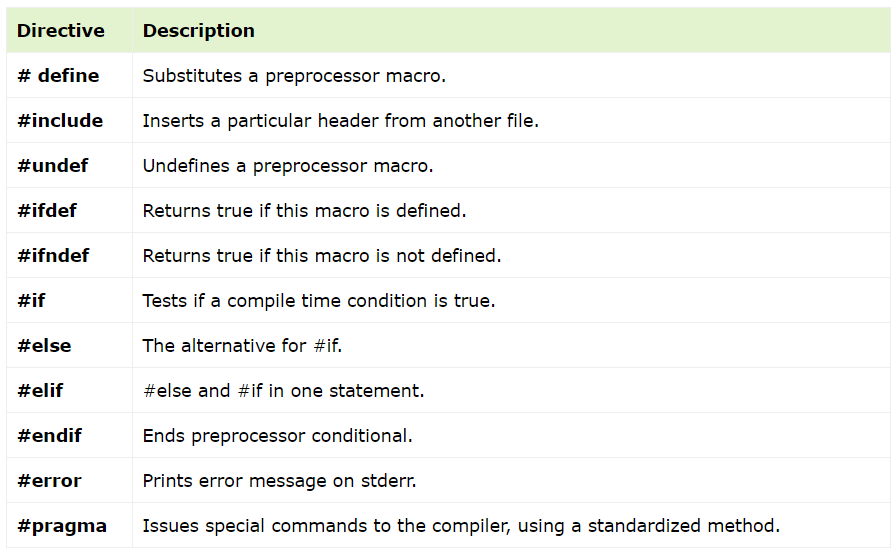
**Preprocessors in C**

By Abdallah Ghazy

* The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process.
* In simple terms, a C Preprocessor is just a **text replacement** tool and it instructs the compiler to do the required pre-processing before the actual compilation.
* preprocessing is the first step in the compilation of a C code.
* It occurs before the tokenization step.
* One of the important functions of a preprocessor is to include the header files that contain the library functions used in the program.
* The preprocessor in C also defines the constants and expands the macros.
* The preprocessor statements in C are called directives.
* A preprocessor section of the program always appears at the top of the C code.
* Each preprocessor statement starts with the hash **(#)** symbol.

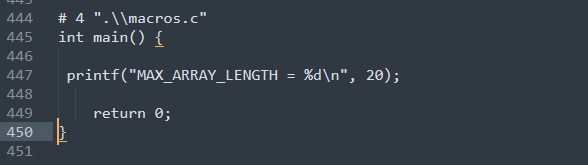
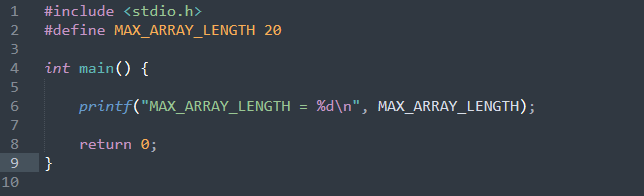
## Preprocessor Directives in C

****

## Preprocessors Examples

* *Analyze the following examples to understand various directives.*
* This **#define** directive tells the compiler to replace the instances of MAX\_ARRAY\_LENGTH with 20. Use **#define** for constants to increase readability.

**After preprocessing**

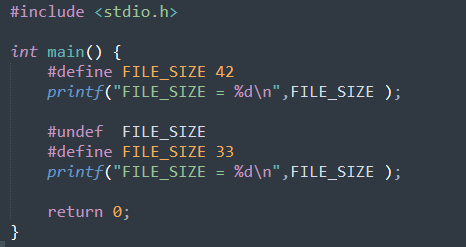
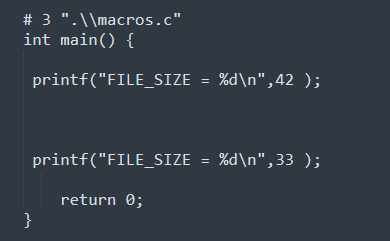
****

.i

.C

* Now, take a look at the following #define and **#undef** directives. They tell the compiler to undefine existing FILE\_SIZE and define it as 42.

**After preprocessing**

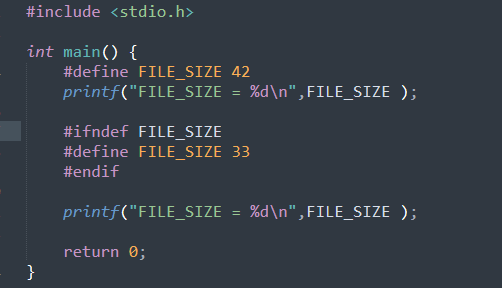
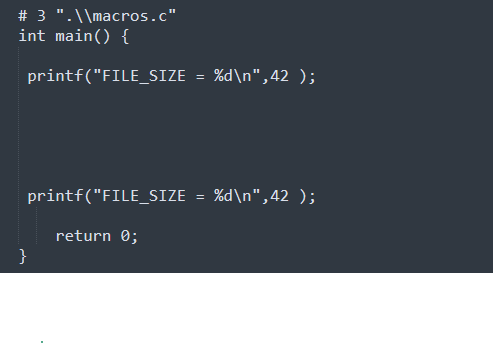


.C

.i

* The following directive tells the compiler to define FILE\_SIZE only if FILE\_SIZE isn't already defined.

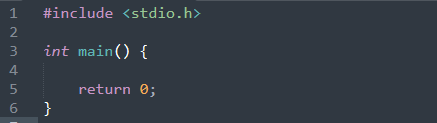
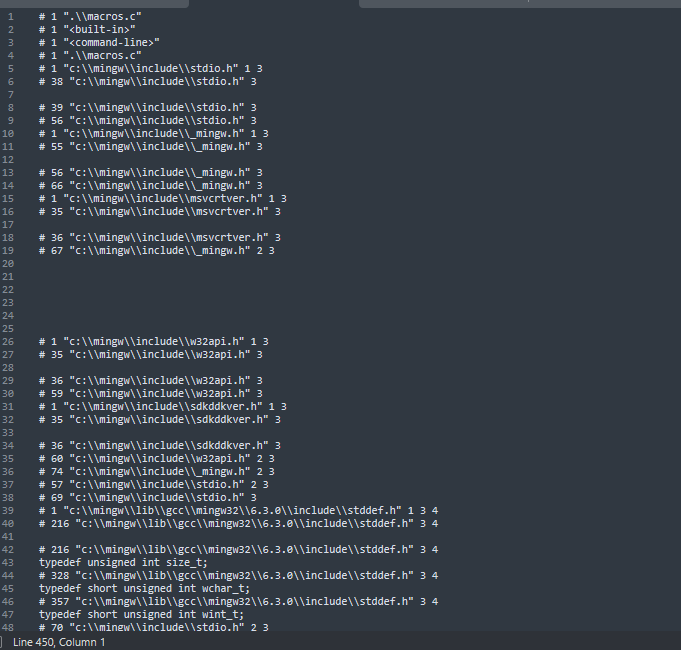
**After preprocessing**



.i

.C

* The following directives tell the compiler to get "stdio.h" from the System Libraries and add the text to the current source file.



.i

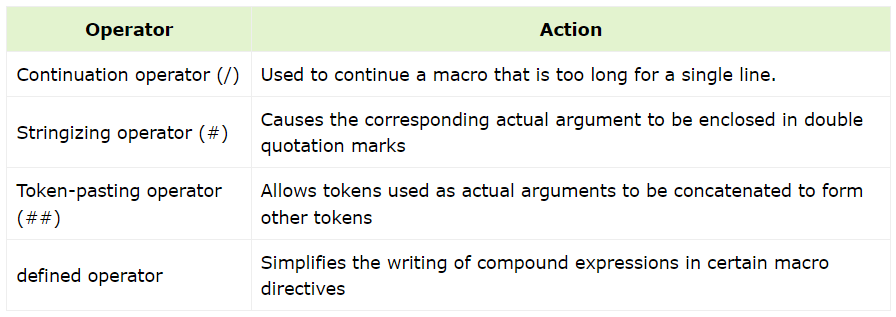
.C

.i

## DIFFERENCE BETWEEN COMPILERS VS INTERPRETERS IN C LANGUAGE?

## Preprocessor Operators in C

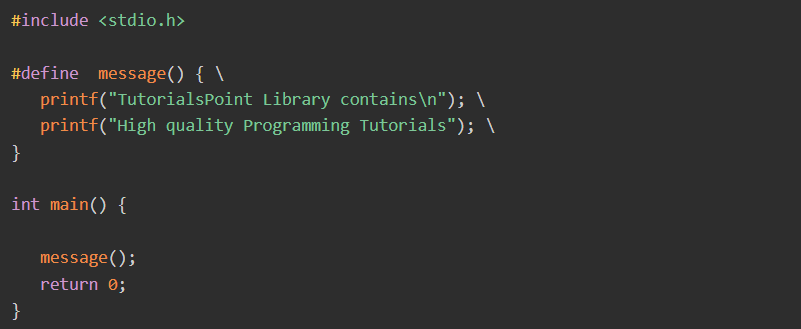
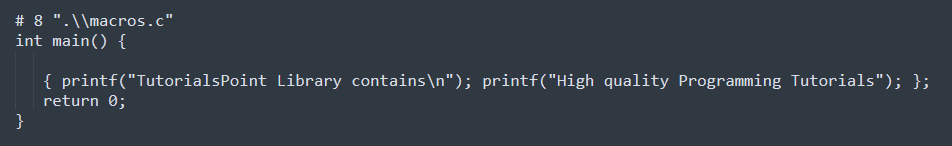
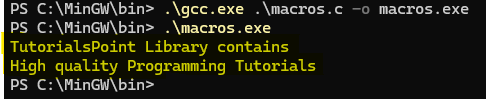
**Preprocessor operators** are special symbol(s) that are used in the context of the **#define** directive. These operators are also called preprocessor-specific operators.



