

CC-112L

Programming Fundamentals

Laboratory 13

Preprocessor Directives

Version: 1.0.0

Release Date: 21-10-2022

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

- Learning Objectives
- Required Resources
- General Instructions
- Background and Overview
 - Preprocessor Directive
 - #include Preprocessor Directive
 - #define Preprocessor Directive
 - Conditional Compilation
 - #error and #pragma Preprocessor Directive
 - # and ## Operators
 - Assertions
- Activities
 - Pre-Lab Activity
 - #include Preprocessor Directive
 - #define Preprocessor Directive
 - Symbolic constant
 - Replacing Symbolic Constants
 - Common Error with Symbolic Constants
 - #define Preprocessor Directive: Macros
 - Macro with One Argument
 - Example Code 1
 - Importance of Parentheses
 - Example Code 2
 - Macro with Two Arguments
 - Task 01: Preprocessor Directive to Accomplish
 - In-Lab Activity
 - Macro Continuation Character
 - #undef Preprocessor Directive
 - Standard-Library Macros
 - Conditional Compilation
 - #if...#endif Preprocessor Directive
 - Example Code 3
 - Conditionally Compiling Debug Code
 - Example Code 4
 - #error and #pragma Preprocessor Directives
 - # and ## Operators
 - Example Code 5
 - Example Code 6
 - Line Numbers
 - Task 01: Volume of sphere
 - Task 02: Totaling an Array's Contents
 - Task 03: Print an Array
 - Post-Lab Activity
 - Assertions
 - Example Code 7
 - Example Code 8
- Submissions
- Evaluations Metric

- References and Additional Material
- Lab Time and Activity Simulation Log

Learning Objectives:

- Understanding #include Preprocessor Directive
- Learn #define Preprocessor Directive
- Learn Conditional Compilation
- Understanding #error and #pragma Preprocessor Directive
- Understanding # and ## Operators
- Understanding Predefined Symbolic Constants
- Learn Assertions

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Lab Instructor	Madiha Khalid	madiha.khalid@pucit.edu.pk
Teacher Assistants	Usman Ali	bitf19m007@pucit.edu.pk
	Saad Rahman	bsef19m021@pucit.edu.pk

Background and Overview:

Preprocessor Directive:

Preprocessor directives begin with #. Only whitespace characters and comments delimited by /* and */ may appear before a preprocessor directive on a line. The C preprocessor executes before each program compiles. It:

- includes other files into the file being compiled,
- defines symbolic constants and macros,
- conditionally compiles program code and
- conditionally executes preprocessor directives.

#include Preprocessor Directive:

You've used the #include preprocessor directive throughout this course. When the preprocessor encounters a #include, it replaces the directive with a copy of the specified file. The two forms of the #include directive are:

```
#include <filename>
```

```
#include "filename"
```

The difference between these is the location where the preprocessor begins searching for the file. For filenames enclosed in angle brackets (< and >), such as standard library headers, the preprocessor searches in implementation-dependent compiler and system folders. You typically use filenames enclosed in quotes (") to include headers that you define for use with your program.

#define Preprocessor Directive:

The #define directive creates:

- symbolic constants, constants represented as identifiers, and
- macros, operations defined as symbols.

The #define directive's format is

```
#define identifier replacement-text
```

Conditional Compilation:

Conditional compilation enables you to control which preprocessor directives execute and whether parts of your C code compile. Each conditional preprocessor directive evaluates a constant integer expression. Cast expressions, sizeof expressions and enumeration constants cannot be evaluated in preprocessor directive.

#error and #pragma Preprocessor Directive:

The #error directive terminates preprocessing, prevents compilation and prints an implementation-dependent message that includes the tokens specified in the directive.

The #pragma directive causes an implementation-defined action. If the #pragma is not recognized by the implementation, it's ignored.

and ## Operators:

The # operator converts a replacement-text token to a string surrounded by quotes. The # operator must be used in a macro with arguments because #'s operand must be one of the macro's arguments.

The `##` operator concatenates two tokens. The `##` operator must have two operands.

Predefined Symbolic Constants :

Constant `__LINE__` is the line number (an integer) of the current source-code line.

Constant `__FILE__` is the name of the file (a string).

Constant `__DATE__` is the date the source file is compiled (a string).

