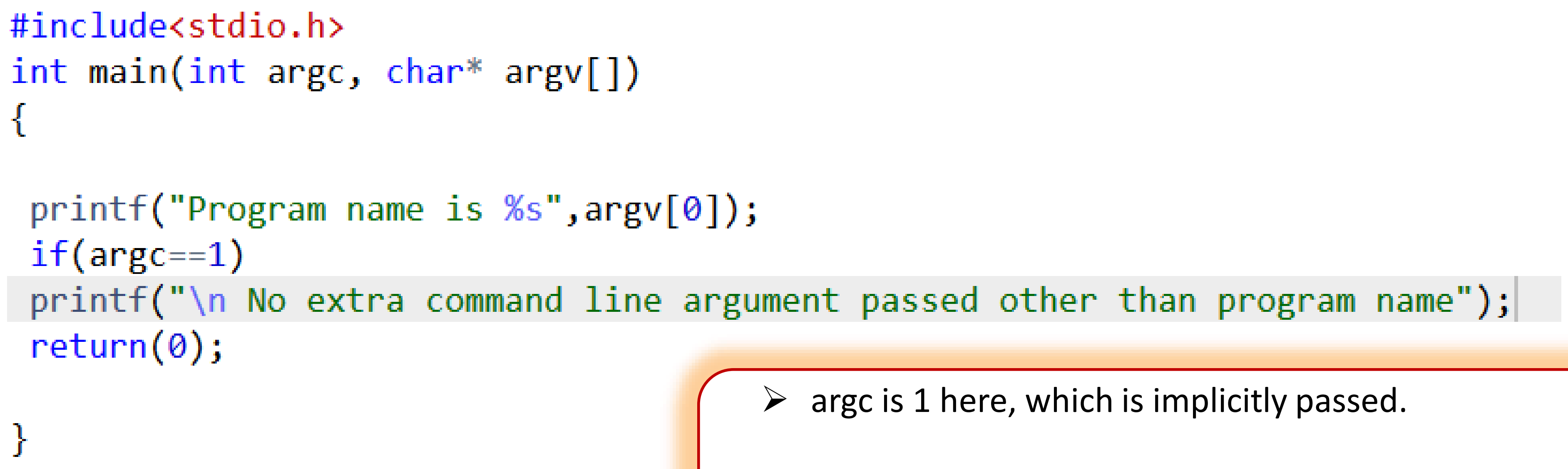
Command Line Arguments

- It is possible to pass some values from the command line to your C programs, when they are executed.

- These values are called **command line arguments.**

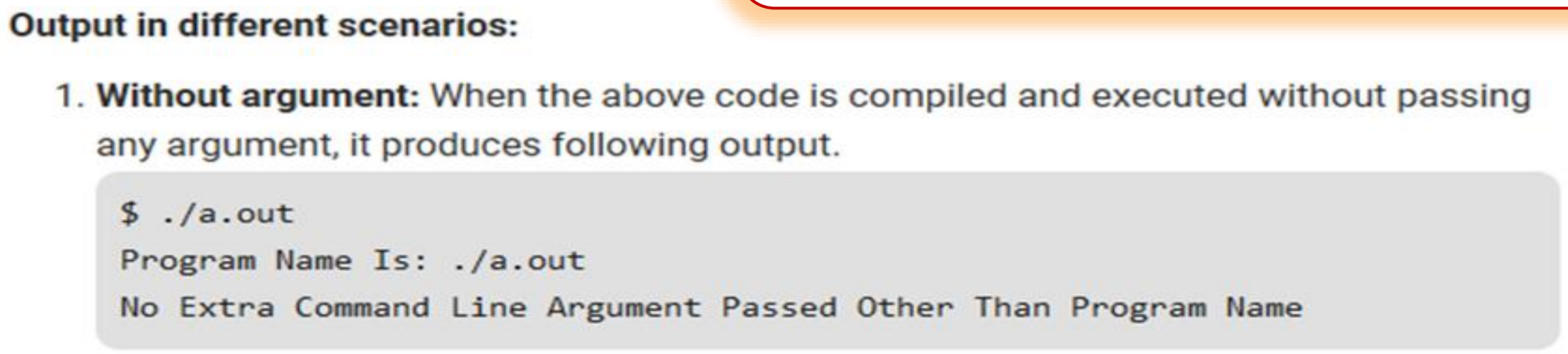
- The command line arguments are handled using main() function arguments **argc** and **argv[].**

- **argc** refers to the number of arguments passed and **argv[]** is a pointer array which points to each argument passed to the program.



```c
#include<stdio.h>
int main(int argc, char* argv[])
{

 printf("Program name is %s",argv[0]);
 if(argc==1)
 printf("\n No extra command line argument passed other than program name");
 return(0);

}
```

