# Assignment Theory

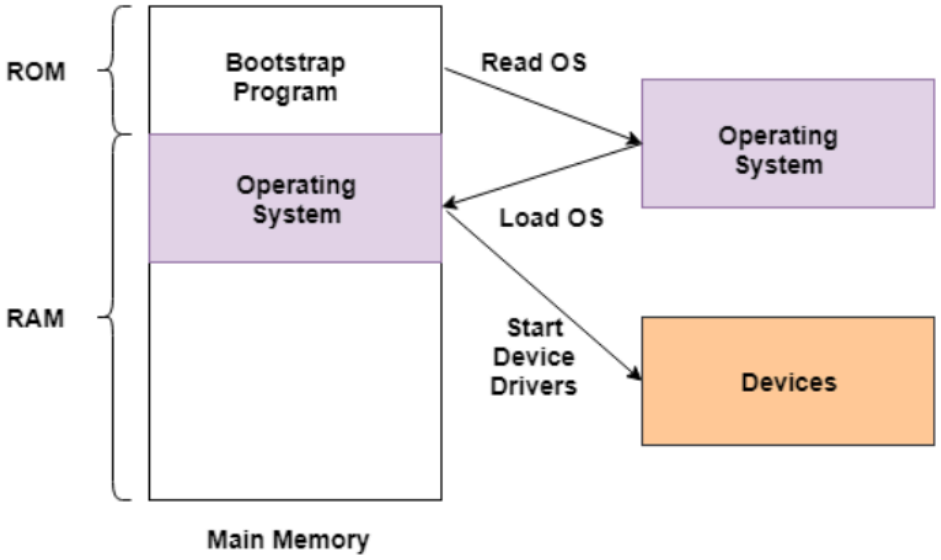


# Introduction Bootstrapping Process

## Bootstrap Program

A bootstrap program is the first code that is executed when the computer system is started. The entire operating system depends on the bootstrap program to work correctly as it loads the operating system.

A figure that demonstrates the use of the bootstrap program is as follows −




In the above image, the bootstrap program is a part of ROM which is the non-volatile memory. The operating system is loaded into the RAM by the bootstrap program after the start of the computer system. Then the operating system starts the device drivers.

## Bootstrapping Process

The bootstrapping process helps a computer start without needing anything extra. It works in steps, with simple programs loading bigger, more complex ones. First, the computer runs tests and starts up its basic settings (like BIOS). Then, it loads important programs like the operating system. Each step helps the computer get more powerful, all by itself.

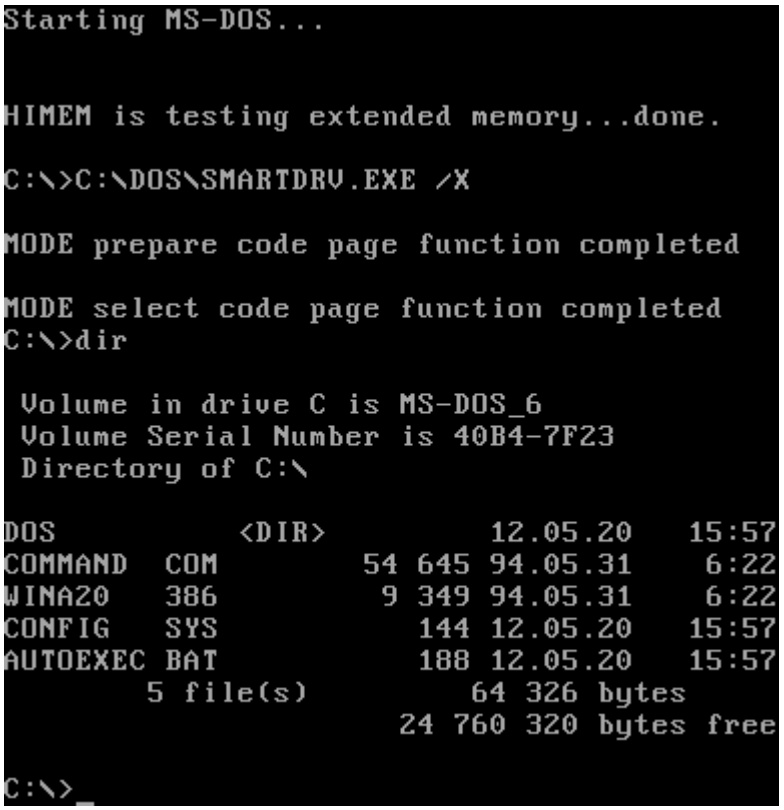
## Benefits of Bootstrapping

- **Efficient Downloads**: Only necessary software components are downloaded, avoiding unnecessary ones.
- **Memory Space Savings**: Frees up memory by excluding extraneous components.
- **Time Saving**: Reduces download time by focusing only on required components.

