

Pointers

Pointer is a location in the memory that contains a memory address. Most commonly, this address is a location for another variable in the memory. If a variable hold the address of another variable, we say that the first variable points to the second.

Declaring a pointer:

Declaring pointer is like declaring any other variable, it must have a name and the type of address that it will hold and an operator that tell the compiler it is a pointer

type *name;

example:

char *ptr; —————> pointer to character

This mean that the variable ptr will hold the address of a location in the memory contains character.

When dealing with pointer, there are two special pointer operators that are commonly used:

The (&) operator : it is a unary operator that returns the memory address of its operand.

The (*) operator : it is a unary operator that returns the value of the variable located at the address that follows.

Example:

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

```
int main()
```

```
{
```

```
    int n;
```

```
    int *pn;
```

```
    { n = 5;
```

```
      pn = &n;
```

```
      printf("The value of n %d \n", n);
```

```
      printf("The address of n %p \n", pn);
```

```
      printf("The value of n using pointer %d \n", *pn);
```

```
      return 0;
```

```
    }
```

1 → These two statements in order has no effect on the program i.e. we can write:

```
    pn = &n;
```

```
    n = 5;
```

2. They can also be written as follow:

```
pn = &n;  
*pn = 5;
```

this is also correct but here we can't change the place of the two statements i.e we can't write:

```
*pn = 5;  
pn = &n;
```

because when executing these two statements, it will put 5 in the location where its address in pn which is not the same address of n.

Then put the address of n in the pointer pn.

Pointer Expression:

- With pointers, we can use the assignment operator to assign a pointer to another pointer

Example:

```
#include <stdio.h>  
int main()  
{  
    int x;  
    int *p1, *p2;  
    p1 = &x;  
    p2 = p1;  
    printf("%p %p", p1, p2);  
    return 0;  
}
```

- We can use arithmetic operator the addition and subtraction (+ , - , ++ , --)

Note:

Each time a pointer is incremented, it points to the memory location of the next element of its base type.

Example:

```
# include <stdio.h>  
int main()  
{  
    int x;  
    int *p1, *p2;  
    p1 = &x;  
    p2 = p1;  
    p1++;  
    printf("%p %p", p1, p2);  
    return 0;  
}
```

in this example, if the address of p1 = 1000 and we increment the pointer p1. P1 will be 1002

That's because p1 hold the address of a variable of type integer and each integer reserve 2bytes in the memory.

char *ch;

3000
3001
3002
3003

int *n;

3000
3002
3004
3006

Pointers as function argument (call by address)

Let's make a function swap to put the content of the first in the second and the second in the first. It will be as follow:

```
# include <stdio.h>
void swap (int x, int y);
int main()
{
    int a,b;
    a = 5;
    b = 7;
    swap(a,b);
    printf("a = %d \n",a);
    printf("b = %d ",b);
}
void swap (int x, int y)
{
    int temp;
    temp =x;
    x = y;
    y = temp;
}
```

When running this program it won't change the content of a and b

It will send a copy of a to x, and y to b then in the function swap will change the content of x and y then won't return any thing

And also we can't return more than one variable. This type of passing the argument is named call by value.

Call by value: In this method, a copy of the argument to the parameter of the function. In this case, changes made to the parameter have no effect on the argument.

To solve this program, there is another way of calling function which is named call by address

Call by address:

It is the second way of passing argument to a subroutine. In this method the address of an argument is copied into the parameter.

This is done as follow:

The same example of the swap

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

```
void swap (int* x, int* y);
```

```
int main()
```

```
{
```

```
    int a,b;
```

```
    a = 5;
```

```
    b = 7;
```

```
    swap(&a,&b);
```

```
    printf("a = %d \n",a);
```

```
    printf("b = %d ",b);
```

```
}
```

```
void swap (int* x, int* y)
```

```
{
```

```
    int temp;
```

```
    temp =*x;
```

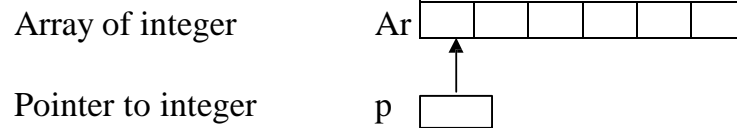
```
    *x = *y;
```

```
    *y = temp;
```

```
}
```

Pointer and Arrays:

There is a close relationship between pointers and arrays.



p is a pointer of type integer, so it can hold the address of any location in the memory contains integer.

Ar is an array of integer.

So we can assign the address of any element of the array to the pointer p.