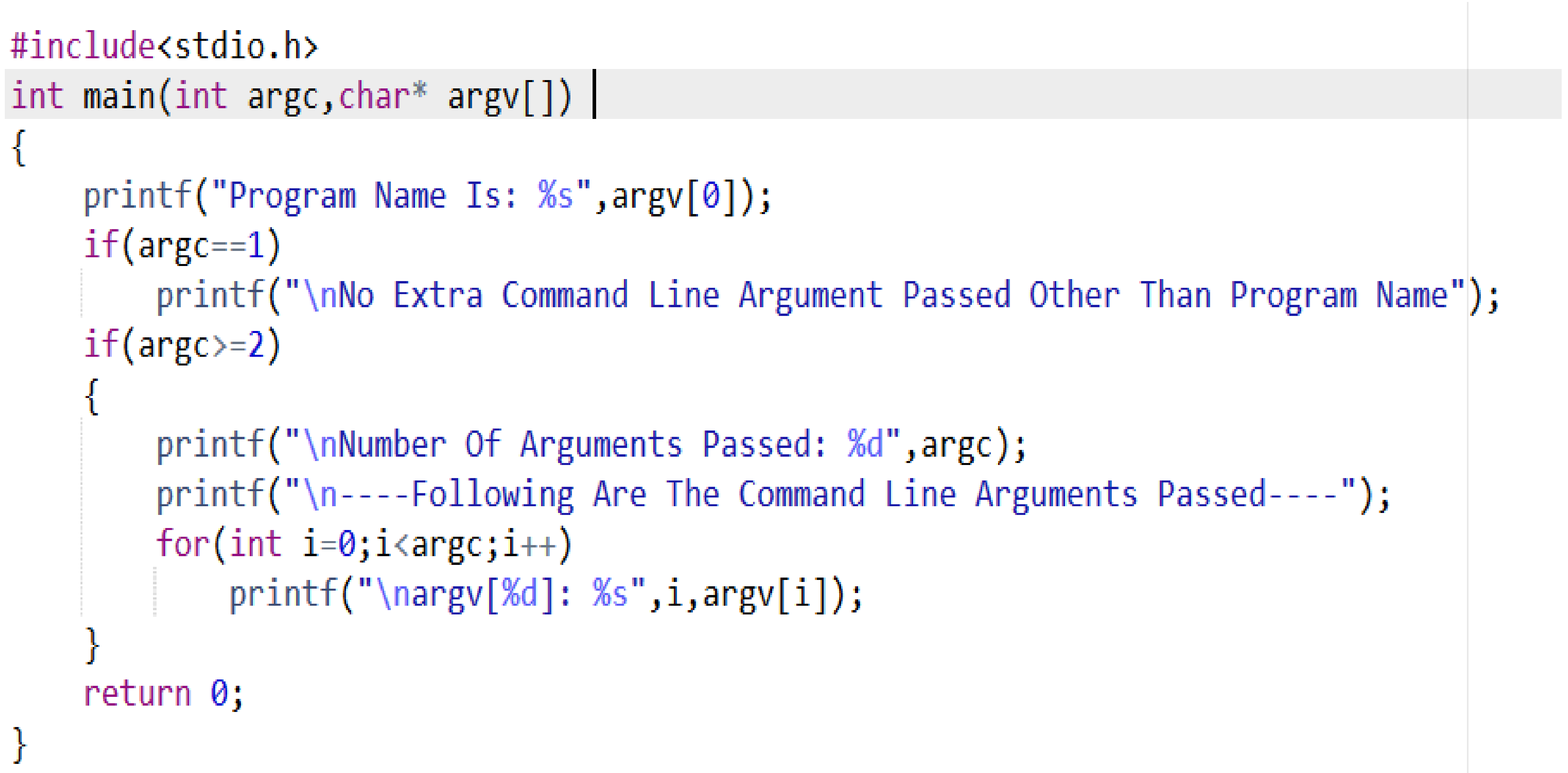
- argc is 1 here, which is implicitly passed.
- The argument passed is the program name which is ./a.out.



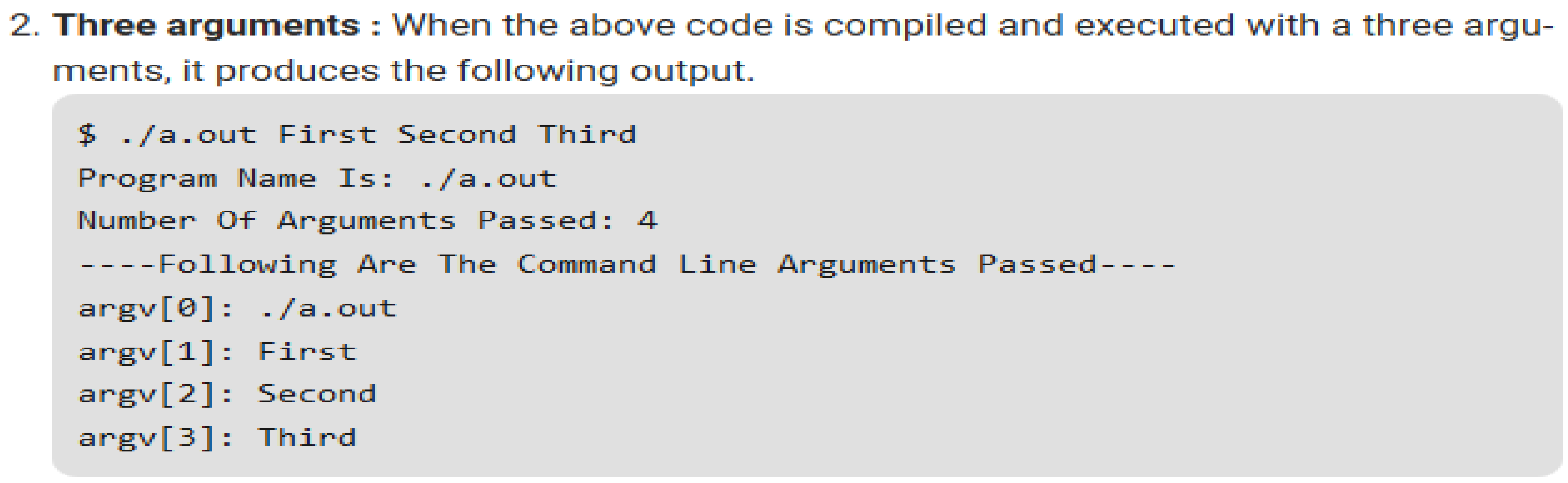
Output in different scenarios:

1. **Without argument:** When the above code is compiled and executed without passing any argument, it produces following output.

```
$ ./a.out
Program Name Is: ./a.out
No Extra Command Line Argument Passed Other Than Program Name
```



