

Pointers in C

By: Atiya Jokhio

Pointer

- Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value.
- A pointer contains the memory address of a variable that, in turn, contains a specific value.
- In this sense, a variable name directly references a value, and a pointer indirectly references a value.

Pointer

- POINTER TYPE DECLARATION

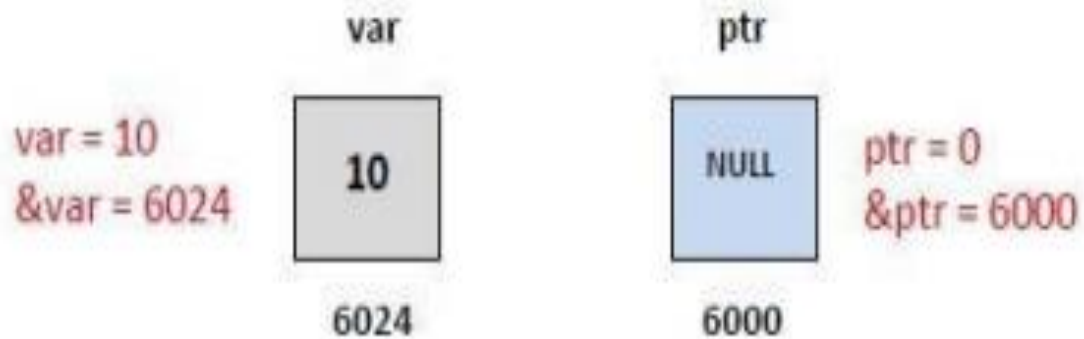
SYNTAX: type * variable;

EXAMPLE: int *ptr;

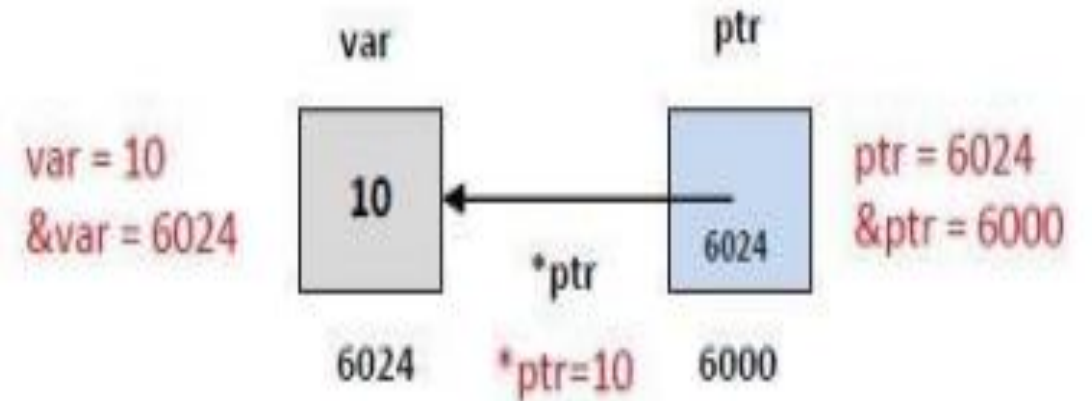
Interpretation: The value of the pointer variable ptr is a memory address. A data item whose address is stored in this variable must be of the specified type.

Pointer Memory Representation

```
int *ptr = 0; or int *ptr = NULL;  
int var = 10;
```



```
ptr = &var;
```



Arithmetic in pointers

- A pointer in c is an address, which is a numeric value.
- Therefore, we can perform arithmetic operations on a pointer just as you can on a numeric value.
- A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.
- When an integer is added to or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.