Constant `__TIME__` is the time the source file is compiled (a string).

Constant `__STDC__` indicates whether the compiler supports Standard C.

Each of the predefined symbolic constants begins and ends with two underscores.

Assertions:

Macro `assert` tests the value of an expression. If the value is 0 (false), `assert` prints an error message and calls the function `abort` (p. 691) to terminate program execution.

Activities:

Pre-Lab Activities:

#include Preprocessor Directive:

You've used the `#include` preprocessor directive throughout this course. When the preprocessor encounters a `#include`, it replaces the directive with a copy of the specified file. The two forms of the `#include` directive are:

- `#include <filename>`
- `#include "filename"`

The difference between these is the location where the preprocessor begins searching for the file. For filenames enclosed in angle brackets (`<` and `>`)—such as standard library headers—the preprocessor searches in implementation-dependent compiler and system folders. You typically use filenames enclosed in quotes (`"`) to include headers that you define for use with your program. In this case, the preprocessor begins searching in the same folder as the file in which the `#include` directive appears.

If the compiler cannot find the specified file in the current folder, it searches the implementation-dependent compiler and system folders. In addition to using `#include` for standard library headers, you'll frequently use it in programs consisting of multiple source files. You'll often create headers for a program's common declarations, then include that file into multiple source files. Examples of such declarations are:

- struct and union declarations,
- typedefs,
- enums, and
- function prototypes.

#define Preprocessor Directive: Symbolic Constants

The `#define` directive creates:

- symbolic constants: constants represented as identifiers, and
- macros: operations defined as symbols.

The `#define` directive's format is:

```
#define identifier replacement-text
```

By convention, a symbolic constant's identifier should contain only uppercase letters and underscores. Using meaningful names for symbolic constants helps make programs self-documenting.

Replacing Symbolic Constants:

When the preprocessor encounters a `#define` directive, it replaces `identifier` with replacement text throughout that source file, ignoring any occurrences of `identifier` in string literals or comments. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant `PI` with `3.14159`. Symbolic constants enable you to create named constants and use their names throughout the program.

Common Error with Symbolic Constants:

Everything to the right of a symbolic constant's name replaces the symbolic constant. For example,

```
#define PI = 3.14159
```

causes the preprocessor to replace every occurrence of PI with "= 3.14159". Incorrect #define directives cause many subtle logic and syntax errors. So, in preference to the preceding #define, you may prefer to use const variables, such as

```
const double PI = 3.14159;
```

These have the additional benefit that they're defined in C, so the compiler can check them for proper syntax and type safety.

Example Code 1:

```
1  #include <stdio.h>
2  #define PI 3.14159
3
4  int main()
5  {
6      float radius, area;
7
8      printf("Enter the radius of a circle\n");
9
10     scanf("%f", &radius);
11
12     area = PI * radius * radius;
13
14     printf("Area of the circle = %.2f\n", area);
15
16     return 0;
17 }
```

Fig. 01 (#define preprocessor directive)

Output:



```
Microsoft Visual Studio Debug Console
Enter the radius of a circle
4
Area of the circle = 50.27
```

Fig. 02 (#define preprocessor directive output)

#define Preprocessor Directive: Macros

Technically, any identifier defined in a #define preprocessor directive is a macro. As with symbolic constants, the macro-identifier is replaced with replacement-text before the program is compiled. Macros may be defined with or without arguments. A macro without arguments is a symbolic constant. When the preprocessor encounters a macro with arguments, it substitutes the arguments in the replacement text, then expands the macro, that is, it replaces the macro with its replacement-text and argument list.

Macro with One Argument:

Consider the following one-argument macro definition that calculates a circle's area:

```
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

Expanding a Macro with an Argument Wherever CIRCLE_AREA(argument) appears in the file, the preprocessor:

- substitutes argument for x in the replacement-text,
- replaces PI with its value 3.14159 (from Section 14.3), and
- expands the macro in the program.

For example, the preprocessor expands