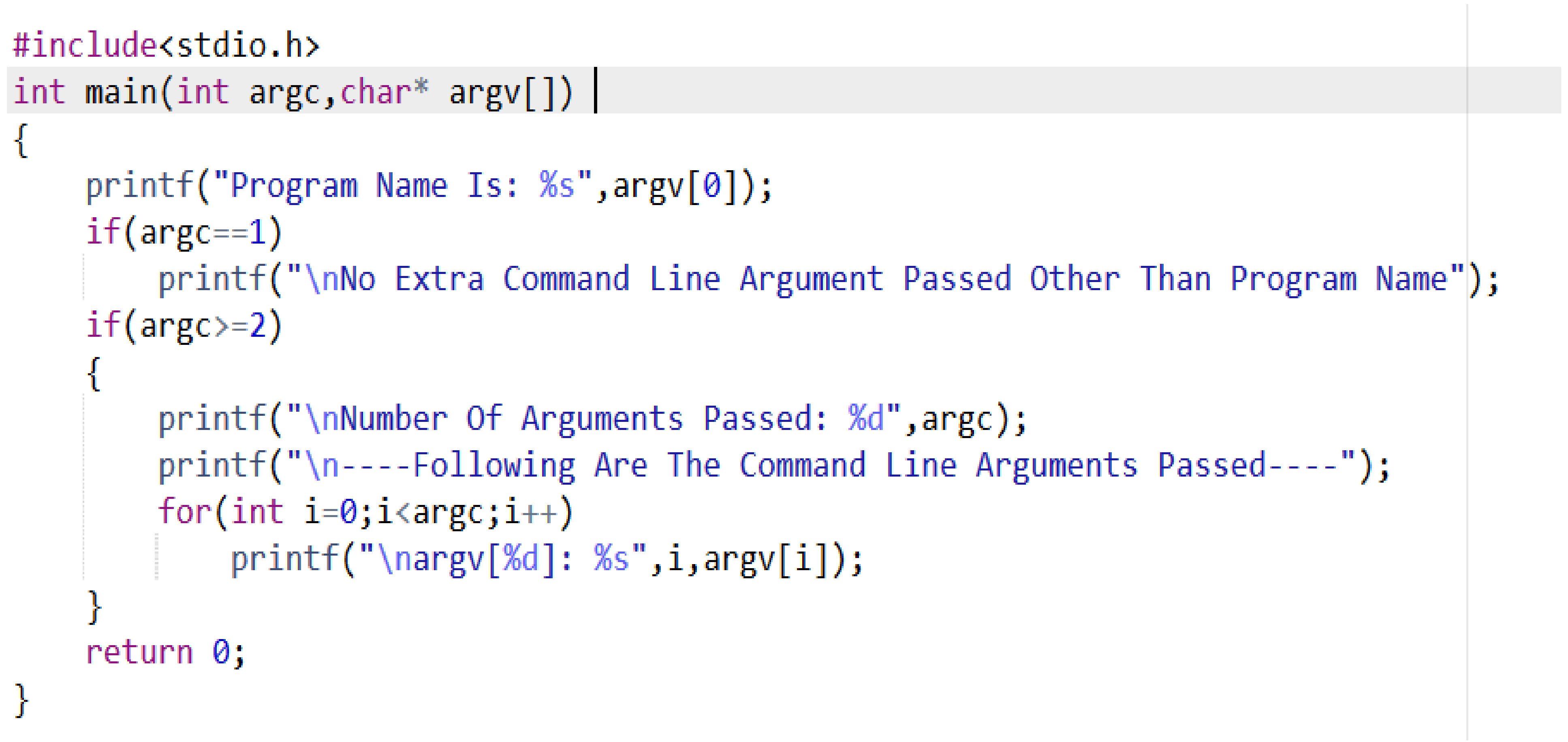
```c
#include<stdio.h>
int main(int argc,char* argv[])
{
    printf("Program Name Is: %s",argv[0]);
    if(argc==1)
        printf("\nNo Extra Command Line Argument Passed Other Than Program Name");
    if(argc>=2)
    {
        printf("\nNumber Of Arguments Passed: %d",argc);
        printf("\n----Following Are The Command Line Arguments Passed----");
        for(int i=0;i<argc;i++)
            printf("\nargv[%d]: %s",i,argv[i]);
    }
    return 0;
}
```



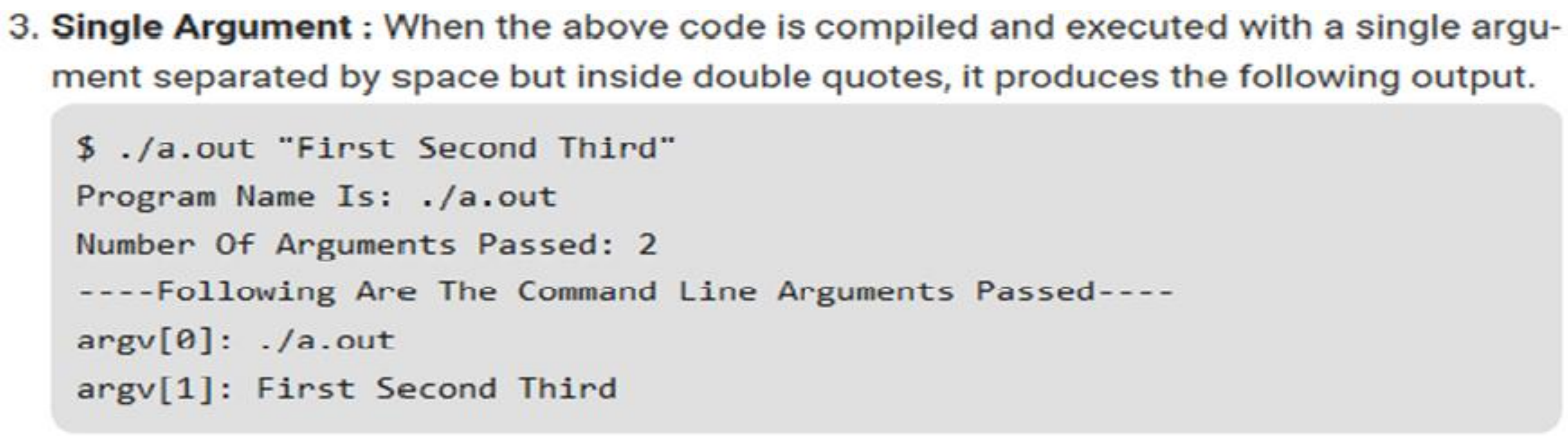
2. **Three arguments :** When the above code is compiled and executed with a three arguments, it produces the following output.

```
$ ./a.out First Second Third
Program Name Is: ./a.out
Number Of Arguments Passed: 4
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First
argv[2]: Second
argv[3]: Third
```

- argc is 4 here.
- Apart from the program name, there are three more arguments.



```c
#include<stdio.h>
int main(int argc,char* argv[])
{
    printf("Program Name Is: %s",argv[0]);
    if(argc==1)
        printf("\nNo Extra Command Line Argument Passed Other Than Program Name");
    if(argc>=2)
    {
        printf("\nNumber Of Arguments Passed: %d",argc);
        printf("\n----Following Are The Command Line Arguments Passed----");
        for(int i=0;i<argc;i++)
            printf("\nargv[%d]: %s",i,argv[i]);
    }
    return 0;
}
```



3. **Single Argument :** When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following output.

```
$ ./a.out "First Second Third"
Program Name Is: ./a.out
Number Of Arguments Passed: 2
----Following Are The Command Line Arguments Passed----
argv[0]: ./a.out
argv[1]: First Second Third
```

- Here, the second argument passed is “First Second Third”.
- Since it is enclosed in double quotes, it is considered as a single argument.
- Instead of double quotes, we can use single quotes also.