Incrementing in pointers

```
#include <stdio.h>

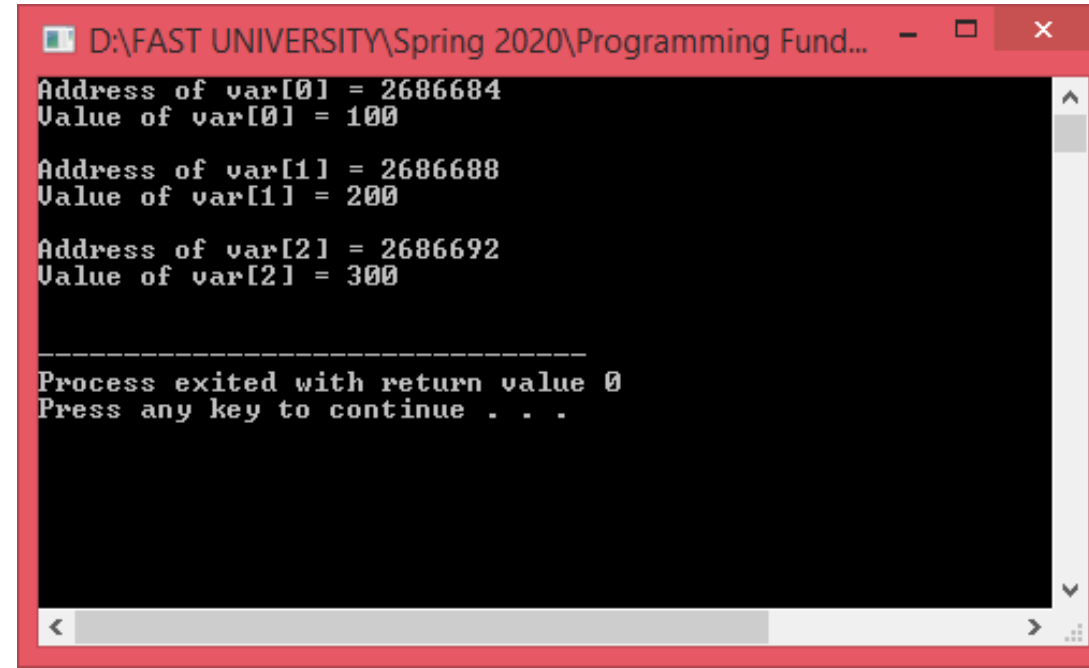
const int MAX = 3;

int main () {

    int var[] = {100, 200, 300};
    int i, *ptr;
    /* Let us have array address in pointer */
    ptr = var;

    for ( i = 0; i < MAX; i++) {

        printf("Address of var[%d] = %u\n", i, ptr );
        printf("Value of var[%d] = %d\n\n", i, *ptr );
        /* move to the next location */
        ptr++;
    }
    return 0;
}
```



```
D:\FAST UNIVERSITY\Spring 2020\Programming Fund...
Address of var[0] = 2686684
Value of var[0] = 100

Address of var[1] = 2686688
Value of var[1] = 200

Address of var[2] = 2686692
Value of var[2] = 300

-----
Process exited with return value 0
Press any key to continue . . .
```

