Lab Session 03

Pointers in C

When a variable is created in C, a memory address is assigned to the variable. The memory address is the location of where the variable is stored on the computer. When we assign a value to the variable, it is stored in this memory address. To access it, we can use the reference operator (&), and the result represents where the variable is stored:

```
int myNum = 43;
printf("%p", &myNum);
```

Note that the memory address is in hexadecimal form. This takes us to the concept of pointers.

Pointers

A pointer is a variable that stores the memory address of another variable as its value. Note that pointers are one of the things that make C stand out from other programming languages, like Python and Java.

A pointer variable points to a data type (like int) of the same type, and is created with the ***** operator. The address of the variable you are working with is assigned to the pointer. Let's have a look at an example:

Example 1

```
int myNum = 43;
int* ptr = &myNum;
printf("%d\n", myNum);
printf("%p\n", &myNum);
printf("%p\n", ptr);
```

In the above example, a pointer variable with the name ptr is created, that points to an int variable. Note that the type of the pointer has to match the type of the variable you're working with (int in our example). Use the **&** operator to store the memory address of the myNum variable, and assign it to the pointer. Now, ptr holds the value of myNum's memory address. Note that the memory location is going to be different for everyone. The output for the above example will be as shown below:

```
43
000000000022FE44
000000000022FE44
```

You can also get the value of the variable the pointer points to, by using the * operator (the dereference operator):

```
printf("%d\n", *ptr); //outputs the value of myNum
```

Null pointer

It is always good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Let's have a look at an example:

Example 2

```
#include <stdio.h>

int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %p\n", ptr  );
   return 0;
}
```

When the above code is compiled and executed, it says that the value of ptr is 0. In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, do the following:

```
if(ptr)     // succeeds if p is not null
if(!ptr)    //succeeds if p is null
```

Pointers and Arrays

The name of an array is actually a pointer to the first element of the array. Let's understand this through an example:

Example 3

```
int myNum[4] = {25, 50, 75, 100};
printf("%p\n", myNum);
printf("%p\n", & myNum [0]);
```

Both the print statements will give the same output. This basically means that we can work with arrays through pointers. Since myNum is a pointer to the first element in myNum, you can use the * operator to access it:

```
printf("%d", *myNum);
```

To access the rest of the elements in myNum, you can increment the pointer/array (+1, +2, etc):

```
printf("%d", *( myNum + 1)); //access first element
printf("%d", *( myNum + 2));  //access third element
```

We can also loop through the array using pointers:

```
int *ptr = myNum;
for (int i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

The elements of an array can also be modified:

```
*myNum = 13; //changes the first value
*(myNum +1) = 17; //changes the second value
```

Pointer arithmetic

There are four arithmetic operators that can be used on pointers: ++, --, +, and -.

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer :

```
ptr++
```

After the above operation, the ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. Similarly, if ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

Example 4

```
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
   /* let us have address of the first element in pointer */
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] ) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
      /* point to the next location */
      ptr++;
      i++;
   }
   return 0;
}
```

Array of pointers

There may be a situation when we want to maintain an array, which can store pointers to variables. Following is the declaration of an array of pointers of type integer.

```
int *ptr[MAX];
```

This declares an array of pointers containing MAX elements. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers.

Example 5

```
const int MAX = 3;
 int main () {
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
   for ( i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
   for ( i = 0; i < MAX; i++) {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
   return 0;
}
```

Pointer to pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int :

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

Example 6

```
int main () {
   int  var;
   int  *ptr;
   int  **pptr;
   var = 3000;
   /* take the address of var */
   ptr = &var;
   /* take the address of ptr using address of operator & */
   pptr = &ptr;
   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

Passing pointers to functions in C

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

Example 7

```
void getSeconds(unsigned long *par);
int main () {
   unsigned long sec;
   getSeconds( &sec );
   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );

   return 0;
}
void getSeconds(unsigned long *par) {
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

A function can also accept an array as a pointer as shown in the following example:

Example 8

```
/* function declaration */
double getAverage(int *arr, int size);
int main () {
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;
   /* output the returned value  */
   printf("Average value is: %f\n", avg );
   return 0;
}
double getAverage(int *arr, int size) {
   int  i, sum = 0;
   double avg;
   for (i = 0; i < size; i++) {
      sum += arr[i];
   }
   avg = (double)sum / size;
   return avg;
}
```

Return pointer from functions in C

C also allows returning a pointer from a function as shown below:

Example 9

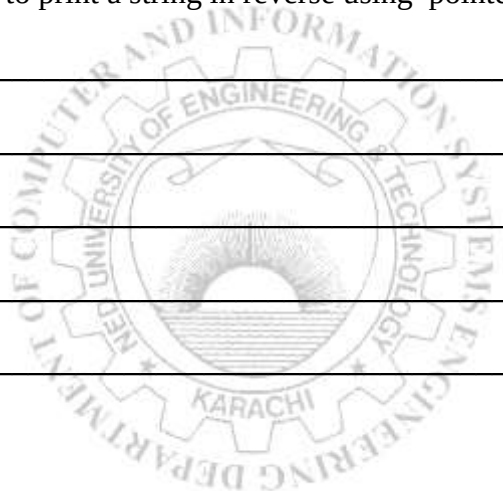
```
/* function to generate and return random numbers. */
int * getRandom( ) {
   static int  r[10];
   int i;
   /* set the seed */
   srand( (unsigned)time( NULL ) );
   for ( i = 0; i < 10; ++i) {
      r[i] = rand();
      printf("%d\n", r[i] );
   }
   return r;
}
int main () {
   /* a pointer to an int */
   int *p;
   int i;
   p = getRandom();
   for ( i = 0; i < 10; i++ ) {
      printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }
   return 0;
}
```

In the above example, it can be seen that the getRandom() function generates 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element. Note that it is not a good idea to return the address of a local variable outside the function, so you would have to define the local variable as static variable.

EXERCISES

1. Write a program in C to swap elements using call by reference.

2. Write a program in C to print a string in reverse using pointers.



3. Write a C program to input and print array elements using pointers.

4. Write a C program to search for an element in an array using pointers.

5. Write a C program to add two matrices using pointers.