

Fundamentals of Programming

CCS1063/CSE1062

Lecture 7 –Macro and Functions

Professor Noel Fernando



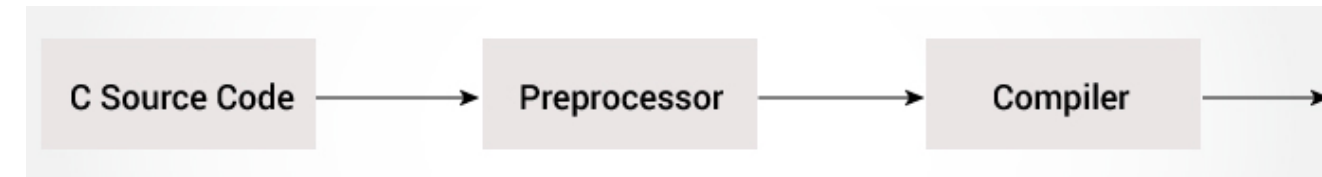
What is macro & Why macro is useful?

- Macro in C programming is known as the piece of code defined with the help of the `#define` directive.
- Macros in C are very useful at multiple places to replace the piece of code with a single value of the macro.
- Macros have multiple types and there are some predefined macros as well
- Macros are one of the convenient ways to write robust and scalable code.

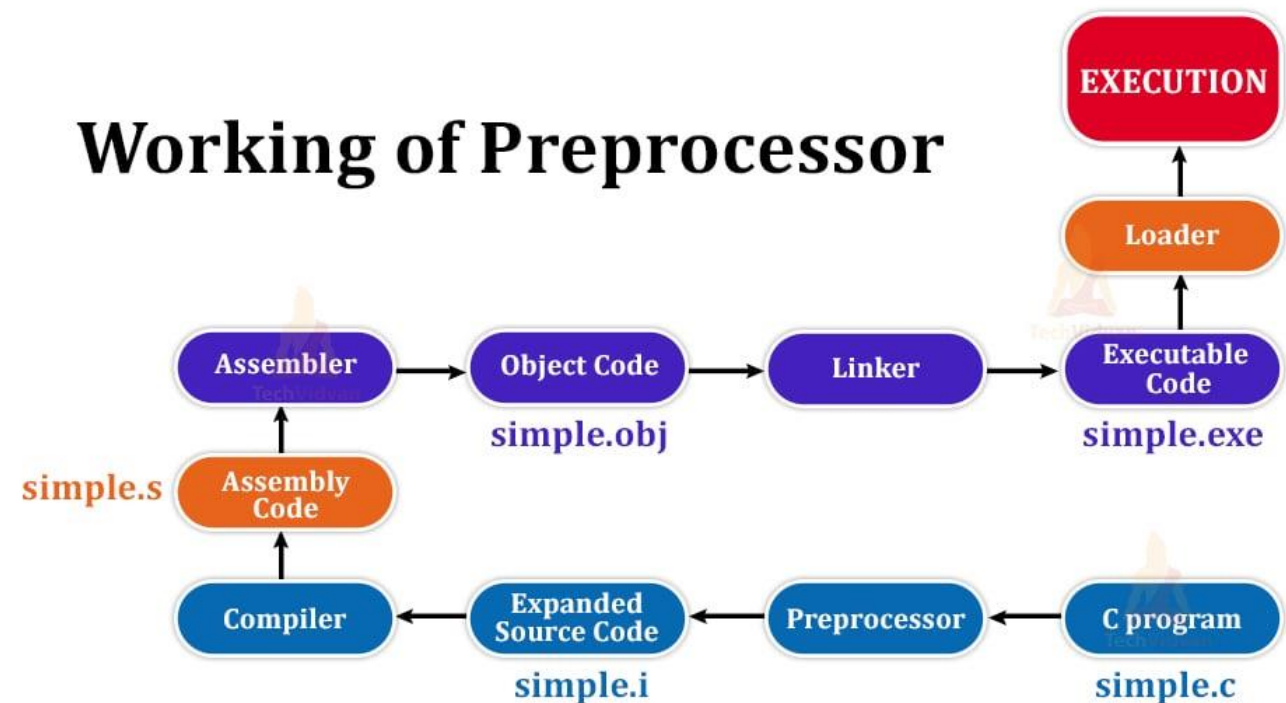
C Preprocessor

- The C Preprocessor is not part of the compiler, but is a separate step in the compilation process.
- The preprocessor performs various tasks, including text manipulation and conditional compilation, to prepare the code for the compilation phase.
- C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before actual compilation.
- All preprocessor commands begin with **#** symbol.

```
# define PI 3.14
```