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

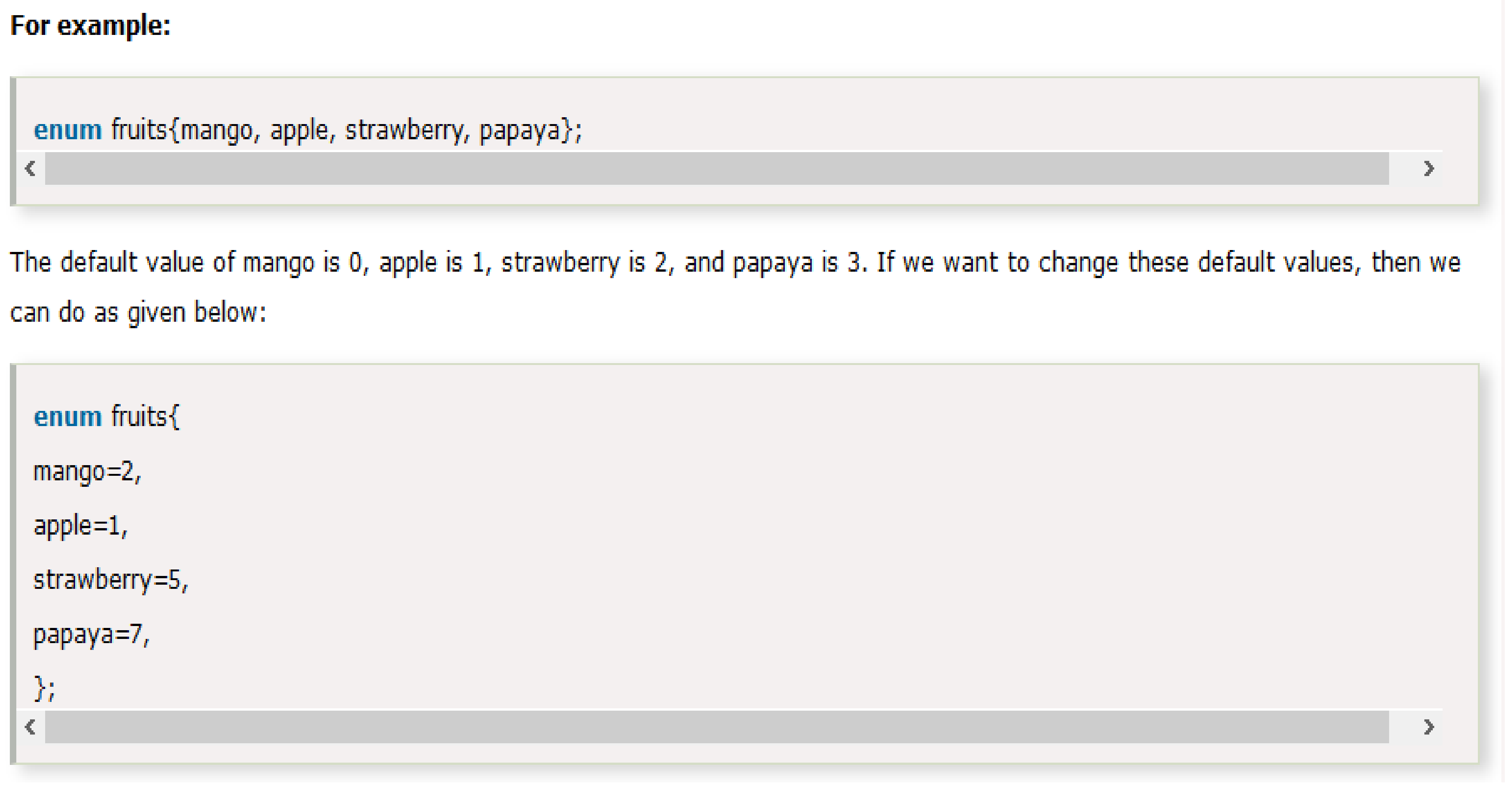
Enumeration (or enum) in C.

Enum in C

- The enum is also known as the enumerated type.
- It is a user defined data type.
- It consistes of integer values and it provides meaningful names to these values.
- The use of enum in C, makes the program easy to understand and maintain.
- The following is the syntax of enum:

  enum flag{ const1,const2….constN};

- The enum is named as flag containing 'N' integer constants. The default value of const1 is 0, const2 is 1 and so on.