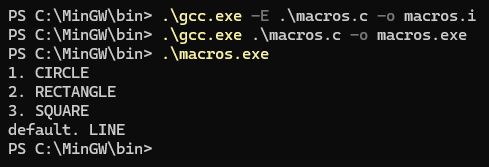
## Continuation Operator (/)

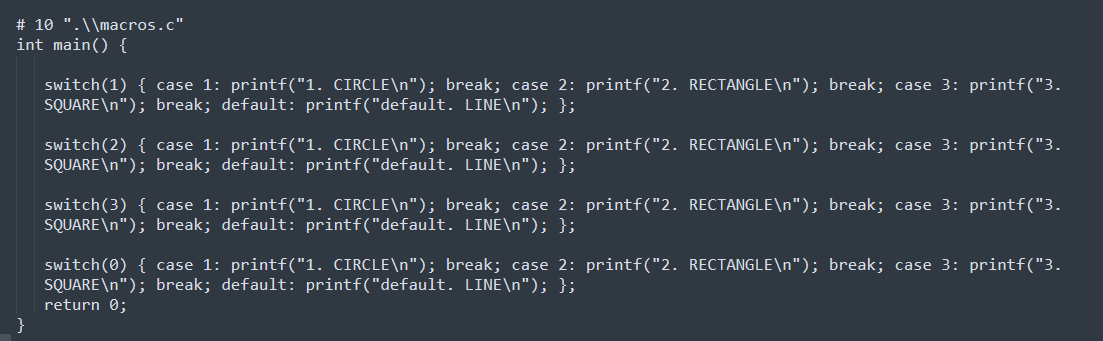
This operator is used where the macro is quite complex, and spreads over multiple lines. In case of a complex logic inside macro expansion, you’ll need to break the line and write code that spreads over more than one lines. In such cases macro continuation operator is very helpful.

### **Example 1: Preprocessor Continuation Operator (/)**



### **Example 2: Preprocessor Continuation Operator (/)**

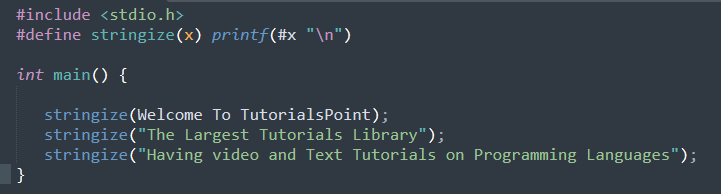




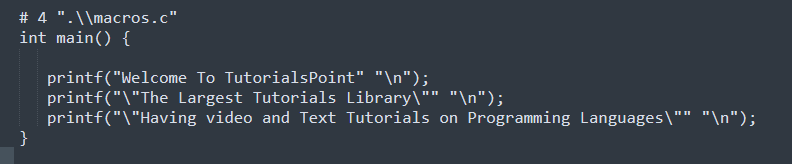
## Stringizing Operator (#)

Sometimes you may want to convert a macro argument into a string constant. The number-sign or "stringizing" operator (#) converts macro parameters to string literals without expanding the parameter definition. This operator may be used only in a macro having a specified argument or parameter list.

### **Example 1: Stringizing Operator**

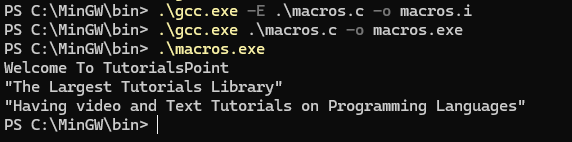


.C



**After preprocessing**

.i

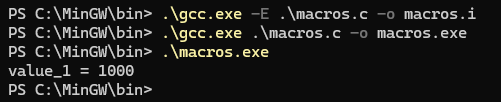


## Token Pasting Operator (##)

The double-number-sign or token-pasting operator (**##**), which is sometimes called the merging or combining operator. It is often useful to merge two tokens into one while expanding macros.

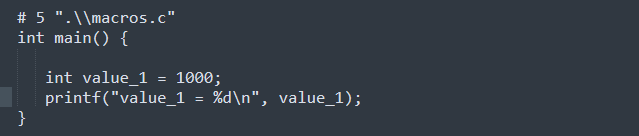
When a macro is expanded, the two tokens on either side of each "**##**" operator are combined into a single token, which then replaces the "**##**" and the two original tokens in the macro expansion.

### **Example 1: Token Pasting Operator (##)**



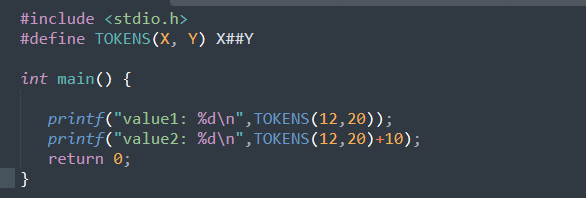
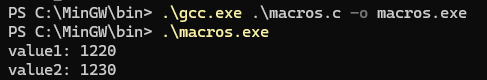
.C

**After preprocessing**



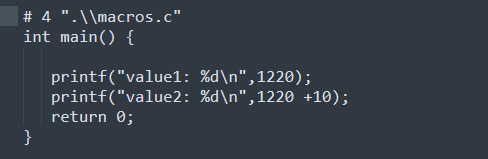
.i

### **Example 2: Token Pasting Operator (##)**



**After preprocessing**

.C



.i

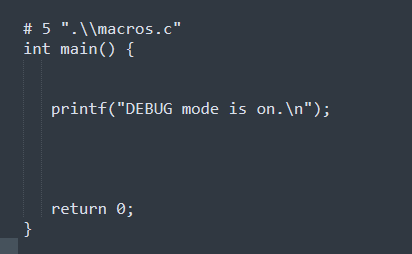
## The defined Operator

The defined preprocessor operator can only be used as a part of **#if** and **#elif** directives.



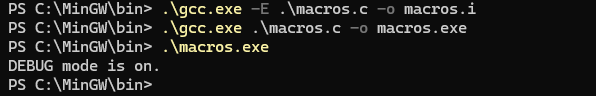
**After preprocessing**

### **Example 1: defined Operator**

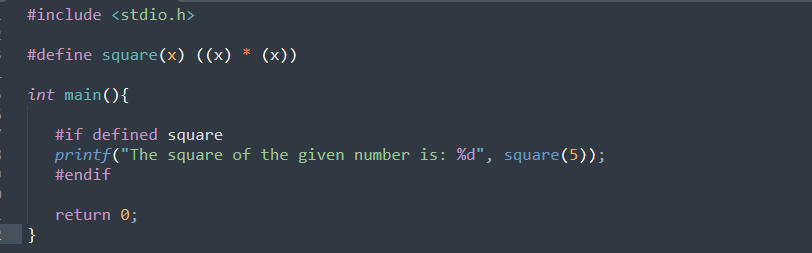


.C

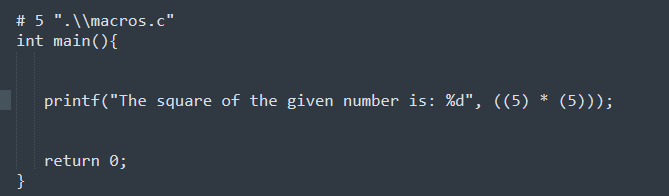
.i



### **Example 2: defined Operator**

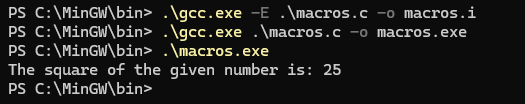


.C



**After preprocessing**

.i



# Macros in C

**Macros in C** are the names given to specific constant values or code statements which are replaced with their value/code before the compilation processor. **C Macros** are defined using the **#define** preprocessor directive.

**Macros** are useful for code reusability, defining constant values, defining inline functions, and conditional compilations.

*The following are the different types of C macros that we are going to cover in this tutorial –*

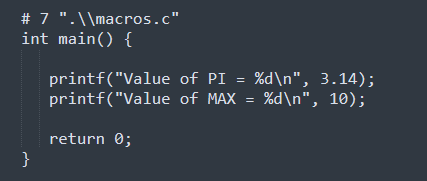
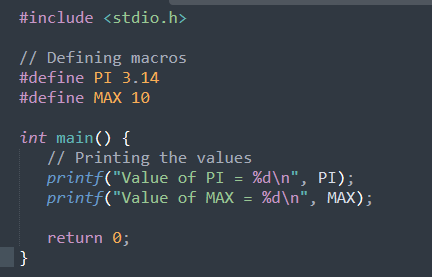
1. Object-like Macros
2. Function-like Macros
3. Chained Macros
4. Variadic Macros
5. Predefined Macros

## Object-like Macros

A macro that defines a constant is an object-like macro.

