Lab Serie No. 01: Pointers

Address in C

Every created variable in C is a memory location and every memory location has its defined address which can be accessed using the **ampersand (&) operator** (*also called **reference operator***).
If we have a variable var in our program, **&var** will give us its address in the memory. We have already used addresses numerous times while using the **scanf()** function.

```
scanf("%d",&var);
```

Consider the following working example, where values (***42** and **HELLO***) are stored in the addresses of the respective variables **var1** and **var2**. The program prints then the address of the defined variables and their values.

```
#include <stdio.h>

int main () {

   int var1= 42 ;
   char var2[10]= "HELLO";

   printf("Address of var1 variable: %p\n", &var1);
   printf("Value of var1 variable: %d\n", var1);

   printf("Address of var2 variable: %p\n", &var2);
   printf("Value of var2 variable: %s\n", var2);

   return 0;
}
```

Note that **&var1** is often called a "pointer". To print pointer values, we use the **%p** format specifier.

Pointer in C

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, we must declare a pointer before using it to store any variable address.

The general form of a pointer variable declaration is :

```
type *Pointer-name;
```

Example :

```
int * ptr;
```

*The **asterisk(*)** can be surrounded by spaces and placed anywhere between the type and the identifier. Thus, the following three definitions are identical.*

```
int * ptr;                 int* ptr;                   int *ptr;
```

Operations on pointers

There are a few important operations to do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using the **asterisk operator *** that returns the value of the variable located at the address specified by its operand. The following example demonstrates these operations:

```
#include <stdio.h>

int main () {

   int  x = 20; /* integer variable declaration */
   int  *p;     /* pointer variable declaration */

   p = &x;      /* store address of x in the pointer p */

   printf("Address of x variable: %p\n", &x  );

   /* address stored in pointer variable */
   printf("Address stored in p variable: %p\n", p);

   /* access the value using the pointer */
   printf("Value of *p variable: %d\n", *p );

   return 0;
}
```

Null pointers

It is always a good practice to assign a NULL value to a pointer variable in case we do not have a valid 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.
To check for a null pointer, We can use an 'if' statement as follows:

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

Pointer arithmetic

We can perform arithmetic operations on a pointer just as we can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –
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. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. 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**.

The same considerations apply to decrementing a pointer **ptr --** , which decreases its value by the number of bytes of its data type.

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, they can be meaningfully compared.

Pointers on arrays

Pointers and arrays in C are closely related concepts. In C, the name of an array is a constant pointer to its first element. Understanding pointers and arrays is essential for efficient memory manipulation and accessing elements in C. Here are some key points regarding pointers on arrays:

- When we declare an array, its name is essentially a pointer to the first element of the array. It cannot be reassigned to point to another location
- Pointer arithmetic allows us to navigate through an array by incrementing or decrementing the pointer.

```
int A[5];
&A[0] ↔ A
&A[1] ↔ A+1
&A[i] ↔ A+i
```

```
A[0] ↔ *A
A[1] ↔ *(A+1)
A[i]  ↔*(A+i)
```

```
int A[5];// Assuming an integer array A of size 5
int * p
for(p=A;p<A+5;p++){
scanf("Introduce a value %d \n",p);}
```

- The pointer **p** is used to iterate through the array **A** and reads integer values using scanf.
- In the loop, the pointer **p** is initialized with the address of the first element of the array **A** and allows to iterate through the array.
- The loop continues as long as **p** is less than the address of the element beyond the end of the array (i.e., **A + 5**).
- scanf("%d", p);: Reads an integer value from the user and stores it at the memory location pointed to by **p**.

Pointers on structures (*records*)

The unique allowed operations are:

- Accessing a member of the structure.
- Get the memory address of the structure (*using the ampersand operator* &)

```
struct date {
int  day ;
int month ;
int year ;
} ;
```

We may use pointers to point to structures, such as shown in the following example:

```
struct date x,y, *p; //p is a pointer to the structure with the type date
```

In order to access to the **day** member in the structure **date**, we write:

(*p).day Or : **p->day**

Memory allocation

Allocating space for an element pointed to by the pointer **p** on any type is done through the function `malloc(sizeof(type))`. This function allows to allocate a memory space of the same size as the type of the pointed variable. Here is an example:

```
int *p;
p = malloc(sizeof(int));  /* Or */ p = malloc(sizeof(*p))
```

Freeing up memory space

Freeing space occupied by the element at **p** is done through the function "Free(p)", this function allows to recover (free) an already allocated memory space. This operation returns nothing, the memory pointed to by **p** is restored. Let's consider the following example:

```
int *p;
p = malloc(sizeof(*p));
Free(p);
```

Exercises 01

Write a program that reads an integer variable and assigns its address to a pointer. Then displays the value and the address of this variable in two ways.

Exercise 02

Write a program that reads two integers and calculates their sum through the use of pointers.

Exercise 03

Write a program that fills an array of integers and calculates the sum of its elements, using a pointer for its traversal.

Exercise 04

Write a program that defines a structure for storing the first name, the last name and the age of a person. Then, fills the array with two individuals and displays the full name of the younger person using a single *printf* function for displaying the result.

Exercise 05

Write a program that allocates memory space for an array of integers of a given size **N**. Fill the array and then display its elements.