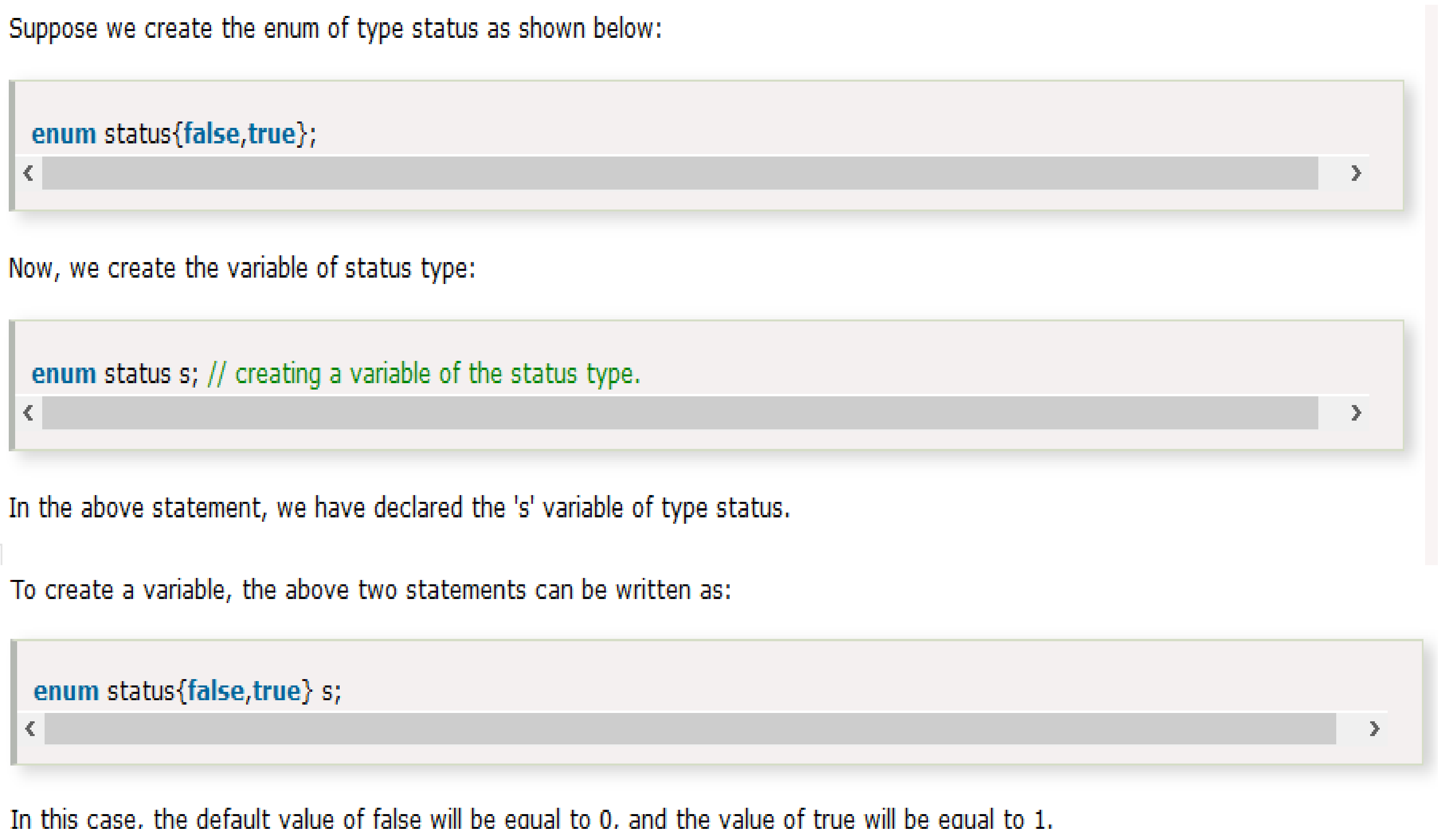
For example:

```
enum fruits{mango, apple, strawberry, papaya};
```

The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3. If we want to change these default values, then we can do as given below:

```
enum fruits{
mango=2,
apple=1,
strawberry=5,
papaya=7,
};
```

Enumerated type declaration



Suppose we create the enum of type status as shown below:

```
enum status{false,true};
```

Now, we create the variable of status type:

```
enum status s; // creating a variable of the status type.
```

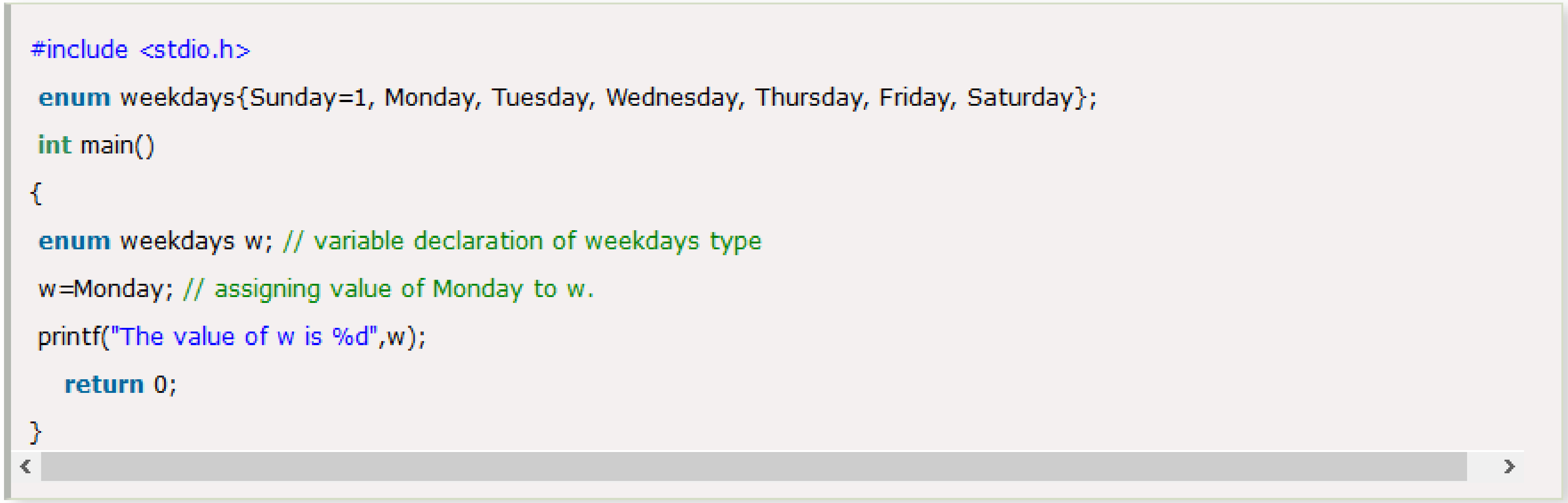
In the above statement, we have declared the 's' variable of type status.

To create a variable, the above two statements can be written as:

```
enum status{false,true} s;
```

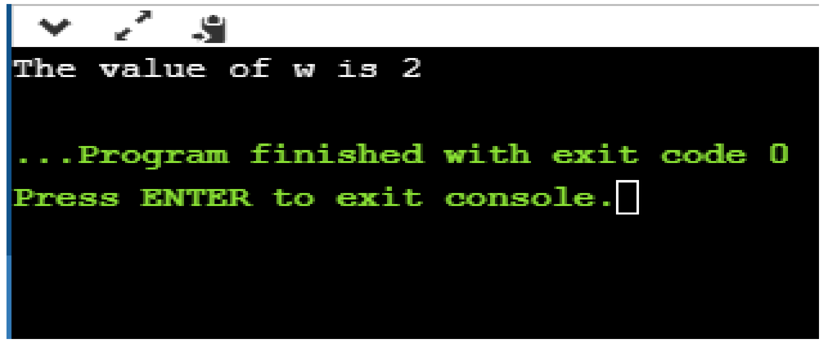
In this case, the default value of false will be equal to 0, and the value of true will be equal to 1.