Decrementing in pointers

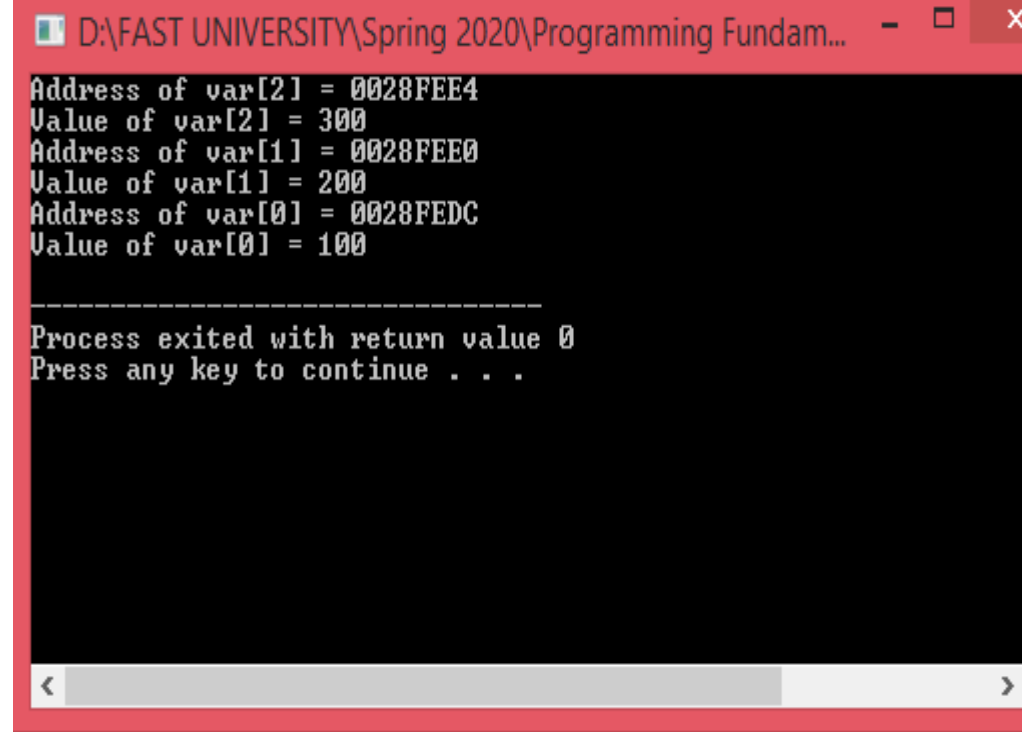
```
#include <stdio.h>
int main () {
    int var[] = {100, 200, 300};
    int i, *ptr;

    /* let us have array address in pointer */
    ptr = &var[2];

    for ( i = 3; i > 0; i--) {

        printf("Address of var[%d] = %x\n", i-1, ptr );
        printf("Value of var[%d] = %d\n", i-1, *ptr );

        /* move to the previous location */
        ptr--;
    }
    return 0;
}
```



```
D:\FAST UNIVERSITY\Spring 2020\Programming Fundam...
Address of var[2] = 0028FEE4
Value of var[2] = 300
Address of var[1] = 0028FEE0
Value of var[1] = 200
Address of var[0] = 0028FEDC
Value of var[0] = 100

-----
Process exited with return value 0
Press any key to continue . . .
```

Pointer – (Function Pass by value)

- In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations.
- So any changes made inside functions are not reflected in actual parameters of caller.

Pointer – (Function Pass by value)

