* which is the first element of the array ie the integer 30. The fourth element in the array *hold[3]* can be obtained using the following pointer notation **(hold+3)*. This works as follows, since *hold* is an address of an array and the array stores integers, then the statement asks for the contents at the address in memory position

(memory position of array)+(array index*no. of bytes per storage type) thus suppose that the address in memory of *hold* is 1700 then the statement **(hold+3)* points to the contents at memory address (1700)+(3*2)=1706 i.e. the integer 75.

***(array+index)** is the same as **array[index]**

The are two ways in which we can refer to the address of an array. As above *hold+index* in pointer notation and *&hold[index]* in array notation. The question of which notation to use is an open one, however there are times when one is forced to use pointer notation.

The difference between pointer constants and pointer variables

Consider the code below which calculates the average of in-putted numbers

```
#include <stdio.h>
int main(void )
      {
       int i=0,number;
       float sum=0.;
       float num[50];

        do
          {
           printf("enter number %d \n",i+1);
           scanf("%f",num+i);
          } while( *(num+i++) > 0);

        number=i-1;
        for(i=0;i<=number;i++)
        sum+=*(num+i);
        printf("average = %f \n",sum/number);
        return(0);
        }
```

The above code could have been written using only array notation. Using pointer notation, the statement *while(*(num+i++)>0)* seems rather sloppy. The question of whether we could use the notation *while(*(num++) > 0)* seems a fair one, however, this is not allowed. The reason is that *num* is a pointer constant not a variable. Thus its address can not change.

However by declaring a pointer variable and setting its value to that of the arrays we can use the shorter notation. This is achieved by using the declaration and assignment statements

```
float *ptr_num;  ptr_num=num;
```

Now we can refer to *ptr_num* just as we have referred to num. The difference is that since *ptr_num* is a variable, we can use the increment operater as follows.

```
ptr_num++;
```

Note that because the pointer actually points to an array of type float, the `++` operator causes it to be incremented by four bytes (recall the storage requirements for a float).

By using a pointer variable, the program should now run faster since only one variable is referenced in the loop. Previously two variables were referenced ie *i* and *num*. This increase in speed may seem negligible however for a very large program with many function calls, it may may be me noticeable.

Passing arrays to a function and using pointers to manipulate them

We shall look at the case where a pointer in a function can manipulate an array whose address is passed as an argument to the function. Consider the following code which squares every element in an array.

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

```
#define SIZE 3
void fuct_sq(int *);

int main(void)
      {
       int i;
       static int hold[SIZE]={3,4,5};
       fuct_sq(hold);
       for(i=0;i< SIZE;i++)
            printf( "%d \n ",hold[i]);
       return(0);
       }

void fuct_sq(int *ptr)
       {
        int k;
        for(k=0;k< SIZE;k++)
            *(ptr+k)=  ( *(ptr+k)  ) *  ( *(ptr+k) ) ;
       }
```

Compile and run the code above and make sure that you are clear on the programs workings.

POINTERS and STRINGS

Many string functions in C work with pointers. An example is the function strchr() which returns a pointer to the first occurrence of a particular character in a given string. You should write a short program that allows the user to type in a sentence and search for a character in the sentence using strchr().

INITIALISING STRINGS as POINTERS

Just as a string can be initialised as an array, we can initialise a string as a pointer. In the following piece of code we use both an array and pointers for a string. Note that your compiler may complain about the gets() function and suggest that you use fgets() instead which is safer. As an exercise, use the fgets() function in the code below.

```
#include <stdio.h>
int main(void)
      {
        static char *ptr = "HELLO";
        char name[50];
        puts(" Enter your name");
        gets(name);
        puts(ptr);
        puts(name);
        return(0);
      }
```

The statement in line one is used to initialise the string. We could have used array notation instead.

```
static char hold[]="HELLO";
```

There is a slight difference however. In the array version, enough memory is set aside so that the word HELLO plus `\0` is stored. The address of the first character of the array is given the name of the array ie *hold*. Also the address *hold* is a constant pointer. In the pointer version, an array is set aside in the same way, but a pointer variable is set aside with it. The key word here is variable.

Thus in the pointer version *name* can be changed. The expression *puts(++name);* now makes sense. See what it outputs.

Initialising an array of pointers to strings

The next program is similar to the array of strings program in the previous chapter. Now we initialise the strings as pointers.

```
#include <stdio.h>
#define num 4
int main(void)
      {
       int i,check=0;
       char name[50];
       static char *list[num] = { "John","Pat","June","Kate" };
       puts(" ENTER YOUR NAME");
       gets(name);
       for(i=0;i<=3;i++)
         if(strcmp(list[i],name)==0) check=1;
       if(check)
        printf("YOU MAY ENTER");
       else
       printf("SORRY ,NO ENTRY");
       return(0);
     }
```