Example: 1.



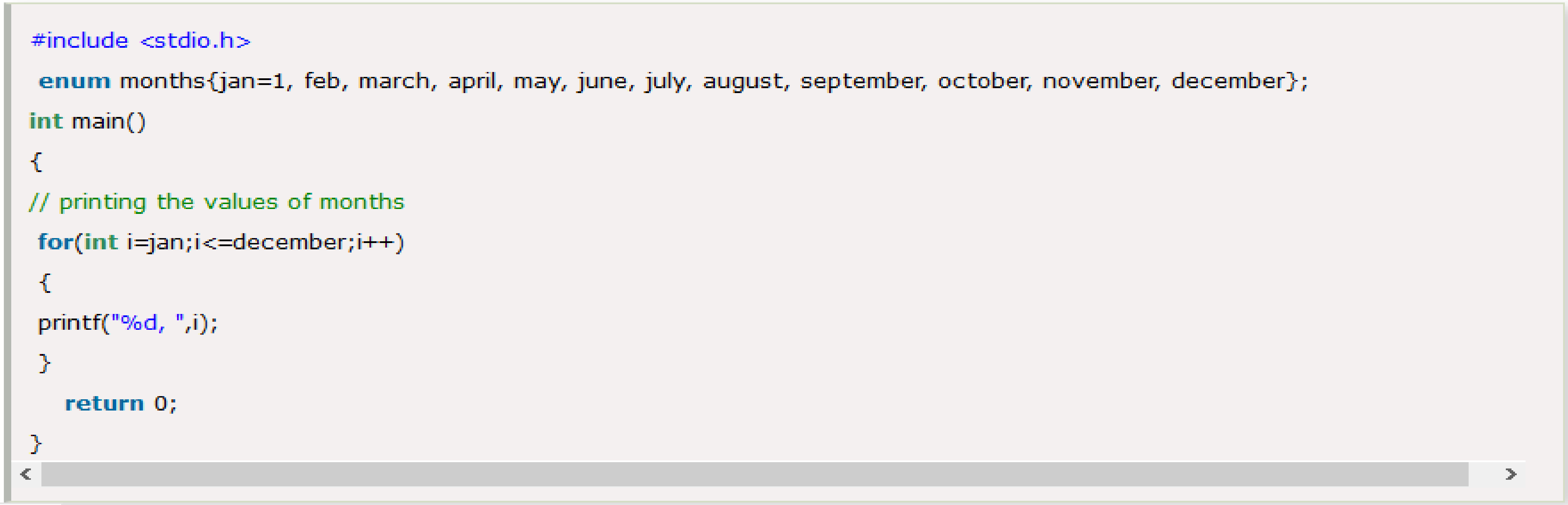
```
#include <stdio.h>
 enum weekdays{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
 int main()
{
 enum weekdays w; // variable declaration of weekdays type
 w=Monday; // assigning value of Monday to w.
 printf("The value of w is %d",w);
    return 0;
}
```

Output



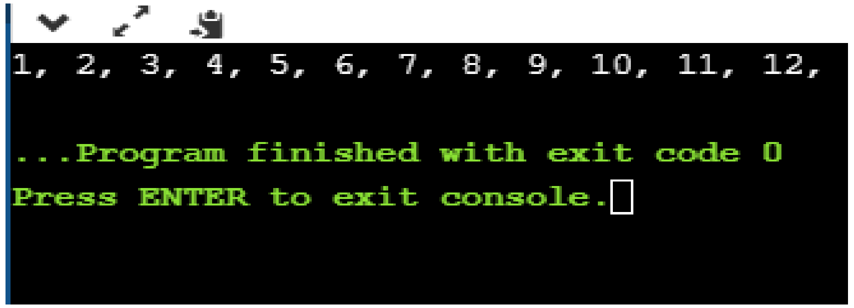
```
The value of w is 2

...Program finished with exit code 0
Press ENTER to exit console.
```

Example: 2.



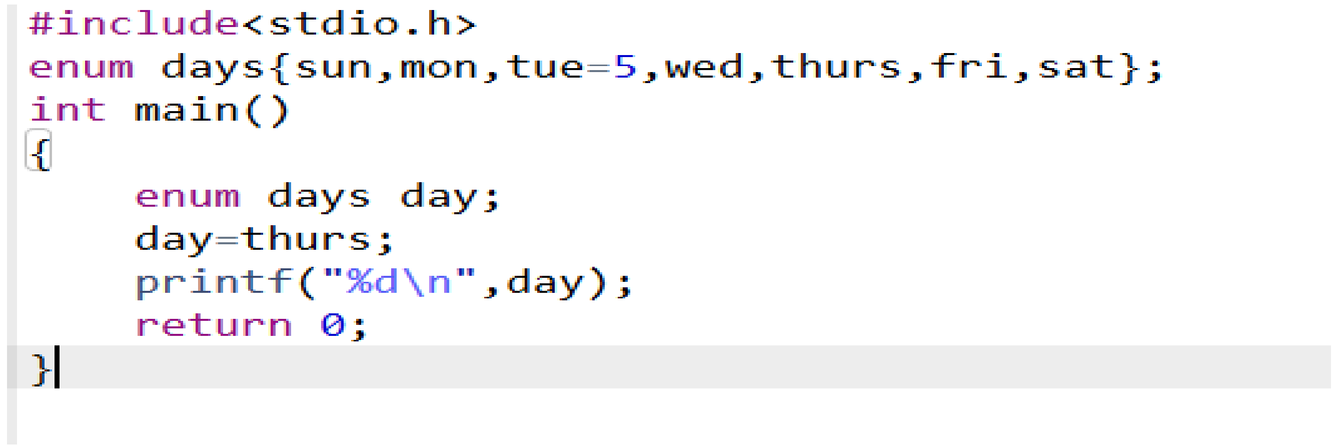
```
#include <stdio.h>
 enum months{jan=1, feb, march, april, may, june, july, august, september, october, november, december};
int main()
{
// printing the values of months
 for(int i=jan;i<=december;i++)
 {
 printf("%d, ",i);
 }
   return 0;
}
```

Output



```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,

...Program finished with exit code 0
Press ENTER to exit console.
```

