# POINTERS IN C

## UNDERSTANDING THE CONCEPT

# Pointer Notation

- Consider the declaration,

```
int i = 3 ;
```

This declaration tells the C compiler to:

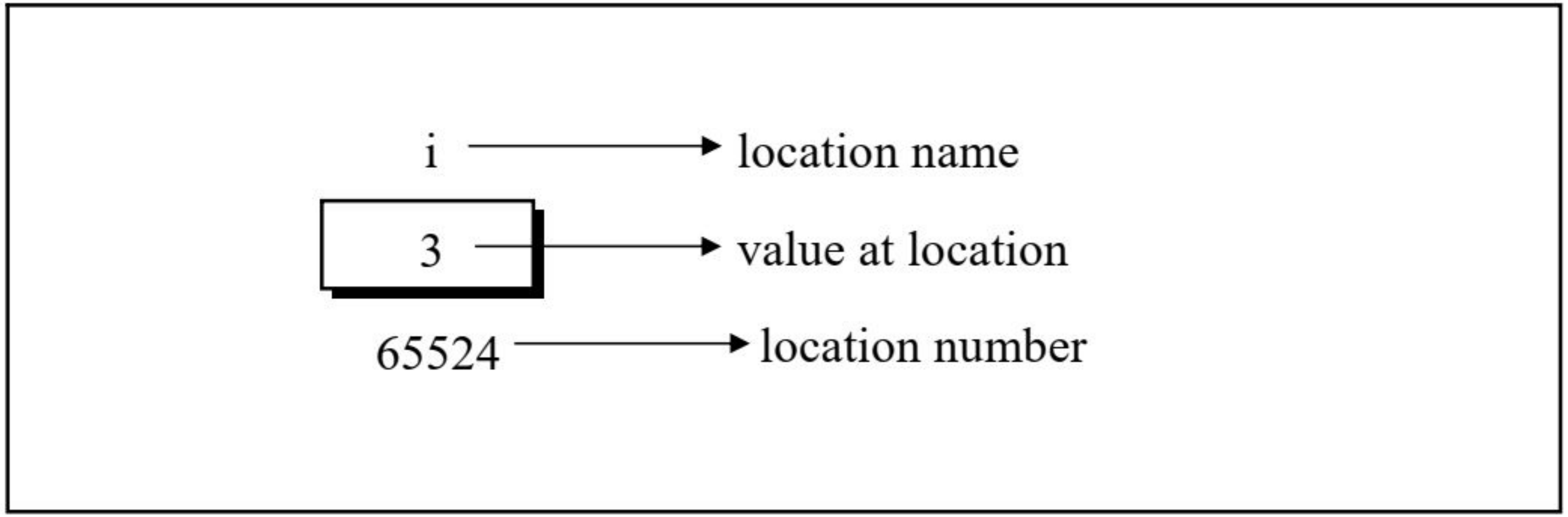
(a) Reserve space in memory to hold the integer value.

(b) Associate the name i with this memory location.

(c) Store the value 3 at this location.

# Pointer Notation

We may represent i's location in memory by the following memory map.



# Pointer Notation

- We see that the computer has selected memory location 65524 as the place to store the value 3.
- The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. (As memory is volatile in nature).
- The important point is, i’s address in memory is a number.

# Pointer Notation

- We can print this address number through the following program:

```
main( )
{
    int i = 3 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
}
```

- The output of the above program would be:
  - Address of i = 65524
  - Value of i = 3

# POINTER NOTATION

- The expression **&i** returns the address of the variable **i**.
- Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using %u, which is a format specifier for printing an unsigned integer.
- We have been using the ‘&’ operator all the time in the scanf( ) statement.

# Pointer Notation

- The other pointer operator available in C is '*', called '***value at address***' operator.

- It gives the value stored at a particular address.

- The 'value at address' operator is also called '***indirection***' operator.