```
double area = CIRCLE_AREA(4);
```

to

```
double area = ((3.14159) * (4) * (4));
```

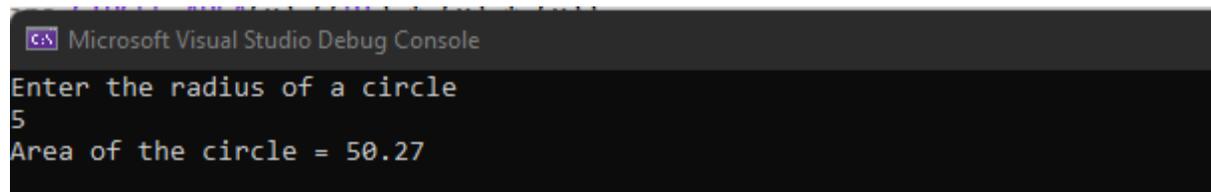
At compile time, the compiler evaluates the preceding expression and assigns the result to the variable area.

Example Code 2:

```
1  #include <stdio.h>
2  #define PI 3.14159
3  #define CIRCLE_AREA(x) ((PI) * (x) * (x))
4
5  int main()
6  {
7      float radius, area;
8
9      printf("Enter the radius of a circle\n");
10
11     scanf("%f", &radius);
12
13     area = CIRCLE_AREA(4);
14
15     printf("Area of the circle = %.2f\n", area);
16
17     return 0;
18 }
19
```

Fig. 03 (Macro with one argument)

Output:



```
Microsoft Visual Studio Debug Console
Enter the radius of a circle
5
Area of the circle = 50.27
```

Fig. 04 (Macro with one argument output)

Importance of Parentheses:

The parentheses around each x in the replacement-text force the proper evaluation order when a macro's argument is an expression. Consider the statement

```
double area = CIRCLE_AREA(c + 2);
```

which expands to

```
double area = ((3.14159) * (c + 2) * (c + 2));
```

This evaluates correctly because the parentheses force the proper evaluation order. If you omit the macro definition's parentheses, the macro expansion is

```
double area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly as

```
double area = (3.14159 * c) + (2 * c) + 2;
```

because of C's operator precedence rules. For this reason, you should always enclose macro arguments in parentheses in the replacement-text to prevent logic errors.

Example Code 3:

```
1  #include <stdio.h>
2  #define PI 3.14159
3  #define CIRCLE_AREA(x) (3.14159 * x + 2 * x + 2)
4
5  int main()
6  {
7      float radius, area;
8
9      printf("Enter the radius of a circle\n");
10
11     scanf("%f", &radius);
12
13     area = CIRCLE_AREA(4);
14
15     printf("Area of the circle = %.2f\n", area);
16
17     return 0;
18 }
19
20
```

Fig. 05 (Importance of parenthesis)

Output:



```
Microsoft Visual Studio Debug Console
Enter the radius of a circle
5
Area of the circle = 22.57
```

Fig. 06 (Importance of parenthesis output)

It's Better to Use a Function

Defining the CIRCLE_AREA macro as a function is safer. The circleArea function

```
double circleArea (double x) {  
    return 3.14159 * x * x;  
}
```

performs the same calculation as macro CIRCLE_AREA, but a function's argument is evaluated only once when the function is called. Also, the compiler performs type checking on functions. The preprocessor does not support type checking.

Macro with Two Arguments:

The following two-argument macro calculates a rectangle's area:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever RECTANGLE_AREA(x, y) appears in the program, the preprocessor substitutes the values of x and y in the macro's replacement-text and expands the macro in the program. For example, the statement:

```
int rectangleArea = RECTANGLE_AREA(a + 4, b + 7);
```

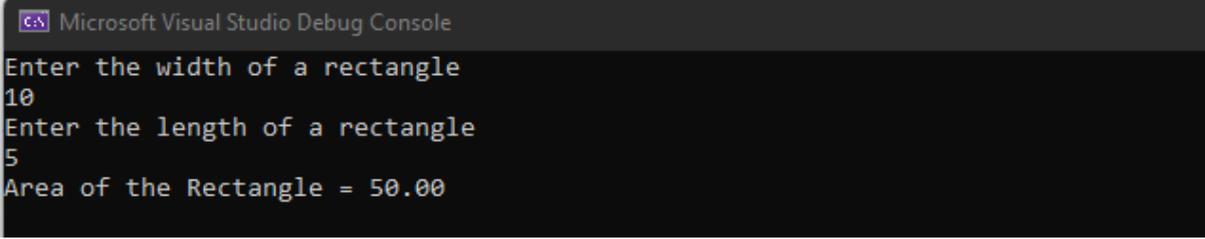
is expanded to

```
int rectangleArea = ((a + 4) * (b + 7));
```