You should compare this program with the program in the previous lesson. The difference between the storage method above and the storage method using arrays is as follows. In the array version, the strings were stored in a rectangular array of size 4 rows and 20 columns. In the pointer version the strings are stored contiguously in memory thus there is no wasted space between them. This is one advantage of using the second approach. Another advantage is the flexibility in the manipulation of strings in the array. Consider the following program which allows the user to type in ten names into an array and then the names are sorted into alphabetical order.

```
#include <stdio.h>

#define NUM 3
#define LEN 50

int main(void)
      {
       static char name[NUM][LEN];
       char *ptr[NUM];
       char *temp;
       int i,j,count = 0;

        while(count < NUM)
             {
               printf("NAME %d: ", count+1);
               gets(name[count]);
               if(strlen(name[count])==0) exit(1);
                      ptr[count++] = name[count];
             }

         for(j=0;j < count-1;j++)
            for(i=j+1;j< count;i++)
               if(strcmp(ptr[j],ptr[i]) > 0)
                 {
                   temp = ptr[i];
                   ptr[i] = ptr[j];
                   ptr[j] = temp;
                 }

          printf("\n SORTED LIST:\n");
          for(j=0;j < count;j++)
              printf(" NAME %d: %s\n",j+1,ptr[j]);
        return(0);
       }
```

In the above program we have used an array of pointers and an array of strings. The statement

ptr[count++] = name[count];

assigns the address of each string, which is stored in the array *name[][]*, to an element of the pointer array *ptr[]*. The array of pointers is then sorted. The variable *name* is a two dimensional array however we have used only one index in the statement above. This is because a two dimensional array can be considered as an array of arrays. Thus we have set up a one dimensional array of size 10 where each entry is a string i.e. an array of size 50. This means that *name[0]* is the first string in the array, *name[1]* the second and so on.

POINTERS TO POINTERS

In C we can have pointers that point to pointers. This is often called double indirection. Consider the following program which uses pointers to pointers to reference a two dimensional array.

```
#define ROW 3
#define COL 5
#include<stdio.h>
#include<math.h>

int main(void)
      {
       static int hold[ROW][COL]= {{3,4,6,6,8},{7,10,2,9,7},{19,4,8,1,5} };
       int j,k;
       for(j=0;j< ROW;j++)
          for(k=0;k< COL;k++)
             *(*(hold+j)+k) = pow(*(*(hold+j)+k),2);
              puts("table of numbers squared \n");

        for(j=0;j< ROW;j++)
           {
            for(k=0;k< COL;k++)
            printf(" %d ", *(*(hold+j)+k) );
            printf("\n");
           }
        return(0);
       }
```

In the example code above, we could have used the notation

hold[j][k]

instead of

((hold+j)+k)

The latter works as follows. Since the compiler knows how many COLS there are, the expression hold+1 takes the address of hold say 1000 and adds the number of bytes in a row i.e. 5 times 2 (since int). This gives the address 1010 which is the address of the second one dimensional array in the array. In a similar way, hold+2 give the address 1020 which is the address of the third array in the array. The address of the first element in this array is given by *&hold[2][0]* i.e. third row first column. In pointer notation this is given by **(hold+2)*. Note that *hold+2* and **(hold+2)* refer to the contents of the same address but they use different units of measure. If you add 1 to *hold+1* you get *hold+1* i.e. the third row address of hold(1020). If you add 1 to *(hold+1) you get the address of the next element in the row i.e. your looking at 1012. This is the element in row 2, column 2 and is given by *(hold+1)+1, thus its contents at that position is given by *(*(hold+2)+1) i.e. 10.

POINTERS TO FUNCTIONS

We can obtain the address of a function just as we obtained the address of an array. As in the case of an array, the name of the function contains the address of that function. We can use the address of a function to

- assigned it to a variable which has been declared to be a pointer to a function
- be passed as an argument to another function

An example of where a pointer to a function is useful can be seen from the following. Suppose we wish to output an independent variable x and dependent variable f(x) i.e. two columns, where f(x) is the square of x, the square root of x, the cube of x etc. We can write a general function which produces two columns of numbers depending on the relationship between the two numbers. This relationship can be a function which can have address passed to the general function. Consider the following code

```
/* pointer to a function program */
#include<stdio.h>
#include<math.h>

void  table(int ,int ,float (*f)(int) );
float sqroot(int);
float square(int);
float cube(int);

int main(void)
     {
      table(1,10,square ); /*change to cube etc. */
      return(0);
     }

void table(int x0,int x1, float(*f)(int)  )
     {
      int i;
      for(i=x0;i < x1;i++)
          printf("%d %f  \n", i, (*f)(i) );
     }

float sqroot(int a)
      {
       return( sqrt(a) );
      }

float square(int a)
      {
       return(a*a);
      }

float cube(int a)
      {
       return(a*a*a);
       }
```

Compile the above program and run it using different arguments to the function *table*. The printf statement contains the function call (*f)(i). In this statement,

- f is a pointer to a function
- *f is the function
- i is the argument of the function and
- brackets are needed around f because of precedence (i.e. brackets are higher than *)

PROGRAMMING EXERCISES 1

1 Explain in detail the idea behind parameter passing in C.
2 Write a program that supplies a function with 2 values and returns a value that lies between the two. (Do not use the return statement).
3 Write a short program that contains a character array with your name in it and uses a pointer to char, to print out a chosen letter from the array characters
4 Write a program to solve a quadratic equation using the well known formulae. The function header that solves the equation should look like the following *void solve_eq(a,b,c,r1,r2);*
5 Modify the program in Q4 so that the function returns a number (int instead of void) that informs the user about the nature of the solutions to the equation ie two distinct solutions, one solution or complex solutions.
6 Write a program that asks a user for the capital city in various countries around the world. Two arrays should be used, one for country and the other for city. The program should display the country and the user must guess the capital city. Incorporate a suitable scoring scheme and allow the user to give up. Print the percentage correct at the end.
7 Modify the program the above so that the user can choose to be tested on countries given the capital city.
8 Write up your own summary notes on pointers to check your understanding.