# Pointer Notation

- Observe carefully the output of the following program:

```
main( )
{
    int i = 3 ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

- The output of the above program would be:
  - Address of i = 65524
  - Value of i = 3
  - Value of i = 3

# Pointer Notation

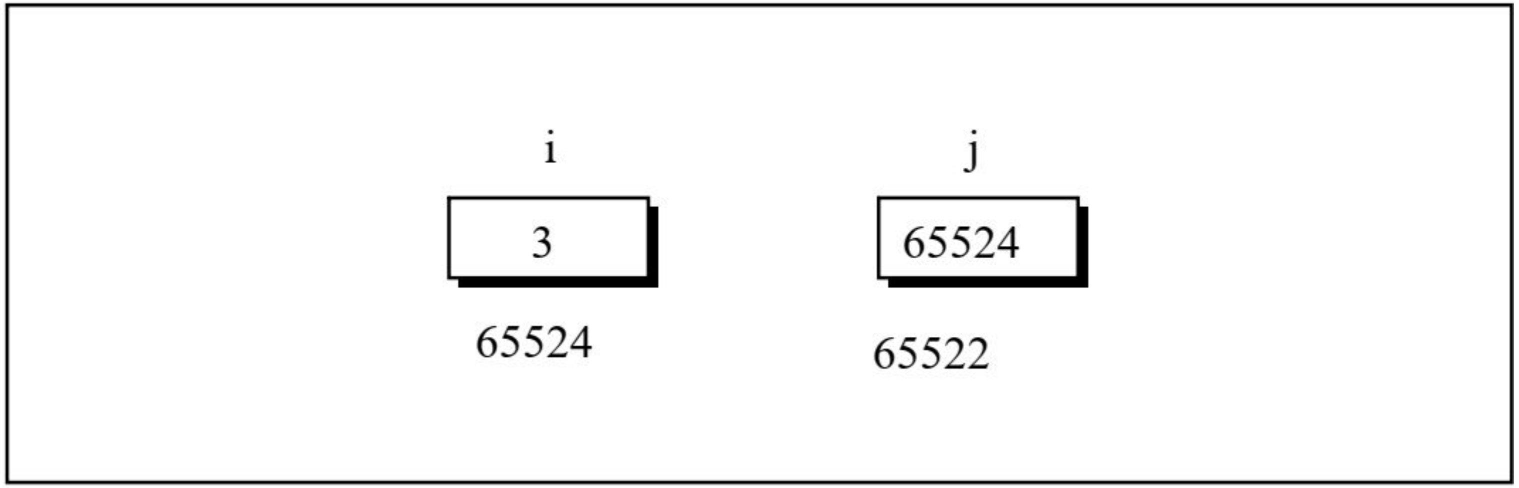
- Note that printing the value of ***( &i )** is the same as printing the value of **i**.
- The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying,

```
j = &i ;
```

- But remember that j is not an ordinary variable like any other integer variable. It is a variable that contains the address of another variable (i in this case).

# Pointer Notation

- Since j is a variable the compiler must provide it space in the memory.
- Once again, the following memory map would illustrate the contents of i and j.



# Pointer Notation

```
main( )
{
    int i = 3 ;
    int *j ;
    j = &i ;
    printf ( "\nAddress of i = %u",
    &i ) ;
    printf ( "\nAddress of i = %u", j
    ) ;
    printf ( "\nAddress of j = %u",
    &j ) ;
    printf ( "\nValue of j = %u", j )
    ;
    printf ( "\nValue of i = %d", i )
    ;
    printf ( "\nValue of i = %d",*(
    &i ) ) ;
```

- The output of the above program would be:
  - Address of i = 65524
  - Address of i = 65524
  - Address of j = 65522
  - Value of j = 65524
  - Value of i = 3
  - Value of i = 3
  - Value of i = 3

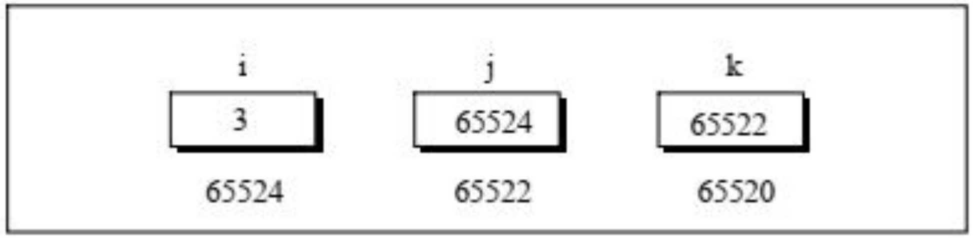
# Pointer Notation

- The concept of pointers can be further extended. Pointer, we know is a variable that contains address of another variable.
- Now this variable itself might be another pointer.
- Thus, we now have a pointer that contains another pointer's address.

# Pointer Notation