Example Code:

```
1  #include <stdio.h>  
2  #define PI 3.14159  
3  #define RECTANGLE_AREA(x, y) ((x) * (y))  
4  
5  int main()  
6  {  
7      float width, length, area;  
8  
9      printf("Enter the width of a rectangle\n");  
10  
11     scanf("%f", &width);  
12  
13     printf("Enter the length of a rectangle\n");  
14  
15     scanf("%f", &length);  
16  
17     area = RECTANGLE_AREA(length, width);  
18  
19     printf("Area of the Rectangle = %.2f\n", area);  
20  
21     return 0;  
22 }  
23  
24
```

Fig. 07 (Macro with Two Argument)

Output:

```
Microsoft Visual Studio Debug Console
Enter the width of a rectangle
10
Enter the length of a rectangle
5
Area of the Rectangle = 50.00
```

Fig. 08 (Macro with Two Argument output)

Macro Continuation Character:

A macro's or symbolic constant's replacement-text is everything to the identifier's right in the #define directive. If the replacement-text is longer than the remainder of the line, you can place a backslash (\) continuation character at the end of the line to continue the replacement-text on the next line.

#undef Preprocessor Directive

Symbolic constants and macros can be discarded for the remainder of a source file using the #undef preprocessor directive. Directive #undef undefines a symbolic constant or macro name. A macro's or symbolic constant's scope is from its definition until it's undefined with #undef, or until the end of the source file. Once undefined, a macro or symbolic constant can be redefined with #define.

Standard-Library Macros:

Some standard-library functions actually are defined as macros, based on other library functions. A macro commonly defined in the <stdio.h> header is

```
#define getchar() getc(stdin)
```

The macro definition of getchar uses function getc to get one character from the standard input stream. The <stdio.h> header's putchar function and the <ctype.h> header's character-handling functions often are implemented as macros as well.

Task 01: Preprocessor Directive to Accomplish**[20 minutes / 20 marks]**

Write a preprocessor directive to accomplish each of the following:

- Define the symbolic constant YES to have the value 1.
- Define the symbolic constant NO to have the value 0.
- Include the header common.h. The header is found in the same directory as the file being compiled.
- Renumber the remaining lines in the file beginning with line number 3000.
- If the symbolic constant TRUE is defined, undefine it and redefine it as 1. Do not use #ifdef.
- If the symbolic constant TRUE is defined, undefine it and redefine it as 1. Use the #ifdef preprocessor directive.
- If the symbolic constant TRUE is not equal to 0, define symbolic constant FALSE as 0. Otherwise define FALSE as 1.
- Define the macro, CUBE_VOLUME that computes the volume of a cube. The macro takes one argument.

Submit “.c” file named your “**preprocessor.c**” on Google Classroom.

In-Lab Activities:**Conditional Compilation:**

Conditional compilation enables you to control which preprocessor directives execute and whether parts of your C code compile. Each conditional preprocessor directive evaluates a constant integer expression. Cast expressions, sizeof expressions and enumeration constants cannot be evaluated in preprocessor directives.

#if...#endif Preprocessor Directive

The conditional preprocessor construct is much like the if selection statement. Consider the following preprocessor code:

```
#if !defined(MY_CONSTANT)
    #define MY_CONSTANT 0
#endif
```

This determines whether MY_CONSTANT is defined—that is, whether MY_CONSTANT has already appeared in an earlier #define directive within the current source file. The expression defined(MY_CONSTANT) evaluates to 1 (true) if MY_CONSTANT is defined; otherwise, it evaluates to 0 (false). If the result is 0, !defined(MY_CONSTANT) evaluates to 1, indicating that MY_CONSTANT was not defined previously, so the #define directive executes. Otherwise, the preprocessor skips the #define directive.

Every #if construct ends with #endif. The directives #ifdef and #ifndef are shorthand for #if defined(name) and #if !defined(name). You can test a multiple part conditional preprocessor construct by using

- #elif (the equivalent of else if in an if statement) and
- #else (the equivalent of else in an if statement) directives.