Working of Preprocessor



C Preprocessor

- The syntax of the macro is as shown in the following figure.
- Here, we will have the three components:
 1. #define - Preprocessor Directive
 2. PI - Macro Name
 3. 3.14 - Macro Value



List of Important Preprocessors

Directive	Description
# define	Substitutes a preprocessor macro
# include	Inserts a particular header from another file
# undef	Undefines a preprocessor macro
# ifdef	Returns true if this macro is defined
# ifndef	Returns true if this macro is not defined
# if	Tests if a compile time condition is true
# else	The alternative for #if
# elif	#else and #if in one statement
# endif	Ends preprocessor conditional
# error	Prints error message on stderr
# pragma	Issues special commands to the compiler, using standardized method

C - Preprocessors

```
#include <stdio.h>
#define PI 3.14159
int main() {
    printf("The value of PI is: %f\n", PI);
#ifdef DEBUG
    printf("Debug mode is
enabled.\n");
#endif
    return 0;
}
```

- In this example,
- The **#include** directive is used to include the standard input/output header file.
- The **#define** directive defines the macro **PI** with the value **3.14159**
- **The #ifdef** directive checks if the **DEBUG** macro is defined; if it is, the debug message will be printed.


Debugging Using Preprocessor Directive in C

- The **C-compiler** can detect only syntax errors whereas it cannot detect any run-time errors.
- So, this may produce an erroneous output if the run-time error persists.
- To detect the run-time errors, the programmer needs to debug the program before execution.
- Using preprocessor directives, the debugging statements can be enabled or disabled as per our needs.


Example

```
int main()
{
    int a = 5, b = 10, sum = 0;
    sum = a + b;
#ifdef DEBUG
    printf("At this point,\nsum of a and
b = %d \n", sum);
#endif
    a++;
    b--;
#ifdef DEBUG
    printf("\nAt this point,\nvalue of a =
%d\n", a);
    printf("value of b = %d", b);
#endif
}
```

Output



At this point,
sum of a and b = 15



At this point,
value of a = 6
value of b = 9

Example

- After the completion of the debugging process, we can remove the macro **DEBUG** by simply replacing the **#define** with **#undef** directive.
- The use of **#ifdef** and **#endif** for each debug statement seems to be lengthier.
- Hence to make the process more concise, we can use of another macro **SHOW**.
- The macro **SHOW** is defined in such a way that when macro **DEBUG** is defined then this **SHOW** will be replaced by a **printf()** statement and if **DEBUG** is not defined then **SHOW** is not replaced by anything.

Example

```
// C program to demonstrating the debugging  
process using
```

```
// preprocessor directive '#define' and macro  
'SHOW'.
```

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

```
#define DEBUG
```

```
#ifdef DEBUG
```

```
#define SHOW printf
```

```
#else
```

```
#define SHOW // macros
```

```
#endif
```

```
int main()
```

```
{
```

```
    int a = 5, b = 10, sum = 0;
```

```
    sum = a + b;
```

```
    SHOW("At this point,\nsum of a and b = %d \n",  
sum);
```

```
        a++;
```

```
        b--;
```

```
    SHOW("\nAt this point,\nvalue of a = %d\n", a);  
        ("value of b = %d", b);
```

```
}
```

Output

```
At this point,  
sum of a and b = 15
```

```
At this point,  
value of a = 6
```

Header Files

- The `#include` preprocessor is used to include header files to a C program.

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

- "`stdio.h`" is a header file. The `#include` preprocessor directive replaces the above line with the contents of `stdio.h` header file which contains function and macro definitions.
- That's the reason why you need to use `#include <stdio.h>` before you can use functions like `scanf()` and `printf()`.
- You can also create your own header file containing function declaration and include it in your program.
- Header files can be included in two ways;
 1. `<stdio.h>` - your header file must be in same location which has been defined by your environment.
 2. "`myheader.h`" – you can place it anywhere and give absolute or relative path.