**After preprocessing**

### **Example of Object-like Macros**



.C

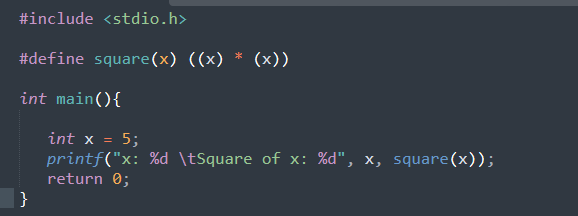
.i



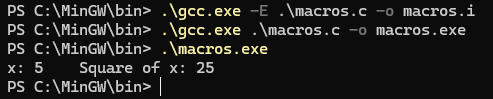
## Function-like Macro

To define a function-like macro, you use the same **"#define"** directive, but you put a pair of parentheses immediately after the macro name, with one or more arguments. Such a macro is expanded only when its name appears with a pair of parentheses after it.

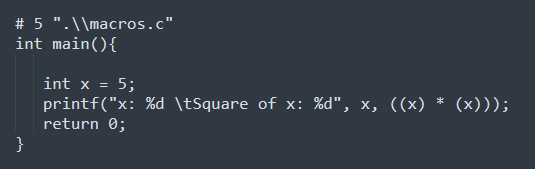
### **Example of Function-like Macros**



.C



**After preprocessing**



.i

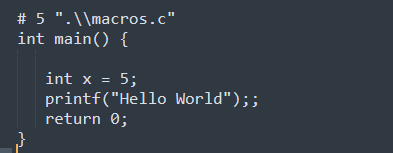
### **Rules for Defining Function-like Macros**

* A macro can be defined without arguments
* A macro can be defined with a fixed number of arguments
* A macro can be defined with a variable number of arguments

### **Function-like Macros without Arguments**

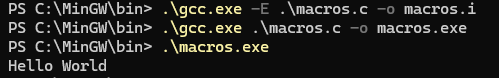
**After preprocessing**

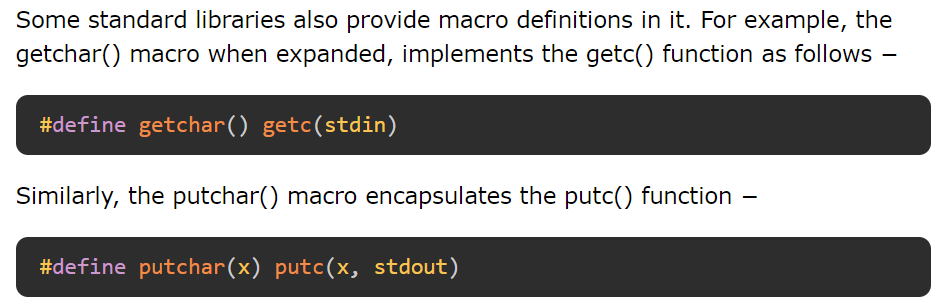
### **Example 1**



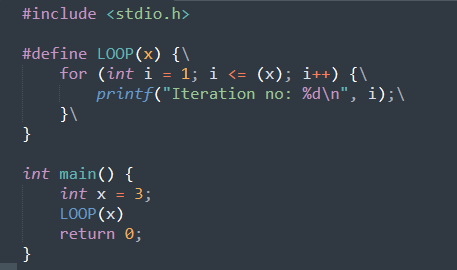
.C

.i

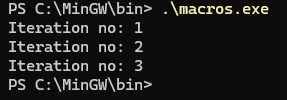


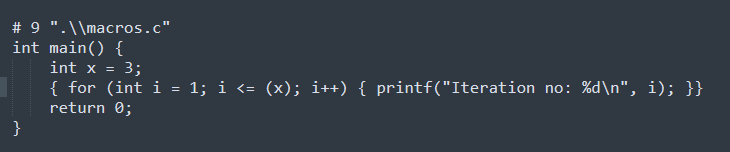


### **Example 2**



.C



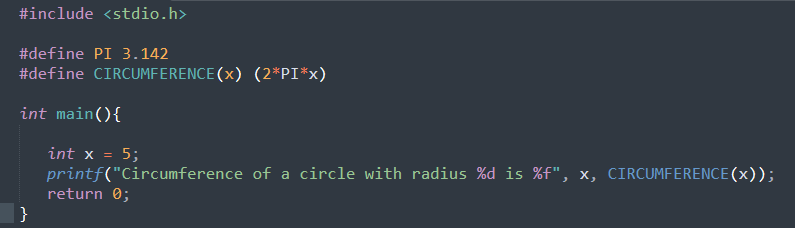


**After preprocessing**

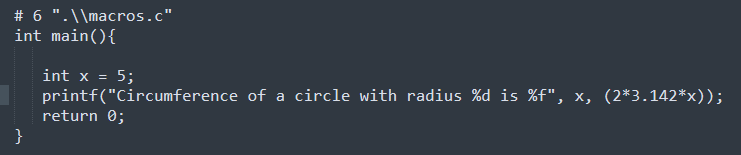
.i

## Chained Macros

When we have a macro nested inside another macro, they are called Chained Macros.

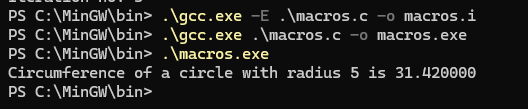


.C



**After preprocessing**

.i



## Variadic Macros

You can also define a macro with variable number of arguments or variadic macros.

A macro with variable-length argument is a feature that enables a macro to accept any number of arguments. You can pass positional as well as keyword arguments to a macro.

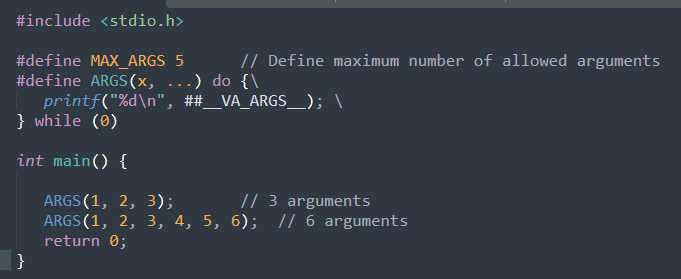
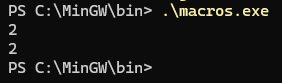
When a variadic macro is defined, the ellipsis (…) is given as an argument to capture variable number of arguments.

To use variadic macros, the ellipsis may be specified as the final formal argument in a macro definition.

This sequence of tokens replaces the identifier \_\_VA\_ARGS\_\_ in the macro body wherever it appears.

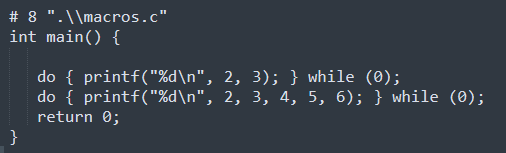
The \_\_VA\_ARGS\_\_ is replaced by all of the arguments that match the ellipsis, including commas between them. Note that the variadic macros can be used only in the C99 compatible C compilers and above.

### **Example of Variadic Macros**



**After preprocessing**

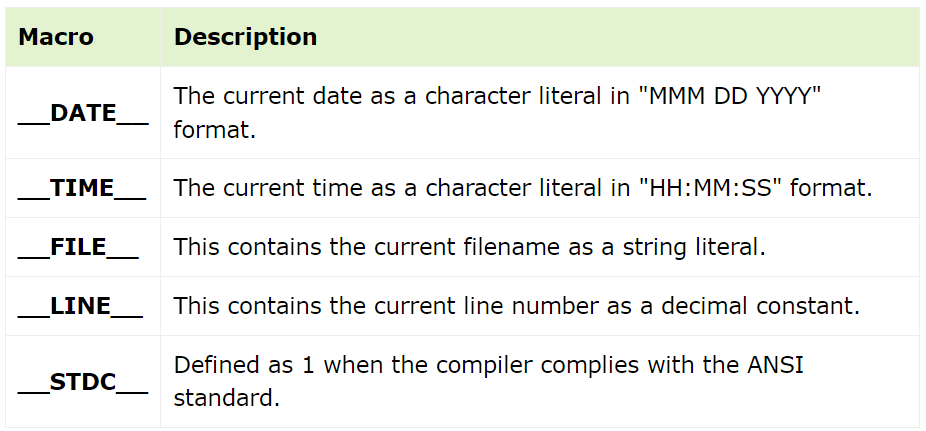
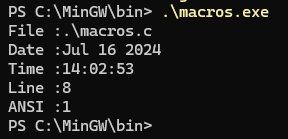
.C

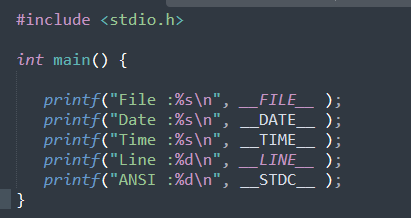


.i

## Predefined Macros

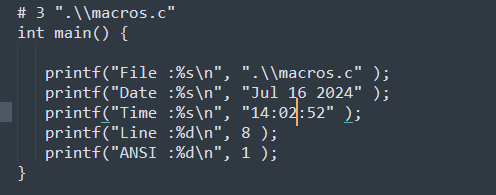
ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.





.C

**After preprocessing**



.i

**Quiz:**

**Create a function factory and use it with a macro.**

* Use one macro to create two functions: one that multiplies the input argument by 4 and another that multiplies the input argument by 2.
* The name of the first function will be fun\_quadruple(int x).
* The name of the second function will be fun\_double(int x).