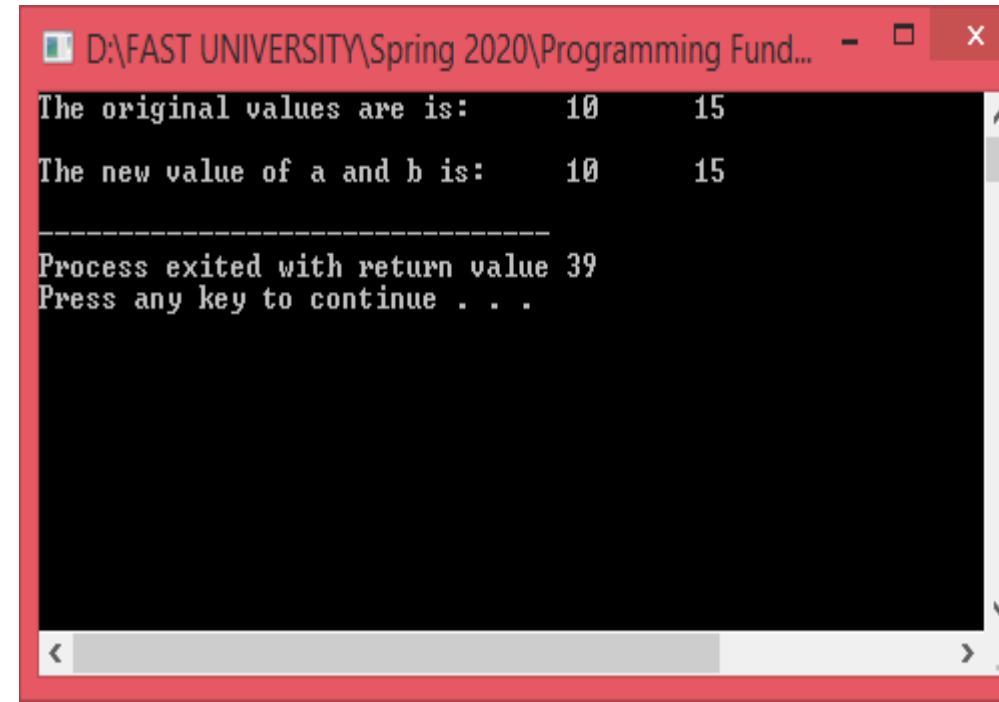
```
#include<stdio.h>

int main( void)
{
    int a=10;
    int b=15; // initialize number

    printf( "The original values are is:\t %d \t %d\n", a, b );
    swap(a,b); // function calling

    printf( "\nThe new value of a and b is:\t %d \t %d\n", a, b );
} // end main

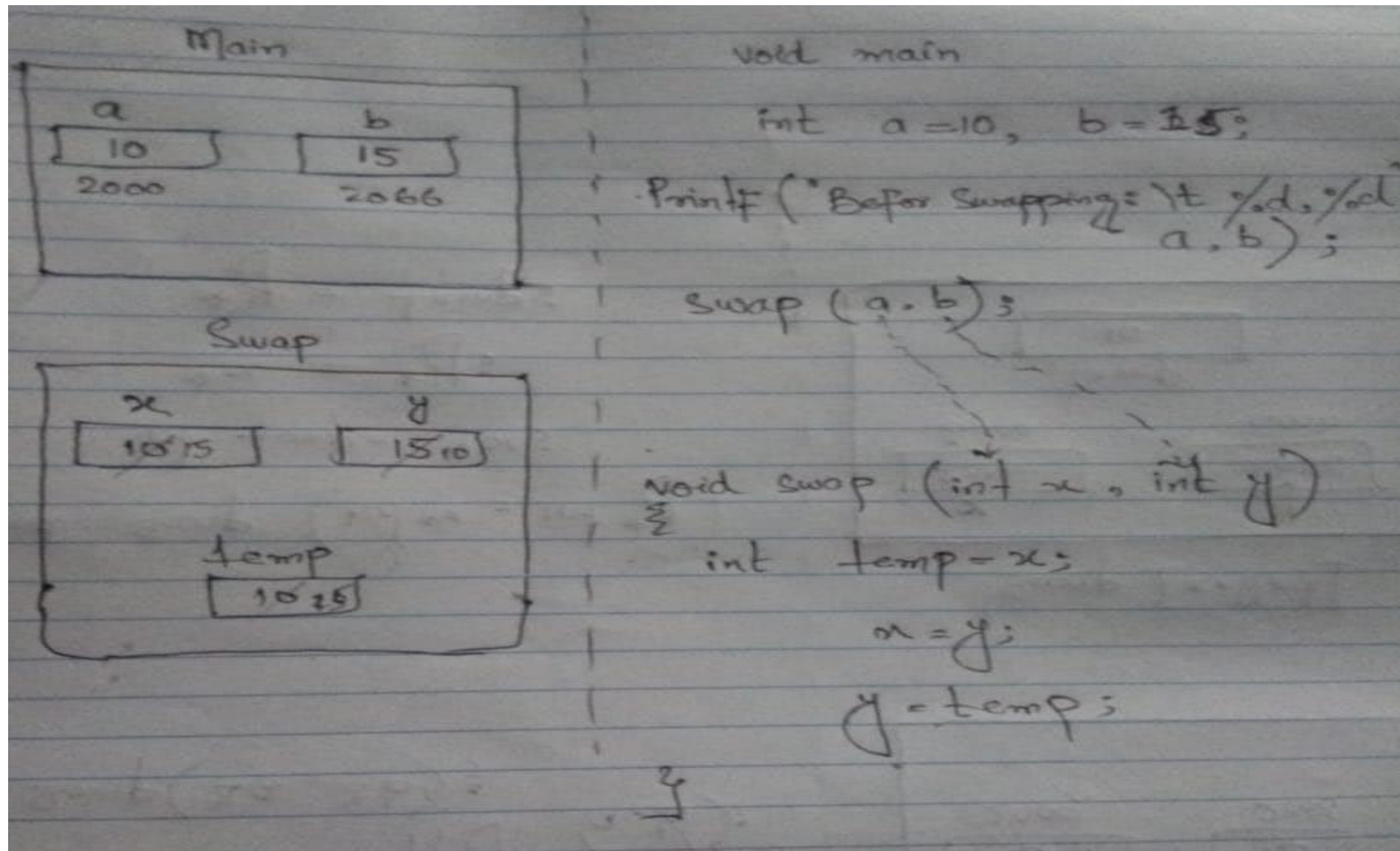
void swap( int x, int y )
{
    int temp=x;
    x=y;
    y=temp;
    //printf("After swaping: \t %d \t %d", x, y);
}
```



```
D:\FAST UNIVERSITY\Spring 2020\Programming Fund...
The original values are is:      10      15
The new value of a and b is:    10      15

-----
Process exited with return value 39
Press any key to continue . . .
```