Preprocessor Operators

- The C preprocessor offers the following operators to help in creating macros:
 - **Macro Continuation (\)**
 - A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line.
 - **Stringize (#)**
 - The Stringize or number-sign operator ('#'), when used within a macro definition, **converts a macro parameter into a string constant**.
 - This operator may be used only in a macro that has a specified argument or parameter list.
 - **Token Passing (##)**
 - The token-pasting operator (##) within a macro definition combines two arguments.
 - It permits two separate tokens in the macro definition to be joined into a single token

Macros using #define

- A macro is a fragment of code that is given a name.
- You can use that fragment of code in your program by using the name.

```
# define PI 3.14
```

- You can also define macros that works like a function call, known as function-like macros.

```
# define PI 3.14  
# define curcleArea(r) (PI*r*r)
```

Macros using #define

```
# define PI 3.14  
# define curcleArea(r) (PI*r*r)
```

- Every time the program encounters `circleArea(argument)`, it is replaced by `(3.14*(argument)*(argument))`.
- Suppose, we passed 5 as an argument during the program, then,
 - `CurcleArea(5)` expands to `(3.14 * 5 * 5)`

Example: #define

- A typical exercise in an algebra book asks you to evaluate an expression like
 $n/3+2$,
 for $n=2$, $n=5$ and $n=9$
- We can formulate such an expression as a program and use the program as many times as necessary.

Example: #define

- A typical exercise in an algebra book asks you to evaluate an expression like
 $n/3+2$,
 for $n=2$, $n=5$ and $n=9$
- We can formulate such an expression as a program and use the program as many times as necessary.

```
#define f(n) (n/3.0 + 2)
```


Example: #define

- A typical exercise in an algebra book asks you to evaluate an expression like
 $n/3+2$, for $n=2$, $n=5$ and $n=9$
- We can formulate such an expression as a program and use the program as many times as necessary.

```
#define f(n) (n/3.0 + 2)
```

```
int main() {  
    for (int i=1; i<=10; i++) {  
        printf("when n = %d, %.2f\n", i, f(i));  
    }  
}
```

Exercise :

- formulate the following six expressions as programs and determine their results for $n=2$, $n=5$ and $n=9$.

1. $n^2 + 10$

2. $(n^2 + 40)/2$

3. $2 - (1/n)$

4. $n^2 + (1/2) * n^2 + 30$

5. $(2 - (1/n)) * (n^2 + 10)$

6. $(1/2) * n^2 + 20 + (2 - (1/n)) * (n^2 + 10)$

Function Like Macros in C

- In the function like macros are very similar to the actual function in C programming.
- We can pass the arguments with the macro name and perform the actions in the code segment.
- In macros, there is no type checking of arguments so we can use it as an advantage to pass different datatypes in same macros in C language.

```
#include <stdio.h>
//object like macro
#define PI 3.14
// function like macro
#define Area(r) (PI*(r*r))
void main()
{
    // declaration and initialization of radius
    float radius = 2.5;
    // declaring and assigning the value to area
    float area = Area(radius);
    // Printing the area of circle
    printf("Area of circle is %f\n", area);
    // Using radius as int data type
    int radiusInt = 5;
    printf("Area of circle is %f", Area(radiusInt));
}
```

Output:

Area of circle is 19.625000

Area of circle is 78.500000

Conditional Compilation

- In C programming, you can instruct preprocessor whether to include certain block of code or not. To do so, conditional directives can be used.
- Uses of Conditional compilation can be;
 1. use different code depending on the machine, operating system
 2. compile same source file in two different programs
 3. to exclude certain code from the program but to keep it as reference for future purpose
- To use conditional compilations, `#define`, `#defined`, `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else` and `#endif` directives are used.
- The special operator `#defined` is used to test whether certain macro is defined or not.
 - It's often used with `#if` directive.

Conditional Compilation

```
#ifdef MACRO
    conditional codes
#endif
```

```
#if expression
    conditional codes if expression is non-zero
#elif expression1
    conditional codes if expression is non-zero
#elif expression2
    conditional codes if expression is non-zero
... ..
else
    conditional if all expressions are 0
#endif
```

Example 1: Conditional Compilation

```
# include <stdio.h>

# define iOS

int main() {

    # ifdef iOS
    /* codes for iOS */
    printf("Your device is operating with iOS...\n");

    # else
    /* codes for Android */
    printf("Your device is operating with Android...\n");

    #endif

    printf("your OS was detected");
}
```

Example 2: Conditional Compilation

- Conditional compilation in C programming language:
- Conditional compilation as the name implies that the code is compiled if certain condition(s) hold true.
- Normally we use if keyword for checking some condition so we have to use something different so that compiler can determine whether to compile the code or not

Example :