Example:3.



```
#include<stdio.h>
enum days{sun,mon,tue=5,wed,thurs,fri,sat};
int main()
{
    enum days day;
    day=thurs;
    printf("%d\n",day);
    return 0;
}
```

Output

```
7
```

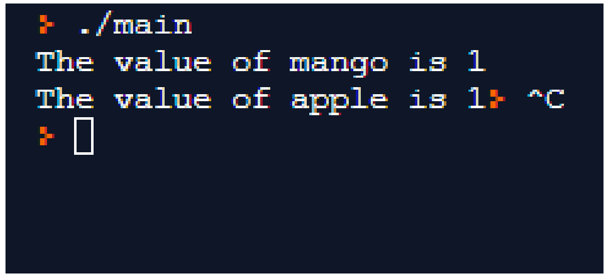
- ➢ The default value of ‘sun’ is 0 and mon is 1.
- ➢ The value of tue is given as 5.
- ➢ So the values of ‘wed’, ‘thurs’, ‘fri’ and ‘sat’ will become 6,7,8 and 9.
- ➢ Thus the value of ‘thurs’ will be printed as 7.

Some important points related to enum

- The enum names available in an enum type can have the same value. Let's look at the example.

```c
#include <stdio.h>


int main(void) {
  enum fruits{mango = 1, strawberry=0, apple=1};
    printf("The value of mango is %d", mango);
    printf("\nThe value of apple is %d", apple);
  return 0;
}
```

Output



```
› ./main
The value of mango is 1
The value of apple is 1› ^C
›
```

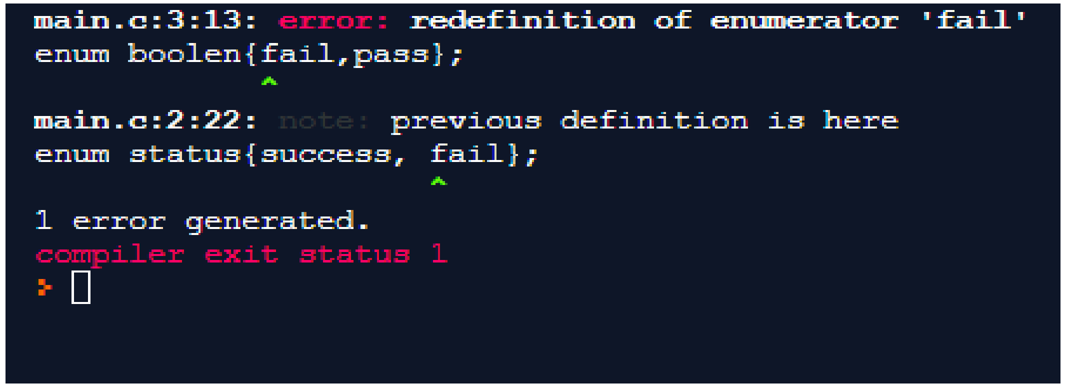
- If we do not provide any value to the enum names, then the compiler will automatically assign the default values to the enum names starting from 0.
- We can also provide the values to the enum name in any order, and the unassigned names will get the default value as the previous one plus one.
- The values assigned to the enum names must be integral constant, i.e., it should not be of other types such string, float, etc.

- All the enum names must be unique in their scope, i.e., if we define two enum having same scope, then these two enums should have different enum names otherwise compiler will throw an error.

Let's understand this scenario through an example.

```c
#include <stdio.h>
enum status{success, fail};
enum boolen{fail,pass};
int main(void) {


    printf("The value of success is %d", success);
  return 0;
}
```

Output



```
main.c:3:13: error: redefinition of enumerator 'fail'
enum boolen{fail,pass};
            ^
main.c:2:22: note: previous definition is here
enum status{success, fail};
                     ^
1 error generated.
compiler exit status 1
▸ ▯
```

Enum vs Macro

We can also use macros to define names constants. For example we can define 'Working' and 'Failed' using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

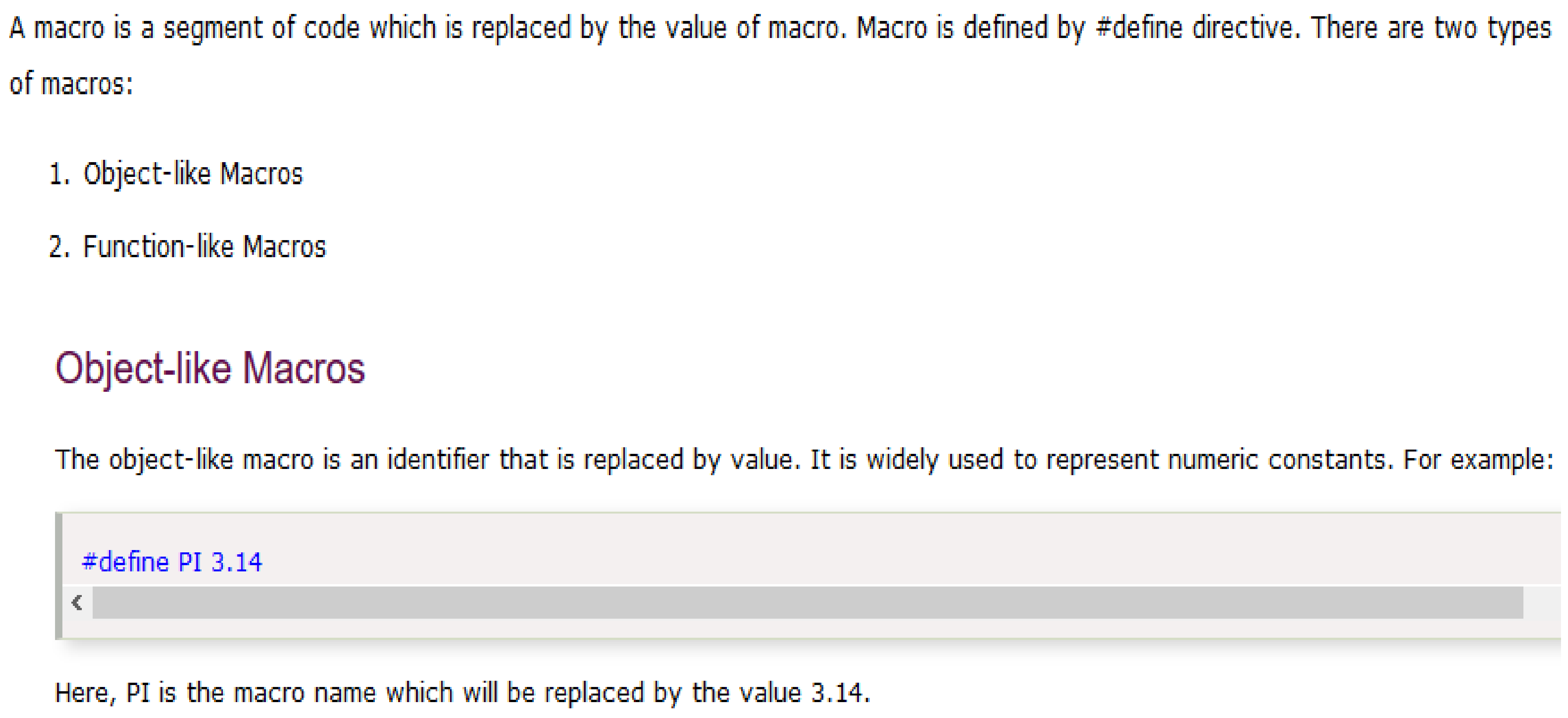
a) Enums follow scope rules.