# 

.C

**After preprocessing**

# 

.i

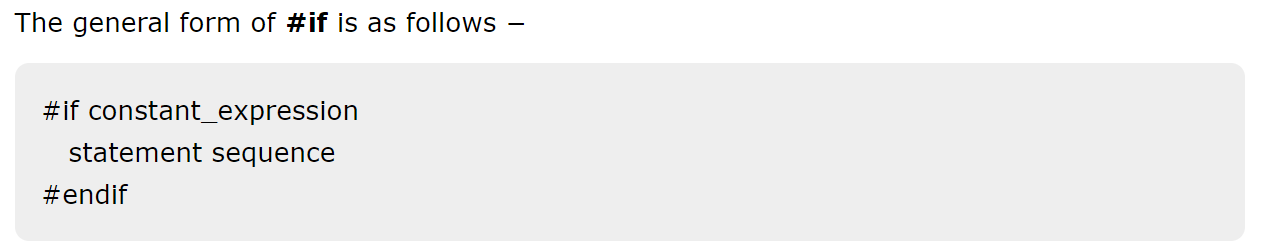
# 

# 

# conditional compilation in C

# n C programming language, several directives control the selective compilation of portions of the program code.

* #if
* #else
* #elif
* #endif

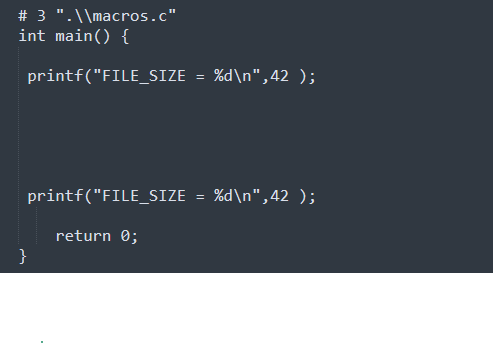


else works much like the C keyword else.

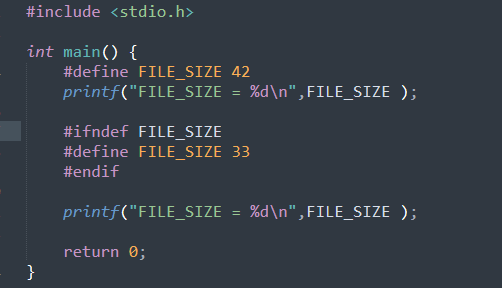
#elif means "else if" and establishes an if else-if compilation chain.

Amongst other things, #if provides an alternative method of "commenting out" code.

# 

* The following directive tells the compiler to define FILE\_SIZE only if FILE\_SIZE isn't already defined.

**After preprocessing**



.i

.c

.C

# Macros Vs Functions

# 

# Header Files in C

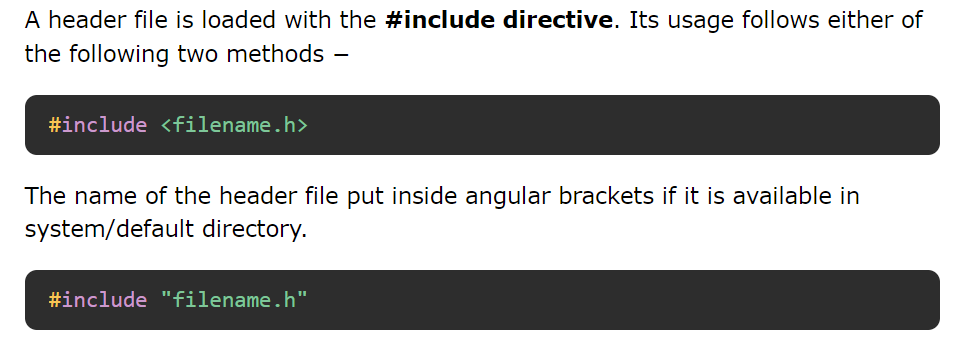
The **#include preprocessor** directive is used to make the definitions of functions, constants and macros etc. from one file, usually called as a header file, available for use in another C code. A header file has "**.h**" extension from which you can include the forward declarations of one or more predefined functions, constants, macros etc. The provision of header files in C facilitates a modular design of the program.

## System Header Files

The C compiler software is bundled with many pre-compiled header files. These are called **system header files**. A well-known example is "stdio.h" – a header file included in almost every C program.

Each of the system header files contains a number of utility functions. These functions are often called **library functions**. For example, printf() and scanf() functions, needed for performing IO operations, are the library functions available in the "stdio.h" header file.

## Syntax to Include Header Files in C



The name of the header file put inside double quotation marks for user defined or non-standard header files available in same directory as source file

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive.

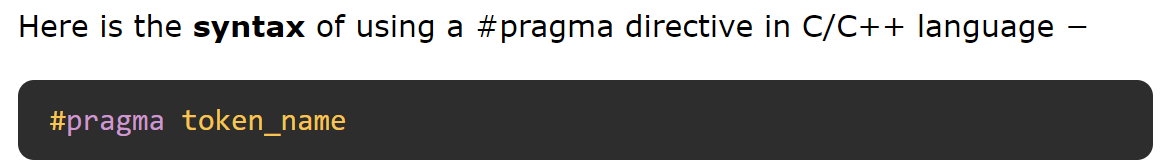
# #pragma Directive in C

## What is #pragma Directive in C?

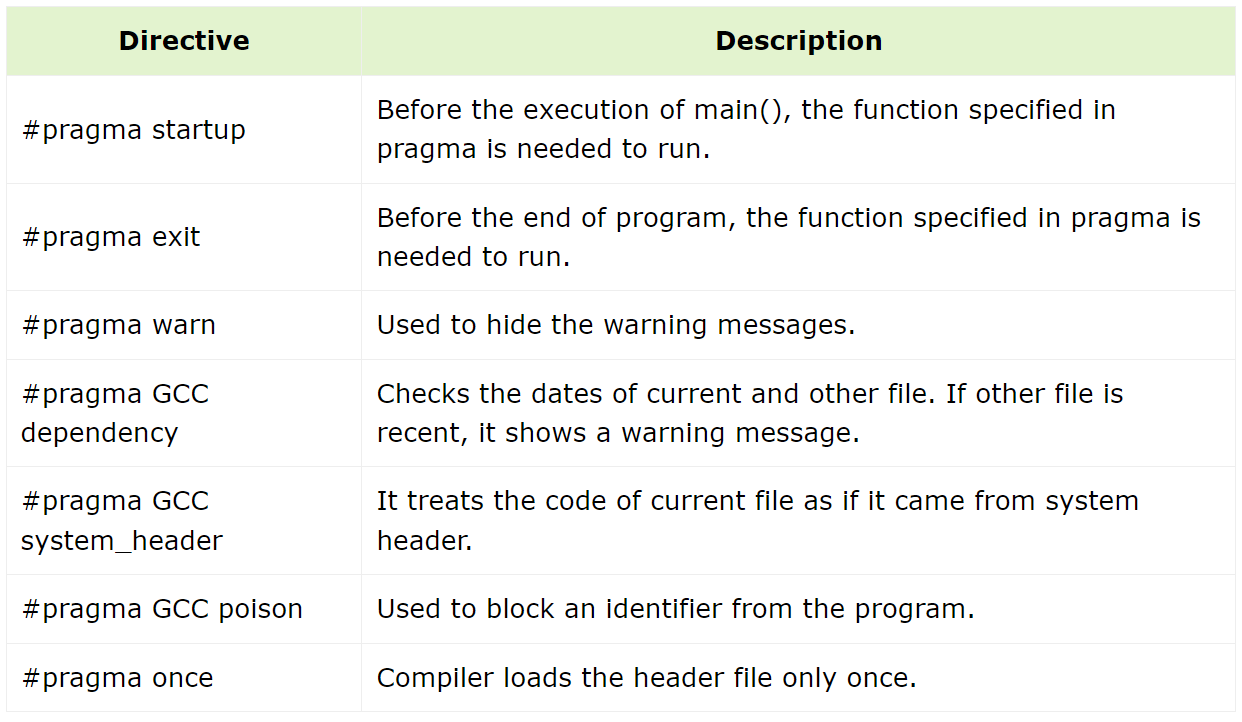
The preprocessor directive **#pragma** is used to provide additional information to the compiler in C/C++ language. This is used by the compiler to provide some special features.

Note that **pragmas are compiler dependent**. Not all the pragma directives are supported by all the compilers.

### **Syntax**



## Types of Pragma Directives in C

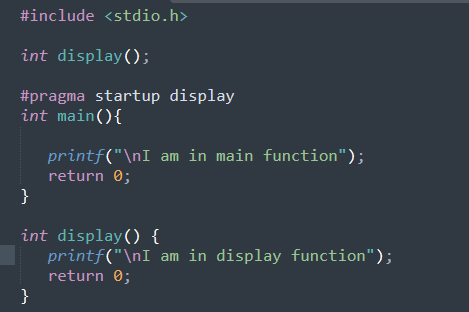
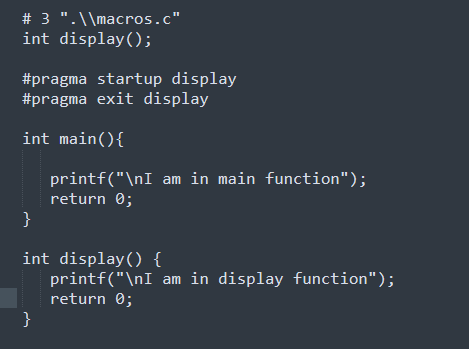


## #pragma startup and exit

These pragma directives are executed before and after the main() function. Not all the compilers support these directives.