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

```
#define x 10
```

```
int main()  
{
```

```
    #ifdef x
```

```
        printf("hello\n"); // this is compiled as x is defined
```

```
    #else
```

```
        printf("bye\n"); // this isn't compiled
```

```
    #endif
```

```
    return 0;
```

```
}
```

Example 3: Conditional Compilation

```
# include <stdio.h>

# define MARKS 10

int main() {

    #if MARKS
        printf("You have faced to the exam ");

        #if MARKS > 50
            printf("and you passed the exam with a grade.\n");

        #elif MARKS > 30
            printf("and you just passed the exam.\n");

        #else
            printf("but you failed the exam.\n");
        #endif

    #else
        printf("You need to sit for the exam.\n");

    #endif

    #if defined GRADE && defined MARKS
        printf("You can get your grades.\n");
    #endif

    printf("Done!");
}
```


Predefined Macros

Predefined Macro	Value
__DATE__	String containing the current date
__TIME__	String containing the current time.
__LINE__	Integer representing the current line number
__FILE__	String containing the file name.
__STDC__	If follows ANSI standard C, then value is a nonzero integer

```
int main() {  
    printf("date: %s\n", __DATE__);  
    printf("time: %s\n", __TIME__);  
    printf("line: %d\n", __LINE__);  
    printf("file: %s\n", __FILE__);  
    printf("stdc: %d\n", __STDC__);  
}
```

ANSI (American National Standards Institute)

Functions

- A **function** is a block of code that performs a specific task.
- Dividing complex problem into small components makes program easy to understand and use.
- There are two types of functions in C programming:
 1. Standard library functions
 2. User defined functions

Standard Library Functions

- The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.
- These functions are defined in the header file. When you include the header file, these functions are available for use.
- For example:
 - The `printf()` is a standard library function to send formatted output to the screen (display output on the screen).
 - This function is defined in "`stdio.h`" header file.
 - There are other numerous library functions defined under "`stdio.h`", such as `scanf()`, `fprintf()`, `getchar()` etc.
 - Once you include "`stdio.h`" in your program, all these functions are available for use.

User-defined Functions

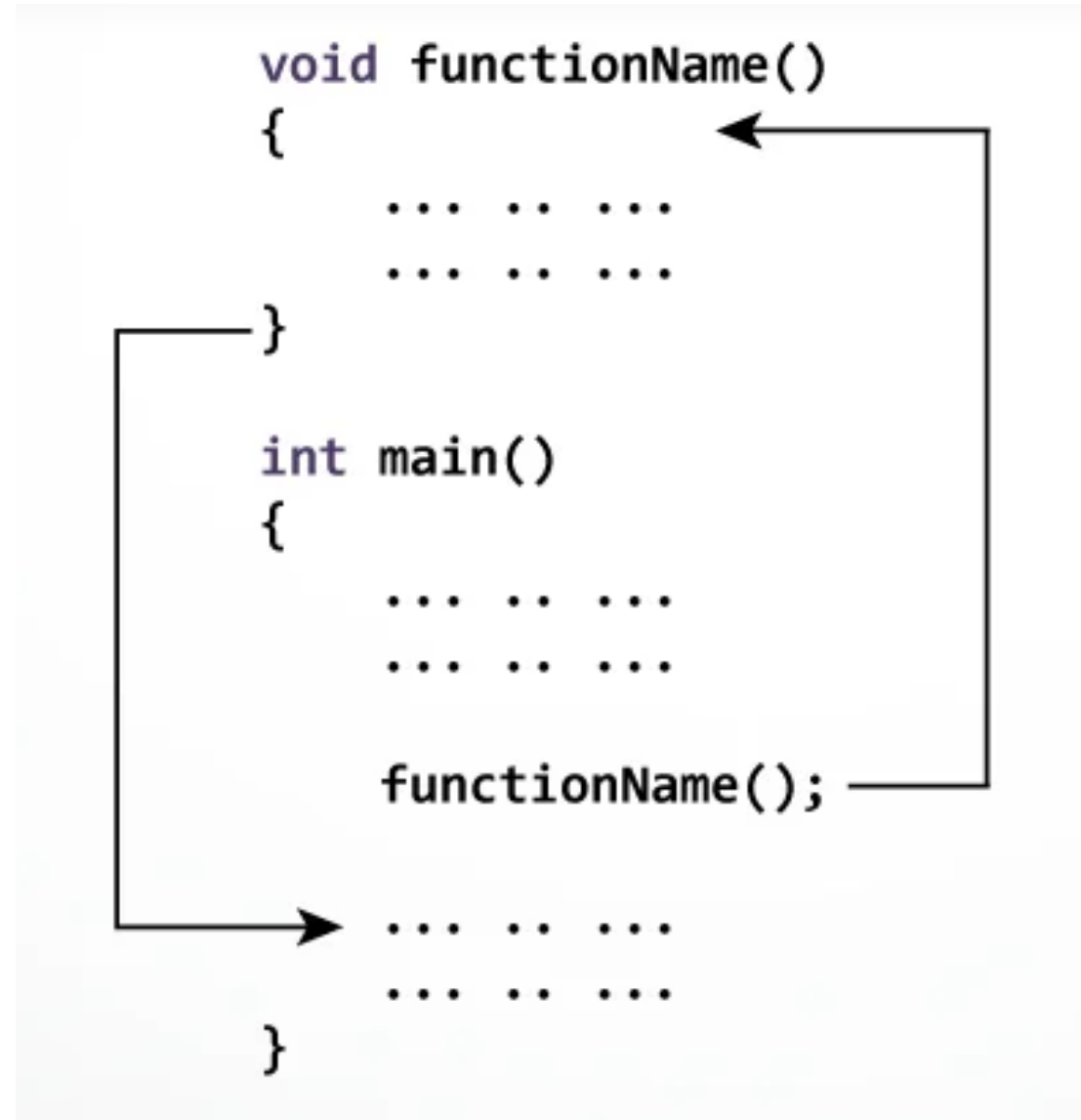
- C allows you to define functions according to your need. These functions are known as user-defined functions.
- For example:
 - Suppose, you need to define a function to find the maximum number between 2 numbers or draw a circle when the radius is given.
 - Then you can create the following functions on your own.
 1. `findMax(int x, int y)`
 2. `drawCircle(int r)`