# Introduction to DOS (Disk Operating System):

DOS (Disk Operating System) is one of the oldest types of operating systems. It is a **Command-Line Interface (CUI)** operating system, meaning you interact with it by typing commands rather than using a graphical interface. DOS is a single-task, single-user system that works based on commands.

When you give DOS a command, it performs the requested action. This is why it is also called the **Command Prompt**.

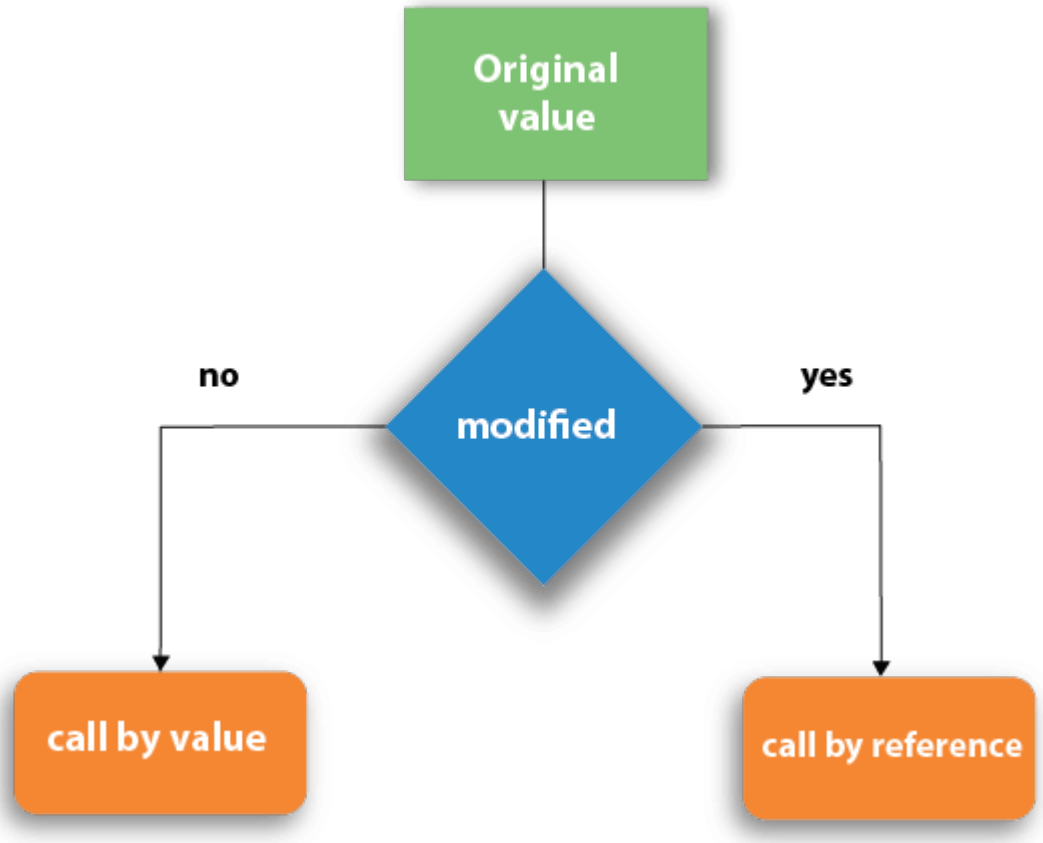



## Types of Commands in DOS:

1. **Internal Commands:** These commands are built into DOS and are already stored in the "Command.Com" file. They are always available to use. Some examples include:
    - CLS (clear the screen)
    - VOL (display disk volume label)
    - TIME (set the system time)
    - DATE (set the system date)
    - COPY (copy files)
2. **External Commands:** These commands are not built into the DOS system but are stored in separate files. They are more complex and can be run when needed. Examples include:
    - TREE (display folder structure)
    - FORMAT (prepare a disk for use)
    - MODE (adjust device settings)

# Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.