Pointer – (Function Pass by value)



Pointer – (Function Pass by reference)

- Both the actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.
- In this method, the address of actual variables in the calling function are copied into the formal variables of the called function.

Pointer – (Function Pass by reference)

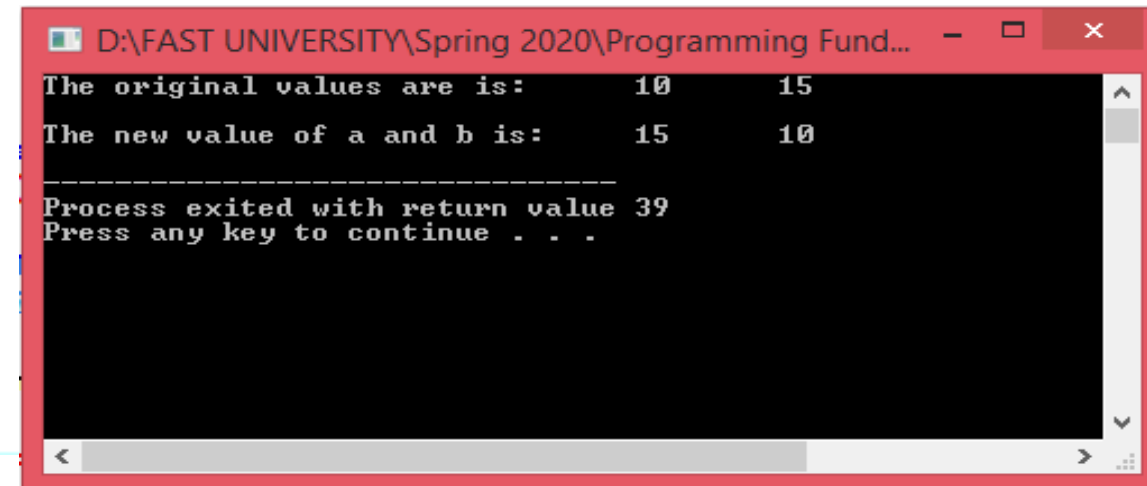
```
#include<stdio.h>

int main( void)
{
    int a=10;
    int b=15; // initialize number

    printf( "The original values are is:\t %d \t %d\n", a, b );
        swap(&a,&b); // function calling

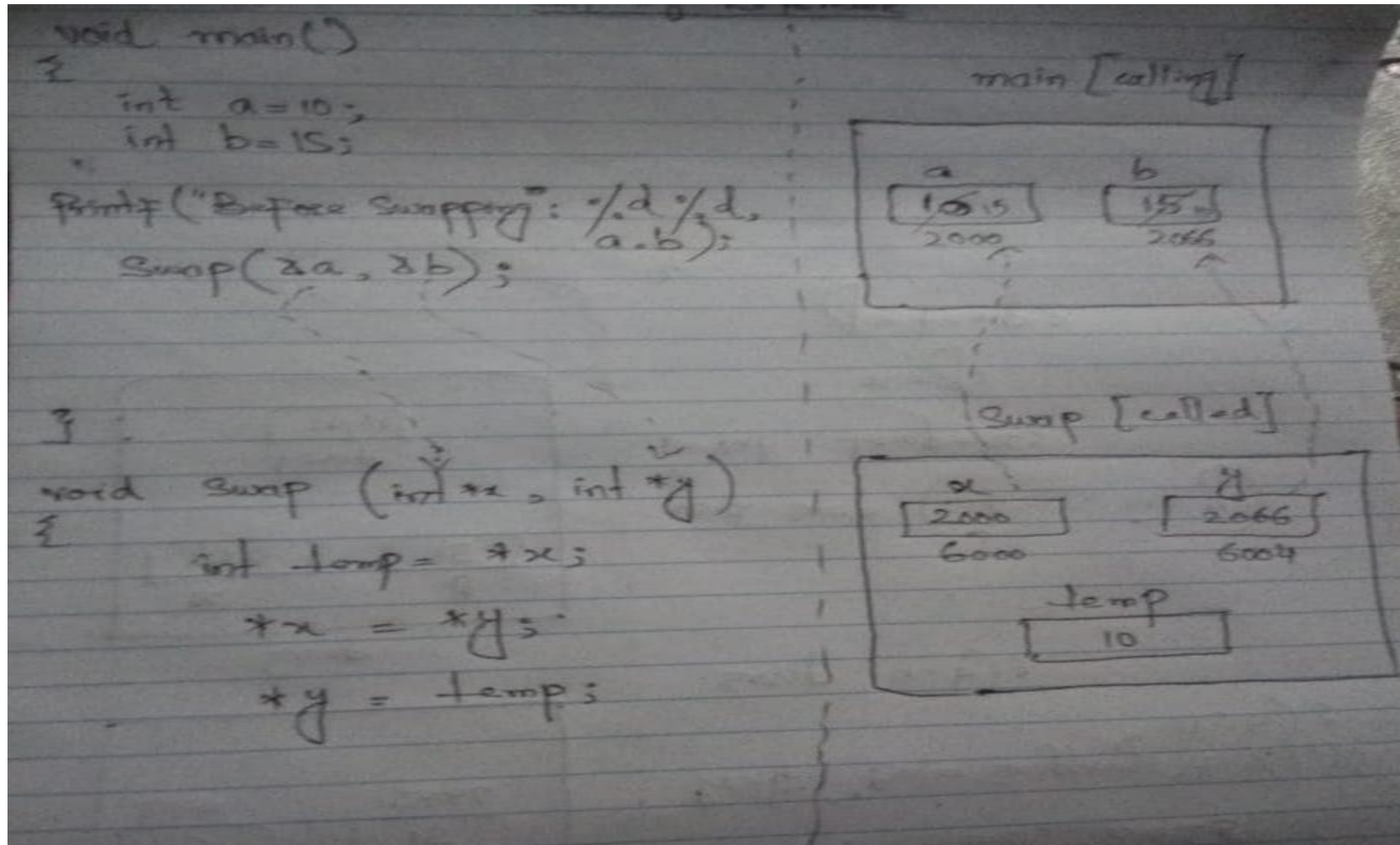
    printf( "\nThe new value of a and b is:\t %d \t %d\n", a, b );
} // end main

void swap( int *x, int *y )
{
    int temp= *x;
    *x = *y;
    *y = temp;
    //printf("After swaping: \t %d \t %d", x, y);
}
```



```
D:\FAST UNIVERSITY\Spring 2020\Programming Fund...
The original values are is:      10      15
The new value of a and b is:      15      10
-----
Process exited with return value 39
Press any key to continue . . .
```

Pointer – (Function Pass by reference)



Const Qualifier in C (Pointer)

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change the value of const variable by using pointer).

Pointer to constant.