Defining a Function

```
return_type function_name( parameter list ) {  
    body of the function  
}
```

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the **function signature**.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument.
- **Function Body** – The function body contains a collection of statements that define what the function does.

How function works in C programming?



Write a C program with function(s) to find the maximum from two numbers

Write a C program with function(s) to find the maximum from two numbers

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```


Write a C program with function(s) to find the maximum from two numbers

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
    ret = max(a, b);

    printf( "Max value is : %d\n", ret );

    return 0;
}
```

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Max value is : 200

User-defined Functions- Example 1

```
#include <stdio.h>
// Create a function
void myFunction() {
    printf("I just got executed!");
}
int main() {
    myFunction(); // call the function
    return 0;
}
```

- Output:
I just got executed!

User-defined Functions- Example 2

- `#include <stdio.h>`
- `// Create a function`
- `void myFunction() {`
- `printf("I just got executed!\n");`
- `}`
- `int main() {`
- `myFunction(); // call the function`
- `myFunction(); // call the function`
- `myFunction(); // call the function`
- `return 0;`
- `}`

- Output:

I just got executed!
I just got executed!
I just got executed!

C - Scope Rules

- There are three places where variables can be declared in C programming language –
- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Local Variables

- Variables that are declared inside a function or block are called local variables.
- Local variables are not known to functions outside their own.

```
#include <stdio.h>

int main () {

    /* local variable declaration */
    int a, b;
    int c;

    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;

    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

    return 0;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

    /* local variable declaration */
    int a, b;

    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;

    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

    return 0;
}
```

What is the output of the program

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

    /* local variable declaration */
    int g = 10;

    printf ("value of g = %d\n", g);

    return 0;
}
```

- Output

What is the output of the program

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

    /* local variable declaration */
    int g = 10;

    printf ("value of g = %d\n", g);

    return 0;
}
```

- Output
- value of g = 10

Formal Parameters


- Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.
- Write the output of the program?

```
1  #include <stdio.h>
2
3  /* global variable declaration */
4  int a = 20;
5
6  int main () {
7
8      /* local variable declaration in main function */
9      int a = 10;
10     int b = 20;
11     int c = 0;
12
13     printf ("value of a in main() = %d\n", a);
14     c = sum( a, b);
15     printf ("value of c in main() = %d\n", c);
16
17     return 0;
18 }
19
20 /* function to add two integers */
21 int sum(int a, int b) {
22
23     printf ("value of a in sum() = %d\n", a);
24     printf ("value of b in sum() = %d\n", b);
25
26     return a + b;
27 }
```

Formal Parameters

- Write the output of the program?

value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30