b) Enum variables are automatically assigned values. Following is simpler

```
enum state  {Working, Failed, Freezed};
```

Macros and Preprocessors

Macros



A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

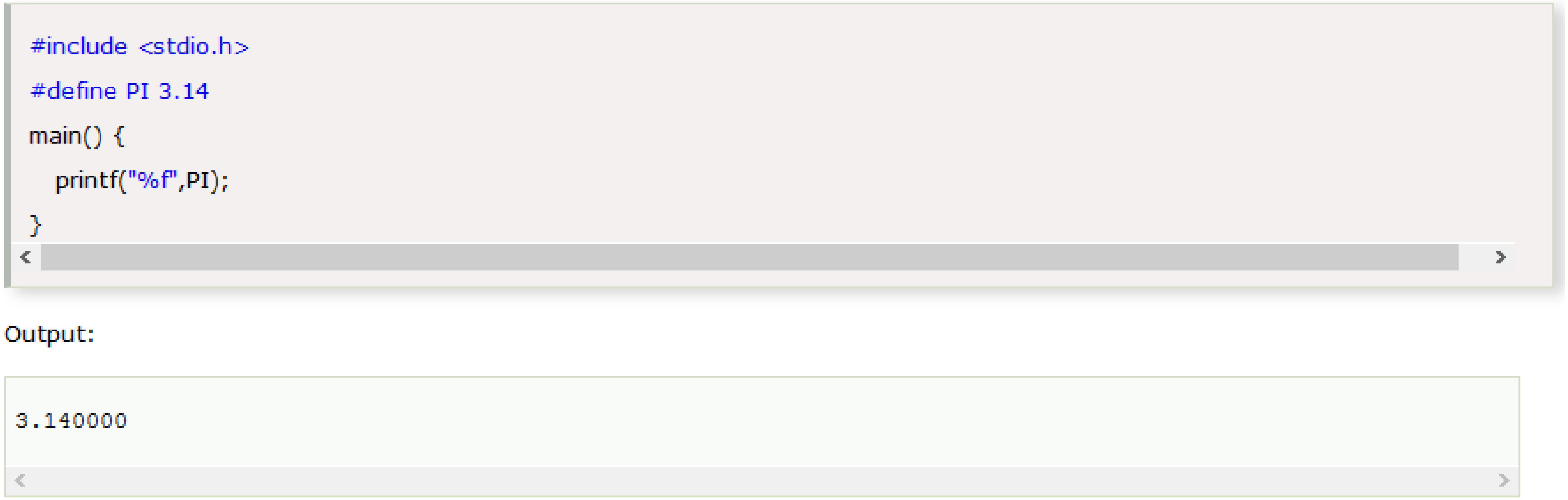
1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

```
#define PI 3.14
```

Here, PI is the macro name which will be replaced by the value 3.14.



```c
#include <stdio.h>
#define PI 3.14
main() {
  printf("%f",PI);
}
```

Output:

```
3.140000
```

Function-like Macros

The function-like macro looks like function call. For example:

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name.

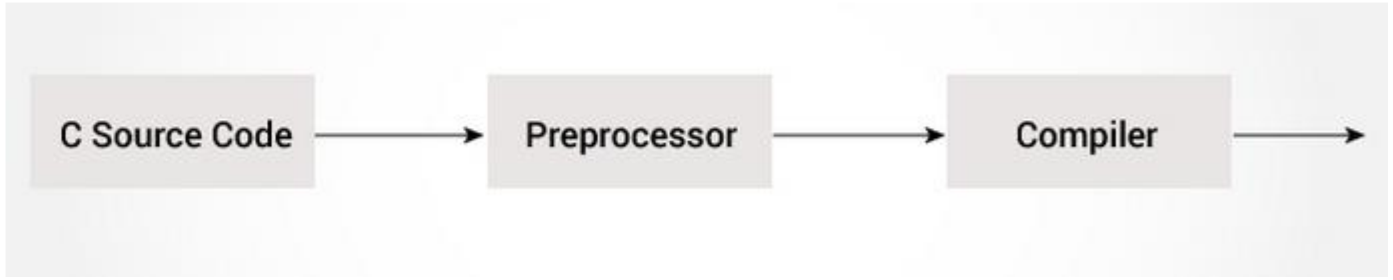
```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main() {
   printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output:

```
Minimum between 10 and 20 is: 10
```

C Preprocessor

- The C Preprocessor transforms your program before it is compiled.
- These transformations can be the inclusion of header file, macro expansions etc.
- C preprocessor is a text substitution tool, and it instructs the compiler to do the required preprocessing, before actual compilation.
- All preprocessor commands start with a # symbol.



Some of the common uses of preprocessor directives are:

1. Including Header Files: # include.

   - eg. # include <stdio.h>
   - Here stdio.h is a header file.
   - The #include preprocessor replaces the above line with the contents of stdio.h header file.
   - You can also create your own header file and include it in your program using this preprocessor directive.

     eg. #include “my_header.h”

2. Macros using # define pre-processor directive.

- ➢ We can define a macro in C using the # define preprocessor directive.
- ➢ eg. *#define PI 3.14159*,
- ➢ When we use *PI* in the program, it is replaced with 3.14159.

Thank You