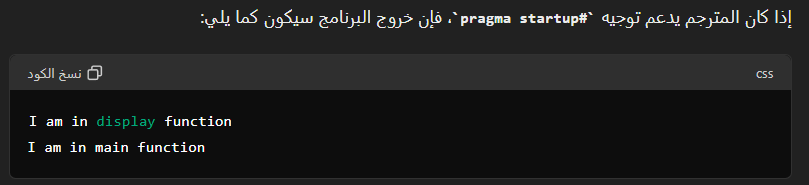
**After preprocessing**

### **Example**



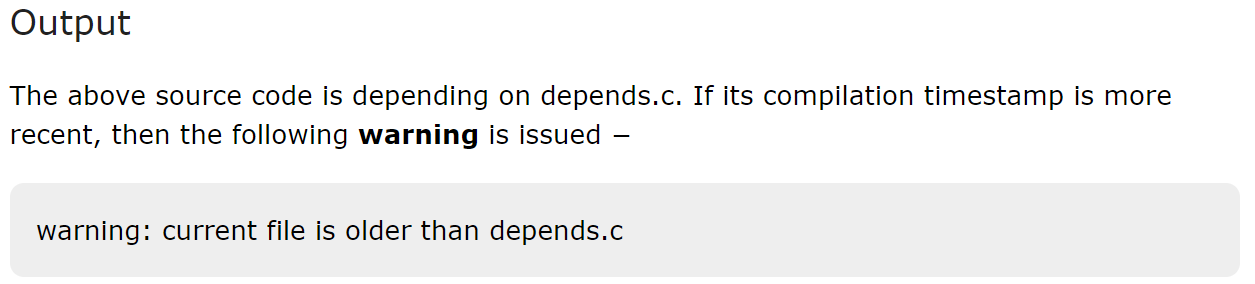
.i

.c



## #pragma GCC dependency

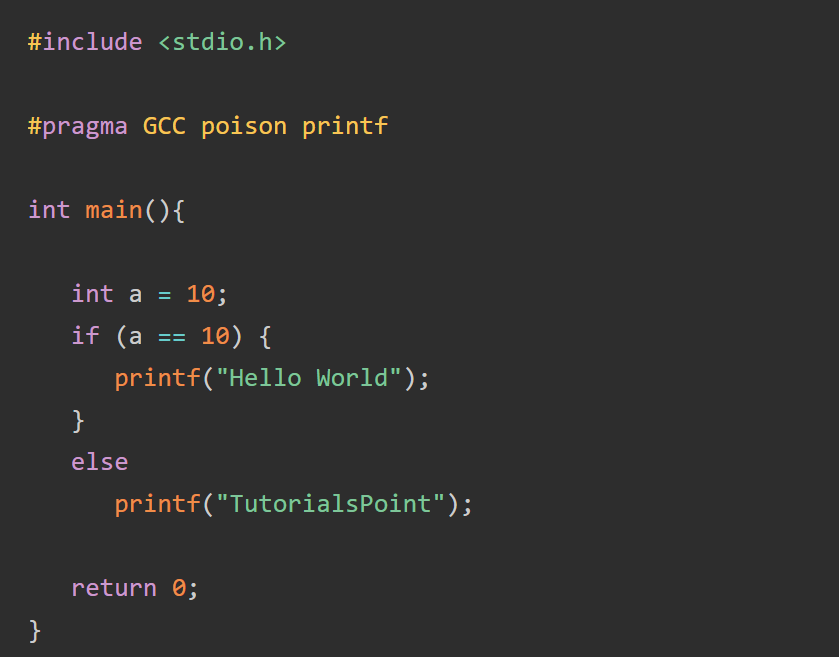
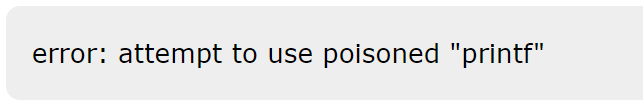
This pragma allows you to check the relative dates of the current file and another file. If the other file is more recent than the current file, a warning is issued.



## #pragma GCC poison

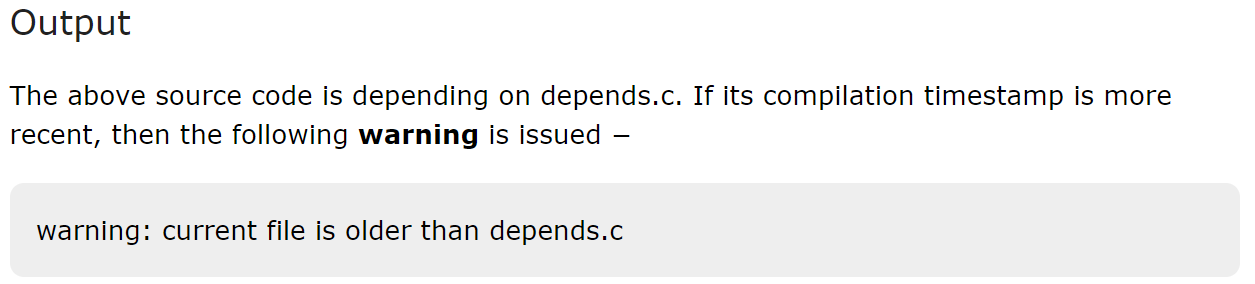
The GCC compiler removes an [identifier](https://www.tutorialspoint.com/cprogramming/c_identifiers.htm) completely from the program. If we want to block an identifier, then we can use the **#pragma GCC poison** directive. Its syntax is as follows −  


### **Example**



## #pragma GCC dependency

## This pragma allows you to check the relative dates of the current file and another file. If the other file is more recent than the current file, a warning is issued.



## #pragma once

The **#pragma once** directive causes the header file to be included only once, even if the programmer includes it multiple times.



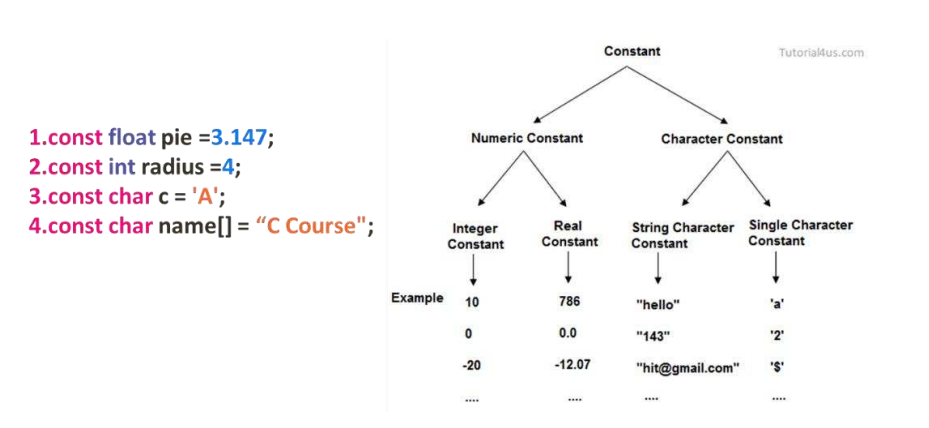
# Type qualifier in C

**There are two important type qualifiers in C**

* const
* volatile

Applying these qualifiers to variable declaration is called **qualifying the declaration**. The qualifier allows the programmer to add some **features to a variable.**

# C – Constants

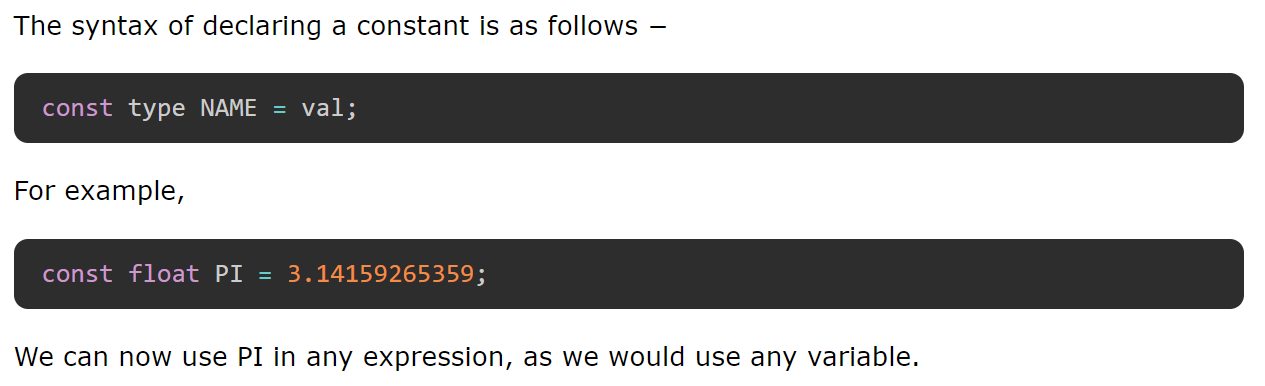
* The const type qualifier in C is used to enforce read only features on variables
* **const** doesn’t mean that the value **NEVER** changes, its only programming **safety feature** to ensure that the programmer shouldn’t try to modify the value.
* All **const variables** are **stored in memory in the same way as standard variables**.
* They are placed in **RAM**. The only difference, in this sense, is the **read-only feature**.
* All **global const variables** are stored in **ROM** or **FLASH memory**. This also further depends on linker script rules and the hardware on which code runs.
* The flash memory of the micro-controller is indeed **write-protected**, that means the operation won’t have any effect.
* **In the **PC**, the program crashes because we’re trying to write in the write-protected section.