The array name is the address of the first element in the array

i.e.: `Ar = &Ar[0]`

So, we can say `p = Ar` OR `p = &Ar[0];`

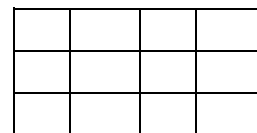
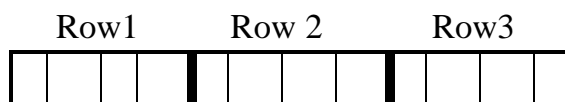
In this case, we can deal with the element of the array using pointers as follow:

Example:

```
int Ar[10];
int *p;
p = Ar;      //suppose p = 100
p++;        //means p = &Ar[1] i.e. 102
*p = 3;      //the address 102 will contain 3
```

Note:

- Pointer doesn't automatic allocated but, it points to address already allocated.
In this case: array allocated addresses
Pointer point to that address
- Don't exceed the element of the array.
- The 2D array is allocated in memory row wise



The 2D array physically
In memory

2D as we think

So, we can use only one pointer to points to all elements in the 2D array.

Example:

Write a program that read 10 element then print their sum using array and pointer

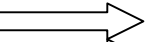
Solution

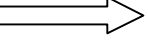
```
#include <stdio.h>
```

```
int main()
{
    int Ar[10];
    int *pn;
    int I, sum = 0;
    pn = Ar;
    for(I = 0; I < 10; I++)
    {
        scanf("%d", pn+I);
    }
    for(I = 0; I < 10; I++, pn++)
    {
        sum += *pn;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

Note:

For better performance:

If we deal with array sequentially  use POINTER

If we deal with array randomly  use INDEX

Example:

Write a program that print a string in a reverse order

Solution

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

```
int main()
{
    char str[100];
    char *pstr;
    pstr = str;
    scanf("%s", str);
    while(*pstr)
    {
        pstr++;
    }
    pstr--;

    printf("The reverse string is : ");
}
```

```

        while(pstr >=str)
        {
            printf("%c", *pstr);
            pstr--;
        }
        return 0;
    }

```

Another Solution:

```

#include<stdio.h>
void PrintStrRev(char *p);
int main()
{
    char Ar[100];
    scanf("%s", Ar);
    PrintStrRev(Ar);
    return 0;
}
void PrintStrRev(char *p)
{
    if(*p)
    {
        PrintStrRev(p+1);
        printf("%c", *p);
    }
}

```

When passing an array to a function, we just send its address, so it is called by address.

The name of string is a pointer which stores the address of the first character.

Array of pointers:

As any other type, we can make array of pointers. This means that we make an array that holds addresses.

To declare array of pointer:

```
Data_type *pointer-name[size];
```

Example:

```
int *x[10];
```

this mean that we declare an array named x of 10 element each will hold the address of a location contain integer.

To assign the address of an integer variable called var to the third element of the array:

```
X[2] = &var;
```

To find the value of var :

```
*x[2]
```

To pass an array of pointer to a function, we can use the same method used for the other arrays- simply call the function with the array name without any indexes.

Example of a function prototype:

```
void display_array(int *o[]);
```


Dynamic Allocation (Under request):

the memory for a program work under dos is 64k. It is divided into:

- 1) A part for the global variables and static variables, in this area, the variables keep its value as the program is running it is about (0.5k).
- 2) The stack is the part that contains all local variables and function, it is about (4k)
- 3) The heap it is the third part of the memory, it is almost empty and it is about $(64 - 4 - 0.5 = 59.5k)$

The problem begin when we declare an 10 arrays each of 50 element of type double this mean that we need amount of memory:

$$10 * 50 * 8 = 4000\text{byte OR } 4K.$$

This means that we need the entire stack for these arrays and it may be that we don't need them at the same time.

So, we must have a way to use the Heap in locating variables instead of using the stack.

THIS IS THE DYNAMIC ALLOCATION

In the dynamic allocation we use the heap to allocate variables, we send a request if there is a free space in the heap equal to the space we request. It allocate it, this space mustn't has any gaps OR it reply by can't allocate. Here in theb dynamic allocation, we will just declare a pointer in the stack and this pointer will hold the address of the first element allocated in the heap.

Note:

We request to allocate the memory, so we must release the memory when finish using it.

In using dynamic allocation, we must do two things: allocation and release. These statements are done by two function defined in the header file `stdlib.h`

To allocate an area:

Use the function **malloc** it has the following prototype:

```
void* malloc(size to be allocated);
```

this function returns a pointer to the newly allocated block of memory but it is pointer to void, so we must make type casting for the return value of malloc() function. If there is not enough space or couldn't allocate the block, it returns NULL

to determine the size to be allocated, we can use the operator **sizeof(type)** it returns the size of the type between () in bytes

To release an area:

Use the function **free** it has the following prototype:

```
void free(void* p);
```

Note:

To release an allocated block, you must have a pointer to the first element in it. So, don't lose the pointer you get from malloc.

Dynamic allocation for one element:

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *pn;
    pn = (int *)malloc(sizeof(int));
    if(!pn)
    {
        printf("Out of memory.\n");
        exit(1);
    }
    else
    {
        printf("%d\n", (*pn) * (*pn));
        free(pn);
    }

    return 0;
}
```

To allocate an array:

```
int *pn;
pn = (int *) malloc(10 * sizeof(int));
/*This will allocate 10 places in the heap, each can hold an
integer. pn hold the address of the first element of them*/.
delete pn;
```

Advantage of dynamic allocation:

- 1) Allocate variable size depend on the user input

```
int *pn;
```

```
int num;
```


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

```
pn = (int *) malloc(num * sizeof(int));
```

↳ determined at runtime by user

- 2) Save stack for important variable and pointer and deal with heap for large size.

Ex: if we declare local variable:

Double x[200];  200 * 8 bytes
= 1600 byte \approx 1.5k

but if we use dynamic allocation, pointer declared in the stack with size (2-4 byte) and the array is stored in the heap. So, we can save the stack size.

Example:

Write a program that calculates the sum of numbers in an array. The array size is variable upon the user need.

Solution

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

```
#include <conio.h>
```

```
int main()
```

$$\{$$

```
int *ptr;
```

```
int size , I , sum , j;
```

```
printf("Enter the element number ");
```

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

```
ptr = (int *) malloc(size * sizeof(int));
```

```
//check if the allocation fail
```

```
if(!ptr)
```

$$\{$$

```
printf("Out of Memory\n");
```

```
exit(1);
```

}

```
//Entering the data
```

```
for(I = 0 ; I < size ; I++)
```

$$\{$$

```
scanf("%d", &ptr[I]);
```

}

```
//processing the data
```

```
sum = 0;
```

```
for(j = 0 ; j <size ; j++)
```

$$\{$$

```

        sum += ptr[j];
    }
    //Output the result
    printf("The Sum = %d", sum);
    //deallocate the pointer
    free(ptr);
    return 0;
}

```

In the last program, in the processing part, we deal with the pointer as an array. We can deal it in another way by using pointer as follow:

(MODIFICATION OF PROCESSING THE DATA)

```

//processing the data
int *ptr1; //this is defined at the beginning of the program
ptr1 = ptr;
sum = 0;
for(j = 0 ; j < size ; j++,ptr++)
{
    sum += *ptr;
}
//Output the result
printf("The Sum = %d", sum);
//deallocate the pointer
free(ptr1);

```

In the last program, in the processing part, we deal with the pointer as an array. We can deal it in another way by using pointer as follow:

(MODIFICATION OF ENTERING DATA)

```

//Entering the data
for(I = 0 ; I < size ; I++,ptr++)
{
    scanf("%d", ptr);
}
ptr = ptr1;

```

So, the whole program will be:

```

#include <stdio.h>
#include <conio.h>
int main()
{
    int *ptr;
    int size , I , sum , j;
    int *ptr1;
    printf("Enter the element number ");
    scanf("%d", &size);
    ptr = (int *) malloc(size * sizeof(int));

```

```

//check if the
allocation
fail if(!ptr)
{
    printf("Out of
    Memory\n");
    exit(1);
}
ptr1 = ptr;
//Entering the data
for(I = 0 ; I < size ; I++,ptr++)
{
    scanf("%d", ptr);
}
ptr = ptr1;
for(j = 0 ; j < size ; j++,ptr++)
{
    sum += *ptr;
}
//Output the result
printf("The Sum = %d", sum);
);
return 0;
}

```

Pointer to structure:

We can declare a pointer to a structure, but to deal with its element, we use the (ARROW "->") operator instead of the dot operator. But before use the pointer, be sure that it points to the structure by assigning the address of a variable of this structure to the pointer or by using dynamic allocation.

Example: struct bal

```

{
float balance; char name[80];
}person;

```

```

int main()
{
struct bal *p; p = &person;
p->balance = 10;
return 0;
}

```

Example 2: (using dynamic allocation)

```
struct bal
{
float balance; char name[80];
};
int main()
{
struct bal *p;
p = (struct bal *) malloc(sizeof(struct bal));
p->balance = 10;
free(p); return 0;
}
```

Lab Assignments

- 1- do swap function (call by value and call by reference) two swap two integers.
- 2-send array of integers to a function as a pointer and in this function ask user to enter data of each element.
- 3- send the previous array to a function to print data of each element to user.
- 4- send the previous array to a function that calculates the sum of all elements and returns the sum to main function to print the sum in main
- 5- send the array to a function and a number to search about it in the array. And returns the first index in where it finds the number, if it does not found it it returns -1.
- 6- allocate one dimensional array dynamic to store degree of student in subjects the number of subjects determined by user.