```
main( )
{
    int i = 3, *j, **k ;
    j = &i ;
    k = &j ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of i = %u", *k ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nAddress of j = %u", k ) ;
    printf ( "\nAddress of k = %u", &k ) ;
    printf ( "\nValue of j = %u", j ) ;
    printf ( "\nValue of k = %u", k ) ;
    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of i = %d", * ( &i )
    ) ;
    printf ( "\nValue of i = %d", *j ) ;
    printf ( "\nValue of i = %d", **k ) ;
}
```

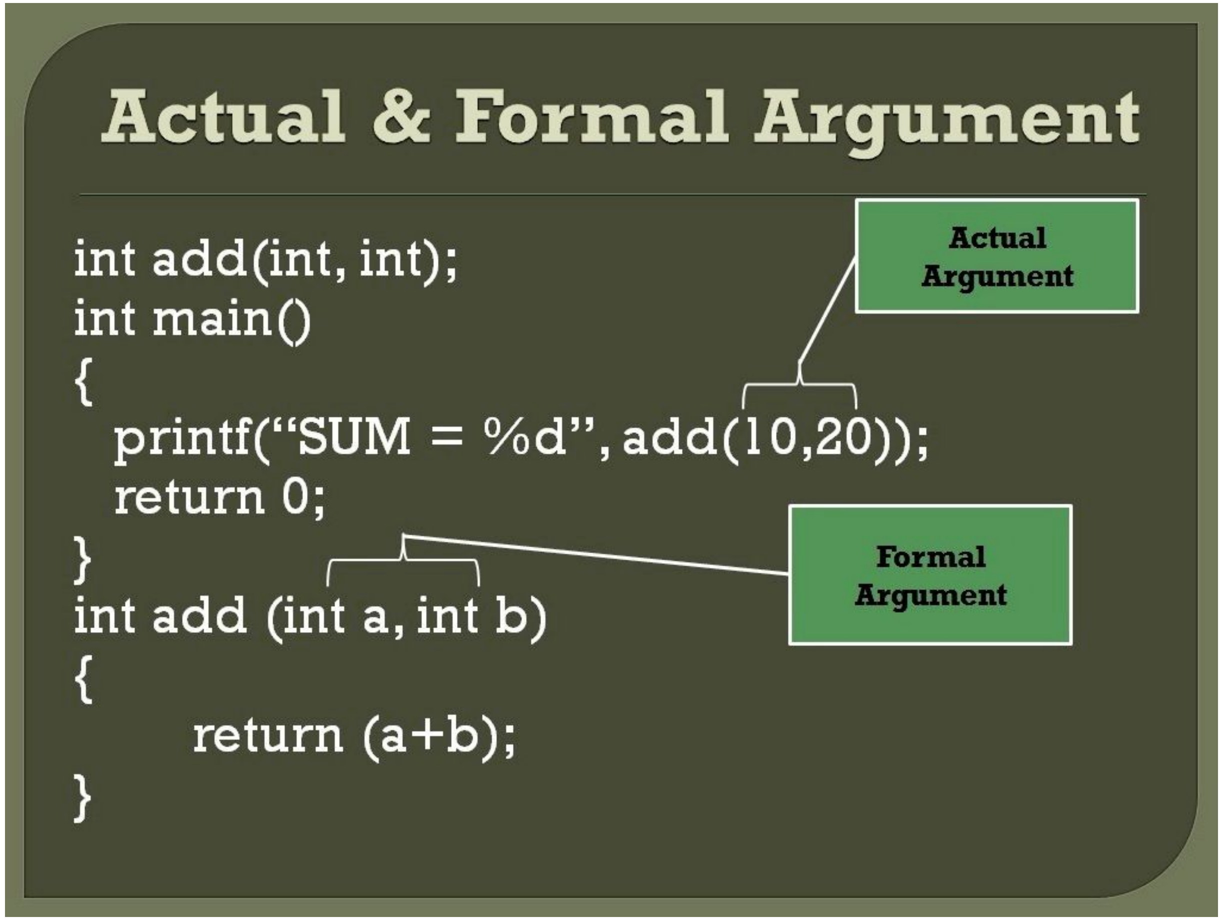
- The output of the above program would be:
    - Address of i = 65524
    - Address of i = 65524
    - Address of i = 65524
    - Address of j = 65522
    - Address of j = 65522
    - Address of k = 65520
    - Value of j = 65524
    - Value of k = 65522
    - Value of i = 3
    - Value of i = 3
    - Value of i = 3
    - Value of i = 3



# Function Calls

- Arguments can generally be passed to functions in one of the two ways:
  - sending the values of the arguments (Call-by-Value)
  - sending the addresses of the arguments (Call-by-Reference)
- In the first method, the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.
- With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function.

# Actual vs Formal Arguments



# Call by Value