*Instead of repeatedly using hard-coded values in a program, it is advised to define a constant and use it.****Constants****in a C program are usually employed to refer to a value which may be error-prone if it is to be used repetitively in the program, at the same time its value is not likely to change.*

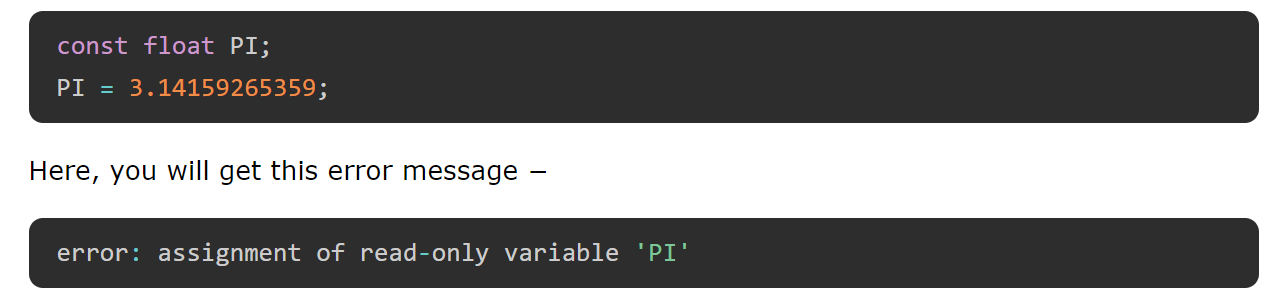
**You an declare a constant in C program with either of the following two ways −**

* Using the const Keyword
* Using the #define Directive

## Defining a Constant Using the const Keyword



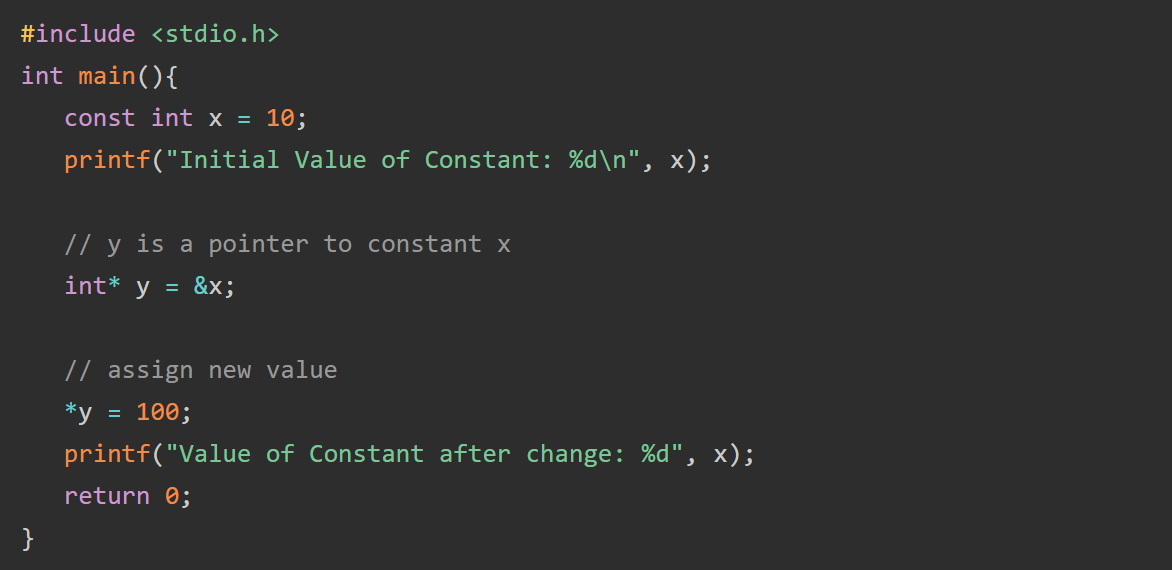
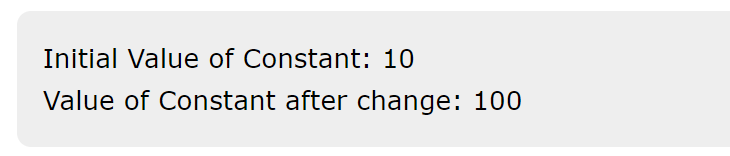


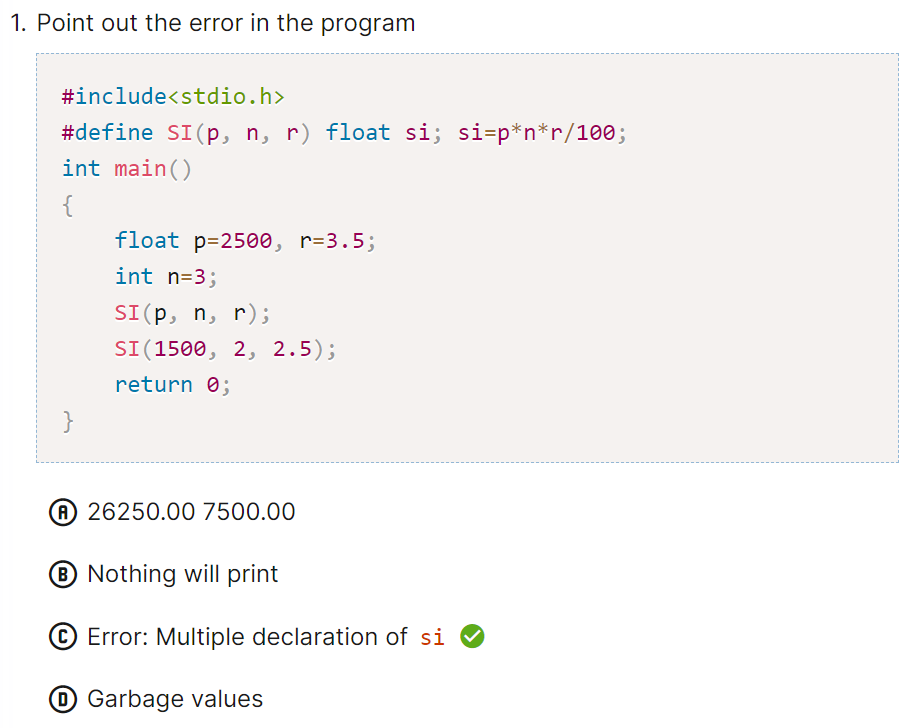
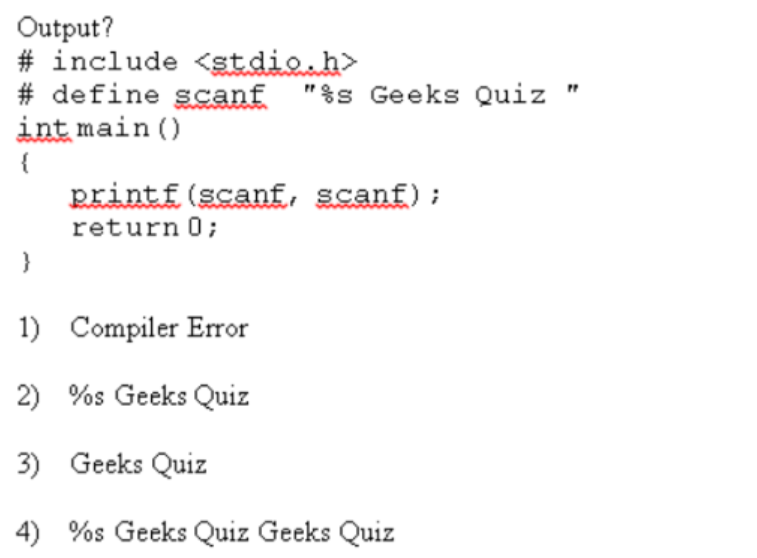


## Changing the Value of a Constant

The technique uses the concept of pointers in C. A Pointer is a variable that stores the address of another variable. Since it is a variable, its value can be changed. Moreover, this change reflects in the original variable.