Let's understand call by value and call by reference in c language one by one.

## Call by Value in C

In the **call by value** method, the value of the actual parameters is copied into the formal parameters. This means that the function operates on a copy of the actual argument, and any changes made to the formal parameters do not affect the original values of the actual parameters.

- The value of the variable is passed to the function.

- The actual parameter (used in the function call) and the formal parameter (used in the function definition) occupy different memory locations.
- Modifying the formal parameter does not change the actual parameter's value.

**Example of Call by Value:**

```
#include<stdio.h>

void change(int num) {
     printf("Before adding value inside function num = %d
\n", num);
     num = num + 100;
     printf("After adding value inside function num = %d \n",
num);
}

int main() {
     int x = 100;
     printf("Before function call x = %d \n", x);
     change(x); // Passing value to function
     printf("After function call x = %d \n", x);
     return 0;
}
```

**Output**

```
Before function call x = 100
Before adding value inside function num = 100
After adding value inside function num = 200
After function call x = 100
```

In this example, the value of x remains unchanged in the main function after the call, even though it is modified inside the change( ) function. This is because the value of x was copied to the formal parameter num, and changes were made to num only.

## Call by Reference in C

In **call by reference**, instead of passing the value of the variable to the function, the **address** (or reference) of the variable is passed. This allows the function to modify the value of the actual parameters directly.

- The function works with the memory address of the actual parameters.
- The formal parameters are linked to the actual parameters through their addresses, so any changes made to the formal parameters directly affect the actual parameters.
- The memory allocated for both formal and actual parameters is the same.

In this case, modifying the formal parameters changes the value of the actual parameters, as both refer to the same memory location.

**Example of call by reference,:**

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d
\n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n",
*num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
```

```
return 0;
}
```

**Output:**

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

In this example, the value of x remains unchanged in the main function after the call, even though it is modified inside the `change()` function. This is because the value of x was copied to the formal parameter `num`, and changes were made to `num` only.

## Difference between call by value and call by reference in c

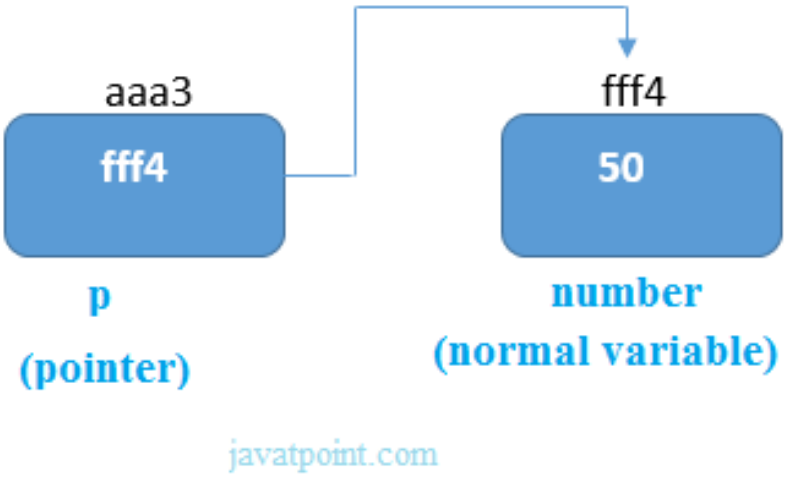
| No | Call by value | Call by reference |
|---|---|---|
| 1 | A copy of the value is passed into the function | An address of value is passed into the function |
| 2 | Changes made inside the function are limited to the function only. The values of the actual parameters do not change by changing the formal parameters. | Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters. |
| 3 | Actual and formal arguments are created at the different memory location | Actual and formal arguments are created at the same memory location |

# C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

## Pointer Example

An example of using pointers to print the address and value is given below.




As you can see in the above figure, the pointer variable stores the address of the number variable, i.e., fff4. The value of the number variable is 50. But the address of the pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of the pointer variable p.

Let's see the pointer example as explained in the above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
```

```
printf("Address of p variable is %x \n",p); // p contains the address of the number therefore printing p gives the address of number.
printf("Value of p variable is %d \n",*p); // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
return 0;
}
```

**Output**

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```