Const Qualifier in C (Pointer)

- **Pointer to constant:**

We can change the pointer to point to any other integer variable, but cannot change the value of the object (entity) pointed using pointer ptr.

Pointer to constant can be declared in following two ways.

```
const int *ptr;
```

or

```
int const *ptr;
```

Const Qualifier in C (Pointer)

- **Pointer to constant**

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    /* ptr is pointer to constant */
    const int *ptr = &i;

    printf("ptr: %d\n", *ptr);

    /* error: object pointed cannot be modified
    using the pointer ptr */
    *ptr = 100;

    ptr = &j;           /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Const Qualifier in C (Pointer)

- **Constant pointer to variable.**

means we can change the value of object pointed by pointer, but cannot change the pointer to point another variable.

```
int *const ptr;
```

Const Qualifier in C (Pointer)

- **Constant pointer to variable.**

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;

    /* constant pointer to integer */
    int *const ptr = &i;

    printf("ptr: %d\n", *ptr);

    *ptr = 100;    /* valid */
    printf("ptr: %d\n", *ptr);

    ptr = &j;      /* error */

    return 0;
}
```

Const Qualifier in C (Pointer)

- **constant pointer to constant**

means we cannot change value pointed by the pointer as well as we cannot point the pointer to other variables.

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

Const Qualifier in C (Pointer)

- **constant pointer to constant**

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    /* constant pointer to constant integer */
    const int *const ptr = &i;

    printf("ptr: %d\n", *ptr);

    ptr = &j;      /* error */
    *ptr = 100;    /* error */

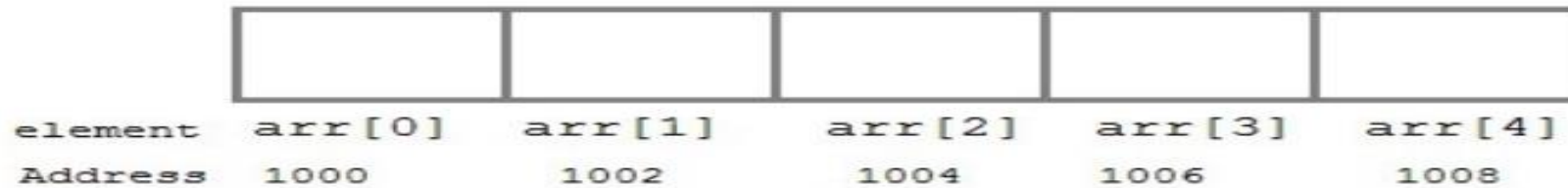
    return 0;
}
```

Pointer to an array

- Pointer to an array is also known as array pointer. We are using the pointer to access the elements of an array.
- When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array.
- Base address i.e address of the first element of the array is also allocated by the compiler.
- Suppose we declare an array arr,
 - `int arr[5] = {1, 2, 3, 4 ,5};`

Pointer to an array

Assuming that the base address of `arr` is 1000 and each integer requires two/four bytes, the five elements will be stored as follows:



Here variable `arr` will give the base address, which is a constant pointer pointing to the first element of the array, `arr[0]`. Hence `arr` contains the address of `arr[0]` i.e 1000. In short, `arr` has two purpose.:

it is the name of the array and it acts as a pointer pointing towards the first element in the array.

`arr` is equal to `&arr[0]` by default

Pointer to an array

We can also declare a pointer of type int to point to the array arr.

```
Int *p;
```

```
P=arr;
```

or

```
P=&arr[0];
```

Pointer to an array

Now we can access every element of the array arr using p++ to move from one element to another.

For example:

```
#include <stdio.h>

int main()
{
    int i;
    char a[4] = {'F', 'A', 'S', 'T'};
    char *p = a;      // same as char *p = &a[0]
    for (i = 0; i < 4; i++)
    {
        printf("%c\n", *p);
        p++;
    }

    return 0;
}
```


Pointer to an array

Now we can access every element of the array arr using p++ to move from one element to another.