### **Example: Change the Value of a Constant**



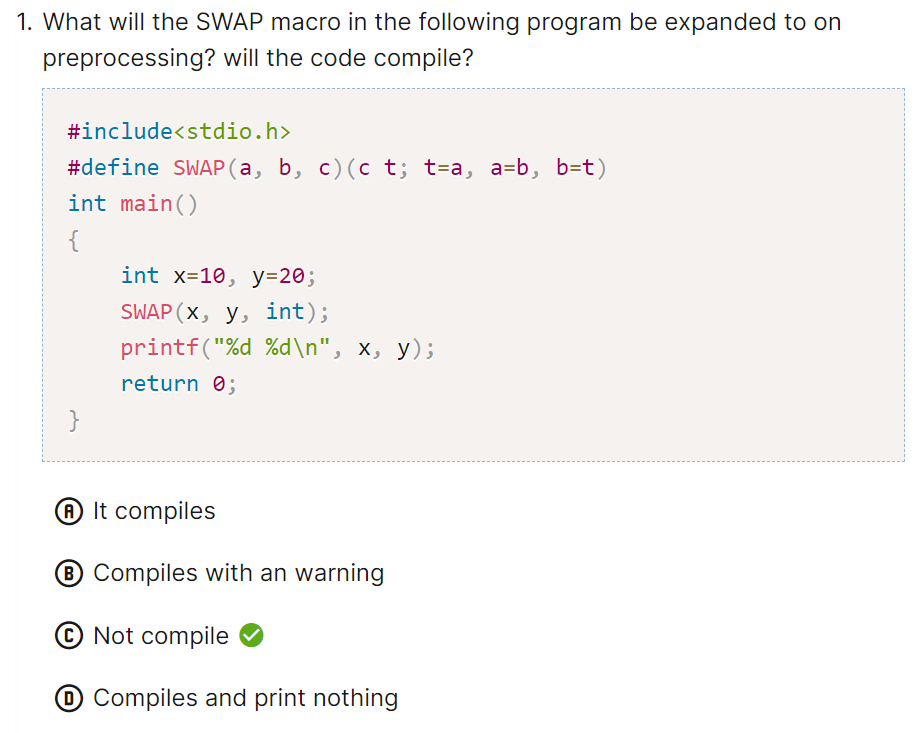
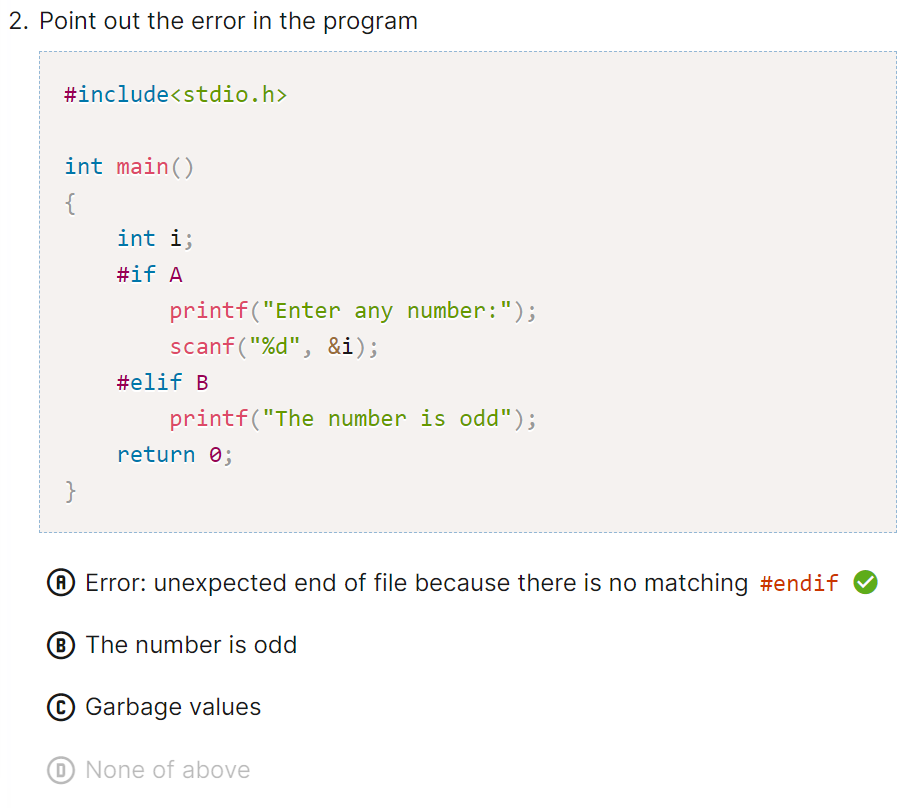


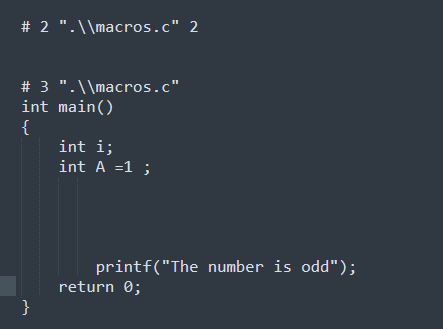
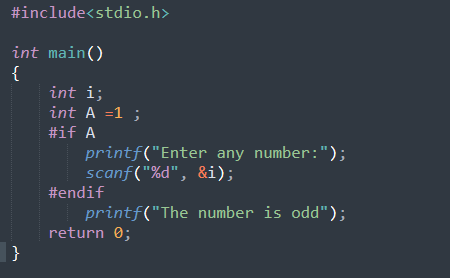
**EX**

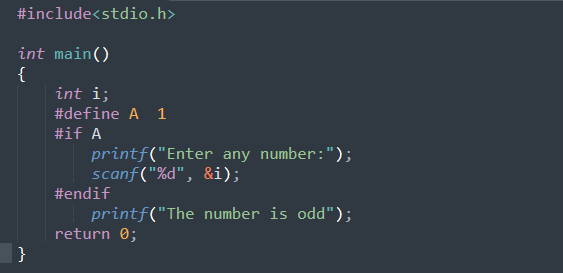
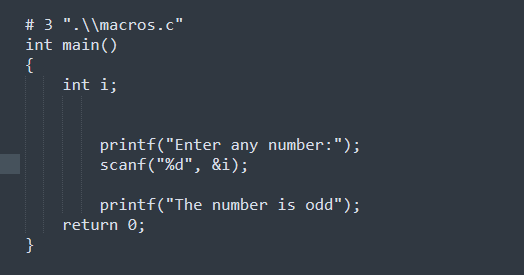
**EX**