```
main( )
{
    int a = 10, b = 20 ;
    swapv ( a, b ) ;
    printf ( "\na = %d b = %d", a, b
    ) ;
}
swapv ( int x, int y )
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
    printf ( "\nx = %d y = %d", x, y
    ) ;
}
```

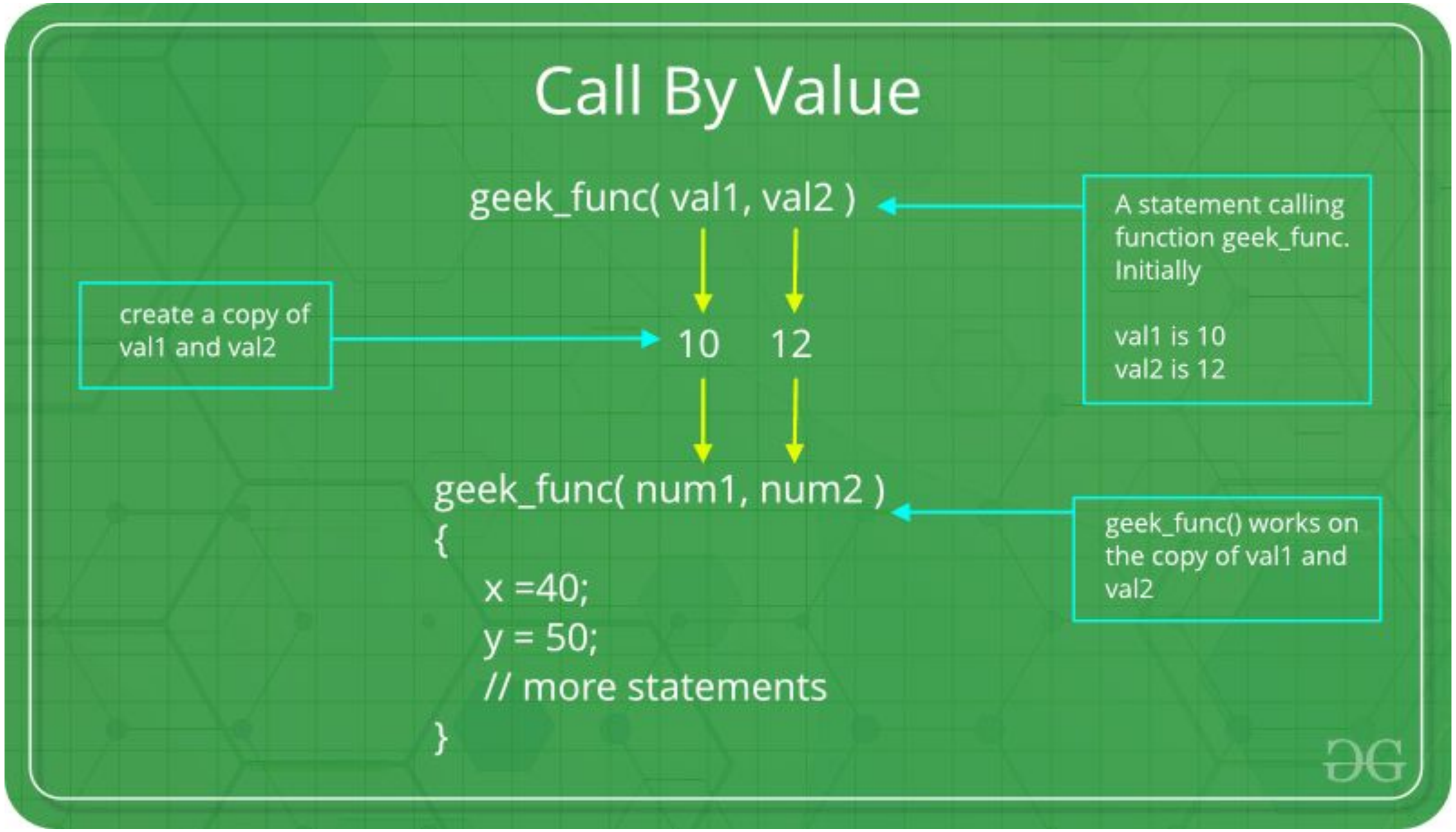
- The output of the above program would be:

  x = 20 y = 10

  a = 10 b = 20

- Note that values of a and b remain unchanged even after exchanging the valu.es of x and y.

# Call by Value



# Call by Reference

- In the second method (call by reference) the addresses of actual arguments in the calling function are copied into formal arguments of the called function.

- This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them.