```
1  #include <stdio.h>
2
3  /* global variable declaration */
4  int a = 20;
5
6  int main () {
7
8      /* local variable declaration in main function */
9      int a = 10;
10     int b = 20;
11     int c = 0;
12
13     printf ("value of a in main() = %d\n", a);
14     c = sum( a, b);
15     printf ("value of c in main() = %d\n", c);
16
17     return 0;
18 }
19
20 /* function to add two integers */
21 int sum(int a, int b) {
22
23     printf ("value of a in sum() = %d\n", a);
24     printf ("value of b in sum() = %d\n", b);
25
26     return a + b;
27 }
```

Signature of a Function

The function_name and parameters list are together known as the signature of a function in C programming

```
(return type) functionName (<arg1Type> <arg1Name>,  
<arg2Type> <arg2Name>, ...)
```



Types of User-defined Functions

1. Functions with **no arguments** and **no return value**
2. Functions with **no arguments** and a **return value**
3. Functions with **arguments** and **no return value**
4. Functions with **arguments** and a **return value**

Types of User-defined Functions

- **Example 1: No Argument Passed and No Return Value**
- The checkPrimeNumber() function takes input from the user,
- Checks whether it is a prime number or not, and displays it on the screen.

```
#include <stdio.h>

void checkPrimeNumber();

int main() {
    checkPrimeNumber();    // argument is not passed
    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeNumber() {
    int n, i, flag = 0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0) {
            flag = 1;
            break;
        }
    }

    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

Types of User-defined Functions

- **Example 2: No Arguments Passed But Returns a Value**
- The empty parentheses in the
- `n = getInteger();`
- statement indicates that no argument is passed to the function.
- And, the value returned from the function is assigned to `n`

```
#include <stdio.h>
int getInteger();

int main() {

    int n, i, flag = 0;

    // no argument is passed
    n = getInteger();

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }
}
```

Types of User-defined Functions

- **Example 2: No Arguments Passed But Returns a Value**

- The empty parentheses in the
- `n = getInteger();`
- statement indicates that no argument is passed to the function.
- And, the value returned from the function is assigned to `n`

Output:

Enter a positive integer: 45
45 is not a prime number.

```
if (flag == 1)
    printf("%d is not a prime number.", n);
else
    printf("%d is a prime number.", n);

return 0;
}

// returns integer entered by the user
int getInteger() {
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    return n;
}
```

Types of User-defined Functions

- **Example 3: Argument Passed But No Return Value**
- The integer value entered by the user is passed to the `checkPrimeAndDisplay()` function.
- Here, The `checkPrimeAndDisplay()` function checks whether the argument passed is a prime number or not and displays the appropriate message.

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main() {

    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
    int i, flag = 0;

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        flag = 1;

    for(i = 2; i <= n/2; ++i) {
        if(n%i == 0){
            flag = 1;
            break;
        }
    }

    if(flag == 1)
        printf("%d is not a prime number.",n);
    else
        printf("%d is a prime number.", n);
}
```


Types of User-defined Functions

- **Example 4: Argument Passed and Returns a Value**
- The input from the user is passed to the checkPrimeNumber() function
- Depending on whether flag is **0** or **1**, an appropriate message is printed from the main() function.

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main() {

    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the returned value is assigned to the flag variable
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// int is returned from the function
int checkPrimeNumber(int n) {

    // 0 and 1 are not prime numbers
    if (n == 0 || n == 1)
        return 1;
    int i;

    for(i=2; i <= n/2; ++i) {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

Advantages of Functions

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules.
4. Hence, a large project can be divided among many programmers.

H/W: Problems

1. Write a program to calculate area of a cylinder by using macros.
2. Company XYZ & Co. pays all its employees Rs.150 per normal working hour and Rs. 75 per OT hour. A typical employee works 40 (normal) and 20 (OT) hours per week has to pay 10% tax. Develop a program that determines the take home salary of an employee from the number of working hours and OT hours given.

H/W: Problems

3. Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer the people who can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of Rs 15.00 per ticket, 120 people attend a performance. Decreasing the price by 5 Rupees increases attendance by 20 and increasing the price by 5 Rupees decreases attendance by 20. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner Rs.500. Each attendee costs another 3 Rupees. The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

when we are confronted with such a situation, it is best to tease out the various dependencies one at a time:

Profit is the difference between revenue and costs.

The revenue is exclusively generated by the sale of tickets. It is the product of ticket price and number of attendees.

The costs consist of two parts: a fixed part (Rs.500) and a variable part (Rs. 3 per attendee) that depends on the number of attendees.

Write a program to solve this problem.