For example:

```
#include <stdio.h>

int main()
{
    int i;
    char a[4] = {'F', 'A', 'S', 'T'};
    char *p = a;      // same as char *p = &a[0]
    for (i = 0; i < 4; i++)
    {
        printf("%c\n", *p);
        p++;
    }

    return 0;
}
```

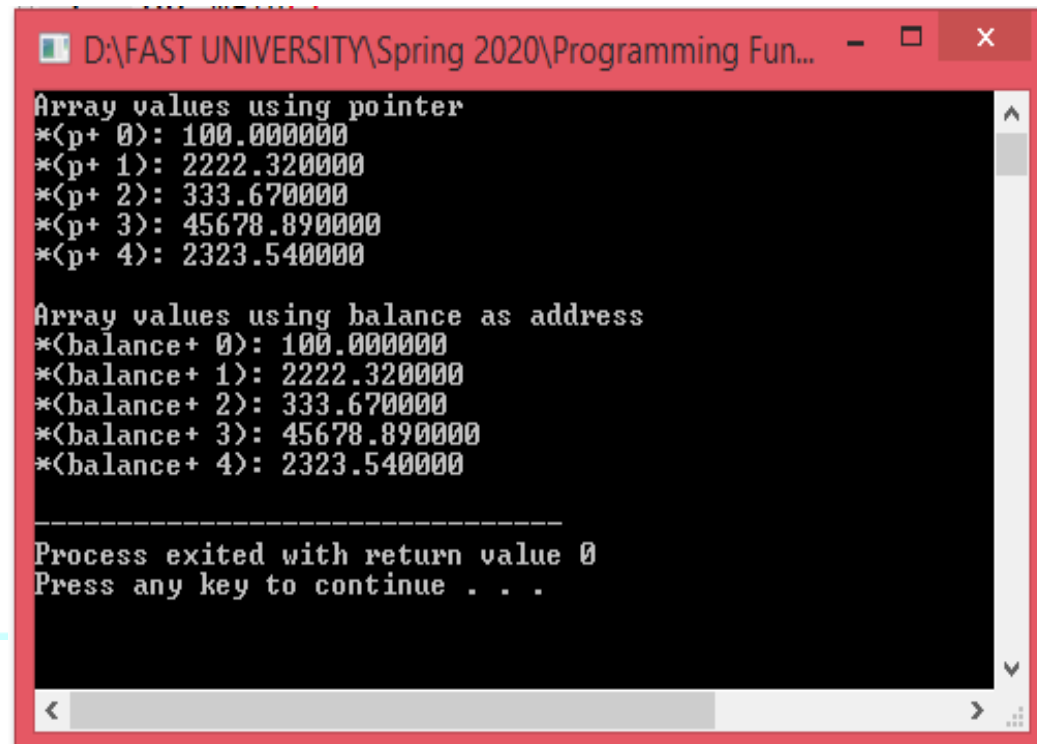
Pointer to an array

Example:

```
#include <stdio.h>
int main()
{
    double balance[5]={100.0, 2222.32, 333.67, 45678.89, 2323.54};
    double *p;
    int i;
    p=balance;

    printf("Array values using pointer\n");
    for (i = 0; i < 5; i++)
    {
        printf("*(p+ %d): %f\n", i, *(p+i));
    }

    printf("\nArray values using balance as address\n");
    for (i = 0; i < 5; i++)
    {
        printf("*(balance+ %d): %f\n", i, *(balance+i));
    }
    return 0;
}
```



```
D:\FAST UNIVERSITY\Spring 2020\Programming Fun...
Array values using pointer
*(p+ 0): 100.000000
*(p+ 1): 2222.320000
*(p+ 2): 333.670000
*(p+ 3): 45678.890000
*(p+ 4): 2323.540000

Array values using balance as address
*(balance+ 0): 100.000000
*(balance+ 1): 2222.320000
*(balance+ 2): 333.670000
*(balance+ 3): 45678.890000
*(balance+ 4): 2323.540000

-----
Process exited with return value 0
Press any key to continue . . .
```