Conditional preprocessor directives are frequently used to prevent header files from being included multiple times in the same source file. These directives frequently are used to enable and disable code that makes software compatible with a range of platforms.

Example Code 5:

```
1  #include <stdio.h>
2
3  #define DEBUG 1
4
5  int main() {
6  #ifdef DEBUG
7      printf("Debug is ON\n");
8      printf("Hi Debugger!\n");
9  #else
10     printf("Debug is OFF\n");
11 #endif
12     return 0;
13 }
14
```

Fig. 09 (Conditional Compilation)

Output:

```
Microsoft Visual Studio Debug Console
Debug is ON
Hi Debugger!
```

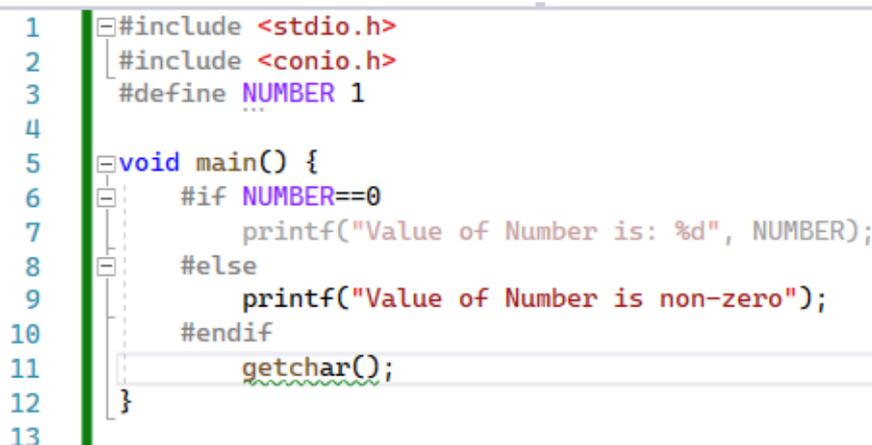
Fig. 10 (Conditional Compilation output)

Conditionally Compiling Debug Code :

Conditional compilation is sometimes used as a debugging aid. For example, some programmers use printf statements to print variable values and to confirm a program's flow of control. You can enclose such printf statements in conditional preprocessor directives so the statements are compiled only while you're still debugging your code. For example,

```
#ifdef DEBUG
    printf("Variable x = %d\n", x);
#endif
```

compiles the printf statement if the symbolic constant DEBUG is defined with #define DEBUG before #ifdef DEBUG. When you complete your debugging phase, you remove or comment out the #define directive in the source file, and the printf statements inserted for debugging purposes are ignored during compilation. In larger programs, you might define several symbolic constants that control the conditional compilation in separate sections of the source file. Many compilers allow you to define and undefine symbolic constants like DEBUG with a compiler flag that you supply each time you compile the code so that you do not need to change the code. When inserting conditionally compiled printf statements in locations where C expects a single statement (e.g., a control statement's body), ensure that the conditionally compiled statements are enclosed in braces ({}).

Example Code 5:

```
1  #include <stdio.h>
2  #include <conio.h>
3  #define NUMBER 1
4
5  void main() {
6      #if NUMBER==0
7          printf("Value of Number is: %d", NUMBER);
8      #else
9          printf("Value of Number is non-zero");
10     #endif
11     getch();
12 }
13
```

Fig. 11 (Conditionally Compiling Debug Code)

Output:

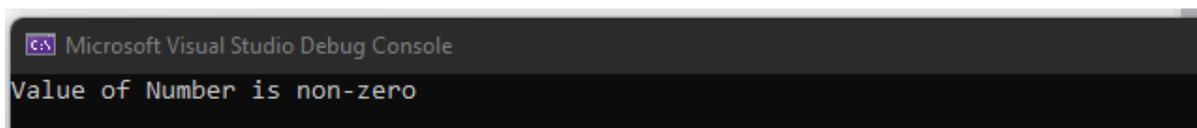


Fig. 12 (Conditionally Compiling Debug Code output)

#error and #pragma Preprocessor Directives :

The #error directive :

#error tokens

prints an implementation-dependent message, including the tokens specified in the directive. The tokens are sequences of characters separated by spaces. For example, #error 1 - Out of range error contains 6 tokens. When the #error directive is processed on some systems, the tokens are displayed as an error message, preprocessing stops, and the program does not compile.