# Call by Reference

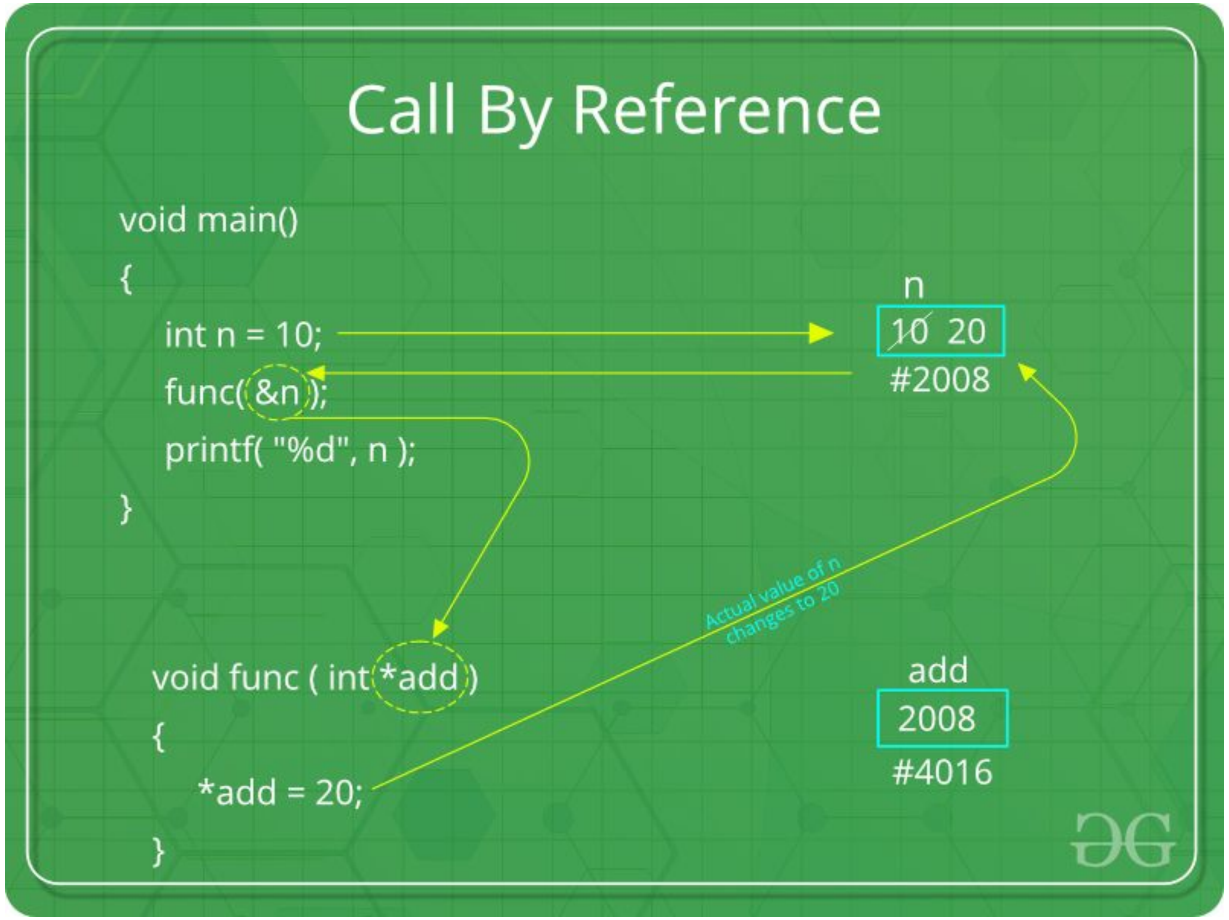
```
main( )
{
    int a = 10, b = 20 ;
    swapr( &a, &b ) ;
    printf("\na = %d b = %d",a,b);
}
  swapr( int *x, int *y )
  {
      int t ;
      t = *x ;
      *x = *y ;
      *y = t ;
  }
```

- The output of the above program would be:

  a = 20 b = 10

- Note that this program manages to exchange the values of a and b using their addresses stored in x and y.
- Usually in C programming we make a call by value. This means that in general, you cannot alter the actual arguments.
- But if desired, it can always be achieved through a call by reference.

# Call by Reference



# Thank You