The #pragma directive :

#pragma tokens causes an implementation-defined action. A #pragma not recognized by the implementation is ignored. For more information on #error and #pragma, see the documentation for your C compiler.

and ## Operators

The # operator converts a replacement-text token to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO(x) puts("Hello, " #x);
```

When HELLO(John) appears in a program file, the preprocessor expands it to

```
puts("Hello, " "John");
```

replacing #x with the string "John". Strings separated by whitespace are concatenated during preprocessing, so the preceding statement is equivalent to

```
puts("Hello, John");
```

Example Code 6:

```
1  #include<stdio.h>
2  #define HELLO(x) puts("Hello, " #x);
3
4
5  int main() {
6      HELLO(programmer!);
7      return 0;
8  }
```

Fig. 13 (# and ## operator)

Output:



```
Microsoft Visual Studio Debug Console
Hello, programmer!
```

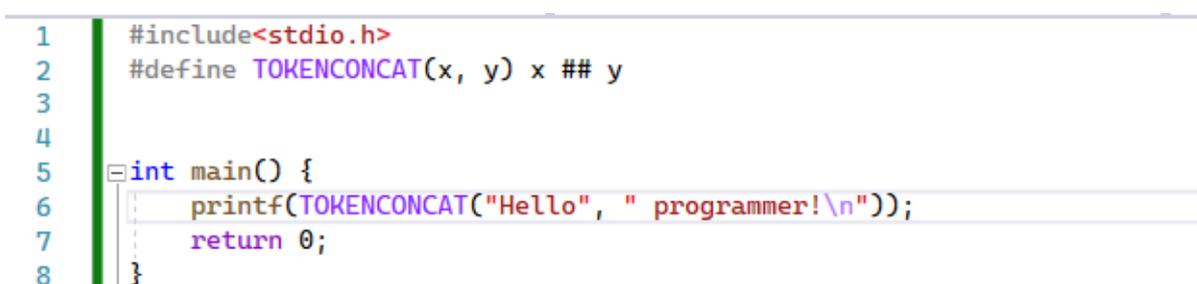
Fig. 14 (# and ## operator output)

The # operator must be used in a macro with arguments because #'s operand refers to one of the macro's arguments. The ## operator concatenates two tokens. Consider the following macro definition:

```
#define TOKENCONCAT(x, y) x ## y
```

When TOKENCONCAT appears in a file, the preprocessor concatenates the arguments and uses the result to replace the macro. For example, TOKENCONCAT(O, K) is replaced by OK in the program. The ## operator must have two operands.

Example Code 7:



```
1 #include<stdio.h>
2 #define TOKENCONCAT(x, y) x ## y
3
4
5 int main() {
6     printf(TOKENCONCAT("Hello", " programmer!\n"));
7     return 0;
8 }
```

Fig. 15 (# and ## operator)

Output:



```
Microsoft Visual Studio Debug Console
Hello programmer!
```

Fig. 16 (# and ## operator output)

Line Numbers:

The #line preprocessor directive causes the subsequent source-code lines to be renumbered, starting with the specified constant integer value. The directive

```
#line 100
```

starts line numbering from 100 beginning with the next source-code line. Including a filename in the #line directive, as in

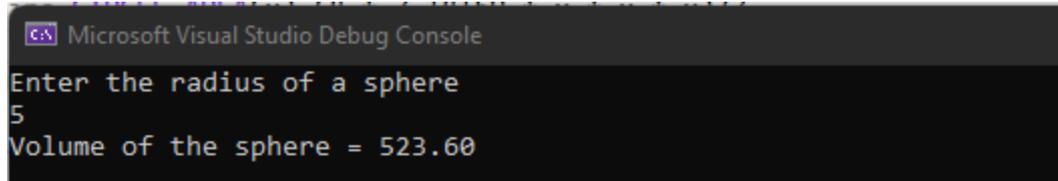
```
#line 100 "file1.c"
```

indicates that lines are numbered from 100 beginning with the next source-code line and that the filename for the purpose of any compiler messages is "file1.c". This version of the #line directive normally helps make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

Task 01: Volume of a Sphere**[10 minutes / 10 marks]**

Write a C program that

- defines a macro with one argument to compute a sphere's volume.
- Use the macro to compute the volumes for spheres of radius 1 to 10 and print the results on the console

Sample Output:

```
Microsoft Visual Studio Debug Console
Enter the radius of a sphere
5
Volume of the sphere = 523.60
```

Fig. 17 (In-Lab Task 01)

Submit “.c” file named your “**volume.c**” on Google Classroom.**Task 02: Totaling an Array's Contents****[20 minutes / 20 marks]**

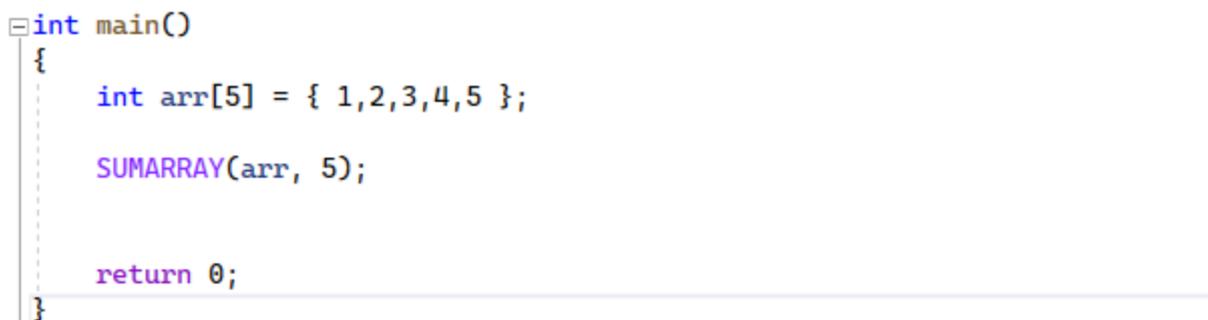
Write a C program that:

- defines and uses macro SUMARRAY to sum the values in a numeric array.
- The macro should receive the array and its number of elements as arguments.
- Declare array of 5 size in main, pass it to the macros to print sum of array.

Sample Output:

```
Microsoft Visual Studio Debug Console
Sum of array is : 15
```

Fig. 18 (In-Lab Task 02)

Main:

```
int main()
{
    int arr[5] = { 1,2,3,4,5 };

    SUMARRAY(arr, 5);

    return 0;
}
```

Fig. 19 (In-Lab Task 02)

Submit “.c” file named your “**SumArray.c**” on Google Classroom.

Task 02: Printing an Array**[20 minutes / 20 marks]**

Write a C program that:

- defines and uses macro PRINTARRAY to print an array of integers.
- The macro should receive the array and its number of elements as arguments.

Submit “.c” file named your “**PrintArray.c**” on Google Classroom.

Post-Lab Activities:

Assertions:

The `assert` macro—defined in `<assert.h>`—tests an expression’s value at execution time. If the value is false (0), `assert` prints an error message and terminates the program by calling function `abort` of the general utilities library (`<stdlib.h>`). The `assert` macro is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable `x` should never be larger than 10 in a program. You can use an assertion to test `x`’s value and print an error message if it’s greater than 10, as in

```
assert(x <= 10);
```

If `x` is greater than 10 when this statement executes, the program displays an error message containing the line number and filename where the `assert` statement appears, then terminates. You’d then focus on this area of the code to find the error. If the symbolic constant `NDEBUG` is defined, subsequent assertions in the source file are ignored. So, when assertions are no longer needed, rather than deleting each assertion manually, you can insert the following line in the source file:

Example Code 8:

```
1  #include<stdio.h>
2  #include <assert.h>
3
4  int main() {
5      int x = 11;
6      assert(x <= 10);
7
8      return 0;
9  }
10
```

Fig. 20 (Assertion)

Output:

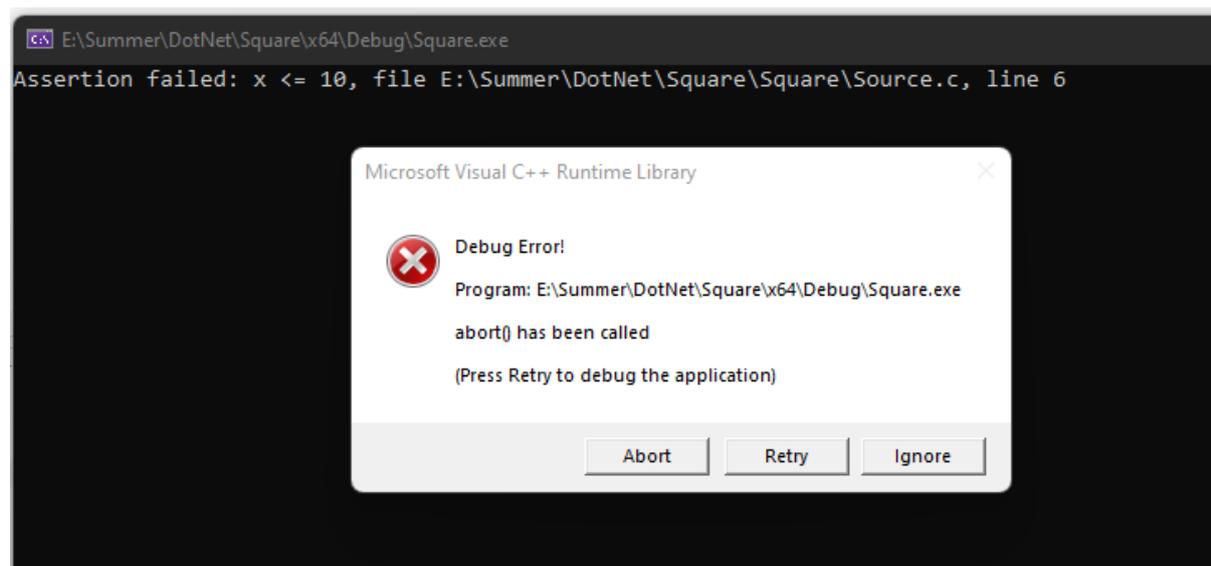


Fig. 21 (Assertion output)

```
#define NDEBUG
```

Many compilers have debug and release modes that automatically define and undefine NDEBUG. Assertions are not meant as a substitute for error handling during normal runtime conditions. You should use them only to find logic errors during program development.

```
1  #include<stdio.h>
2  #define NDEBUG
3  #include <assert.h>
4
5  int main() {
6      int x = 11;
7      assert(x <= 10);
8
9      return 0;
10 }
11
```

Fig. 22 (Assertion - II)

Submissions:

- For In-Lab Activity:
 - Save the files on your PC.
 - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
 - Submit the .c file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** [20 marks]
 - Task 01: Preprocessor directive to accomplish [20 marks]
- **Division of In-Lab marks:** [50 marks]
 - Task 01: Volume of sphere [10 marks]
 - Task 02: Totaling an Array's Contents [20 marks]
 - Task 03: Print an Array [20 marks]

References and Additional Material:

- #include Preprocessor Directive
<https://www.javatpoint.com/c-preprocessor-include>
- #define Preprocessor Directive:
<https://www.javatpoint.com/c-preprocessor-define>
- Conditional Compilation
<https://www.tutorialspoint.com/what-is-conditional-compilation-in-c-language>
- Assertion
<https://www.javatpoint.com/assert-in-c-programming>

Lab Time Activity Simulation Log:

- Slot – 01 – 00:00 – 00:15: Class Settlement
- Slot – 02 – 00:15 – 00:30: Execute C on Visual Studio
- Slot – 03 – 00:30 – 00:45: Execute C on Visual Studio
- Slot – 04 – 00:45 – 01:00: In-Lab Task
- Slot – 05 – 01:00 – 01:15: In-Lab Task
- Slot – 06 – 01:15 – 01:30: In-Lab Task
- Slot – 07 – 01:30 – 01:45: In-Lab Task
- Slot – 08 – 01:45 – 02:00: In-Lab Task
- Slot – 09 – 02:00 – 02:15: In-Lab Task
- Slot – 10 – 02:15 – 02:30: In-Lab Task
- Slot – 11 – 02:30 – 02:45: Evaluation of Lab Tasks
- Slot – 12 – 02:45 – 03:00: Discussion on Post-Lab Task