**EX**

**EX**



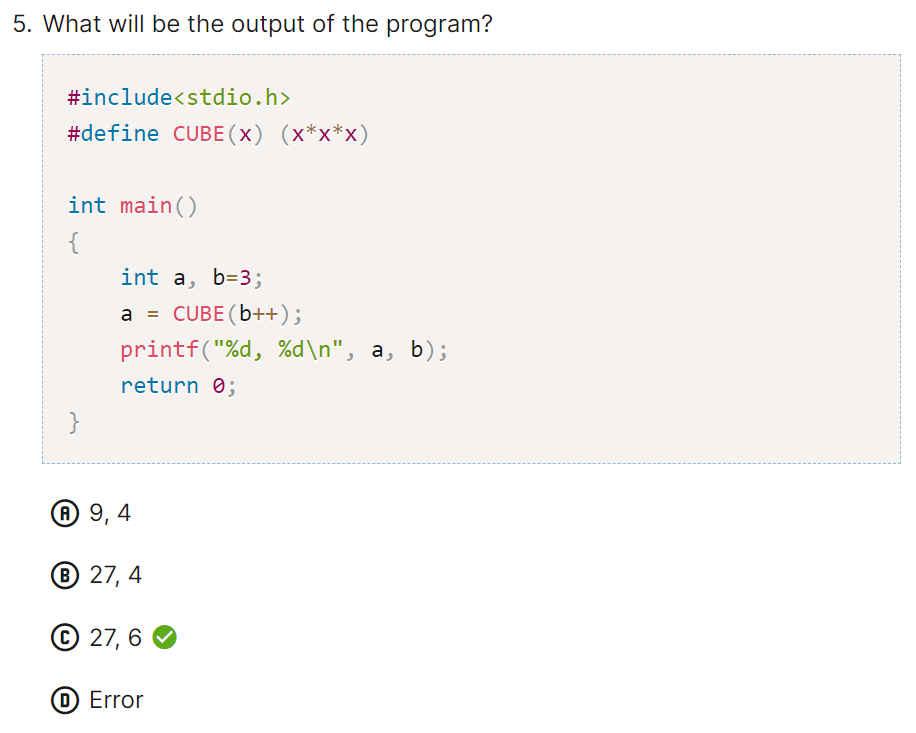


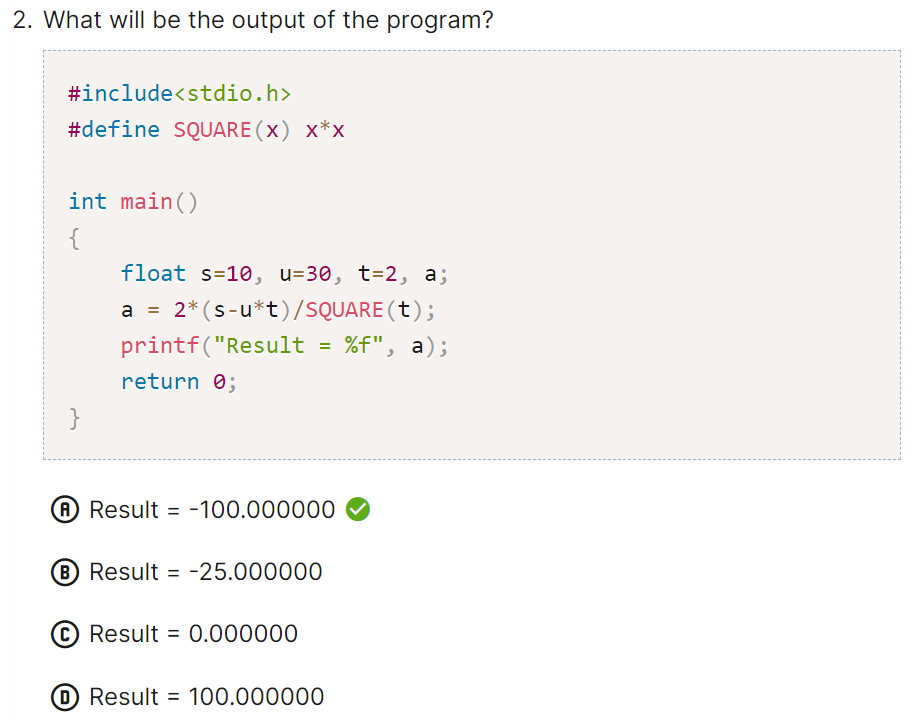




**EX**

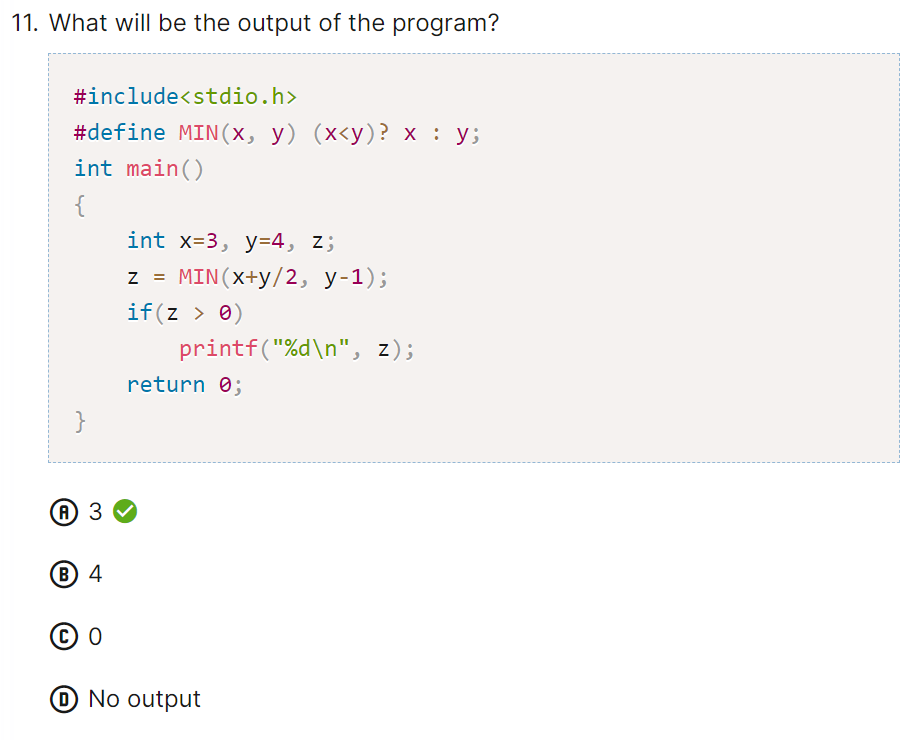
**EX**

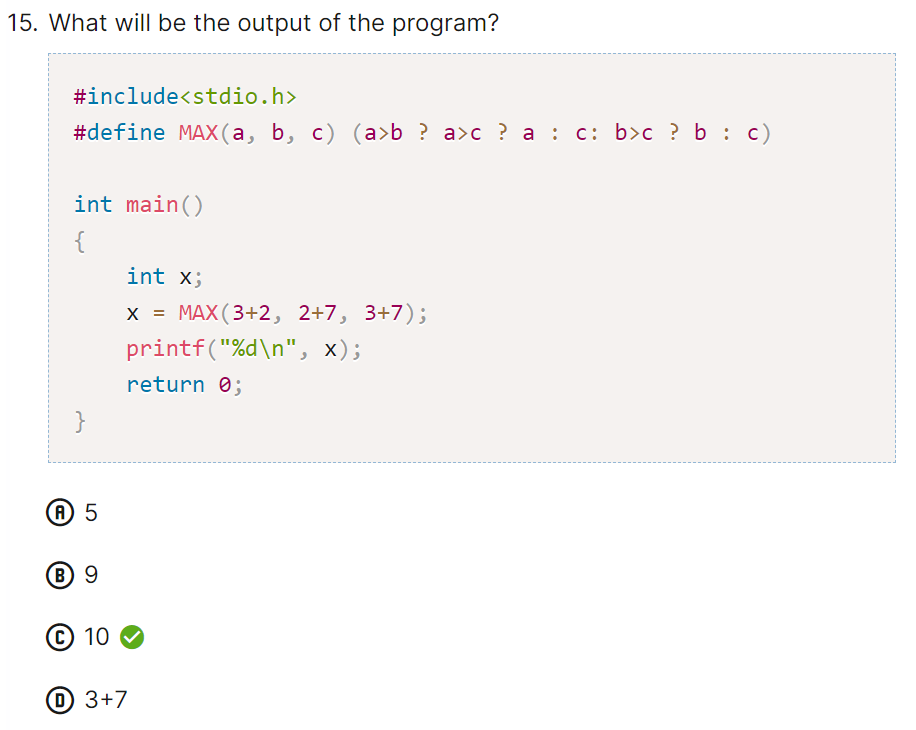
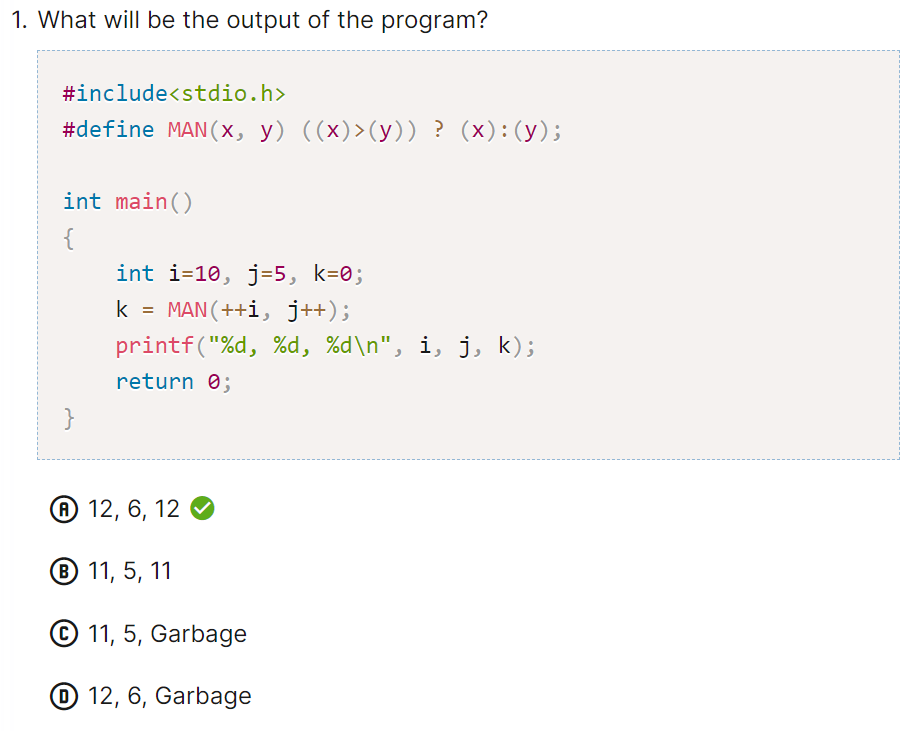




**EX**

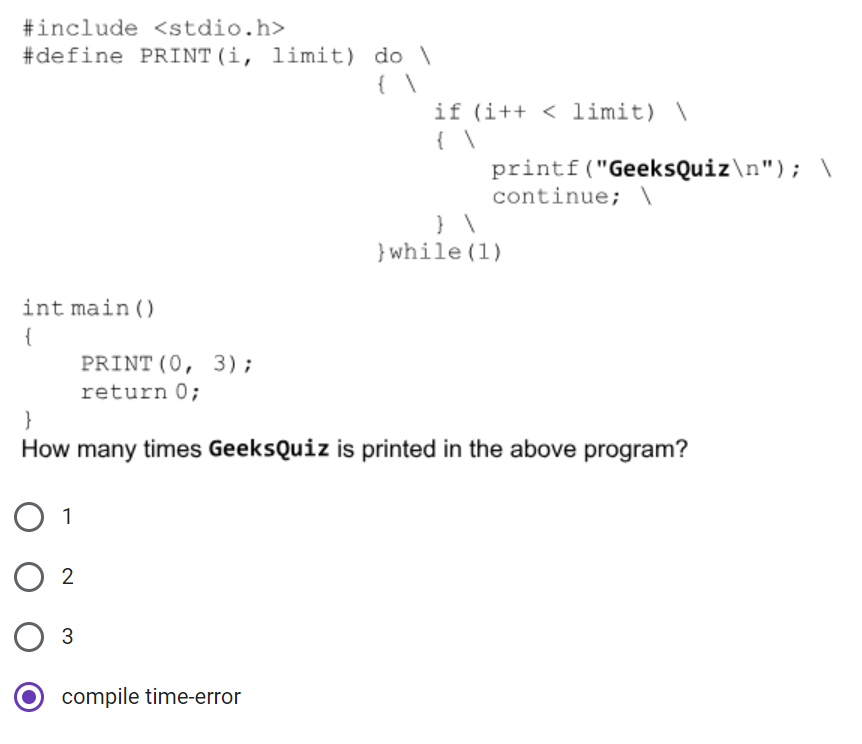
**EX**



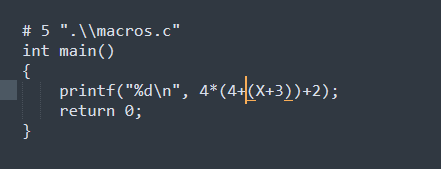


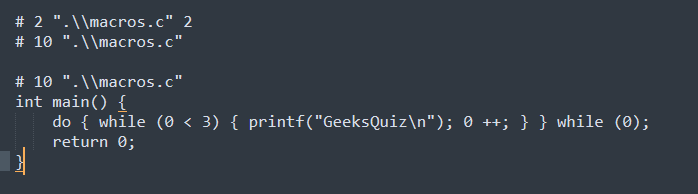
**EX**

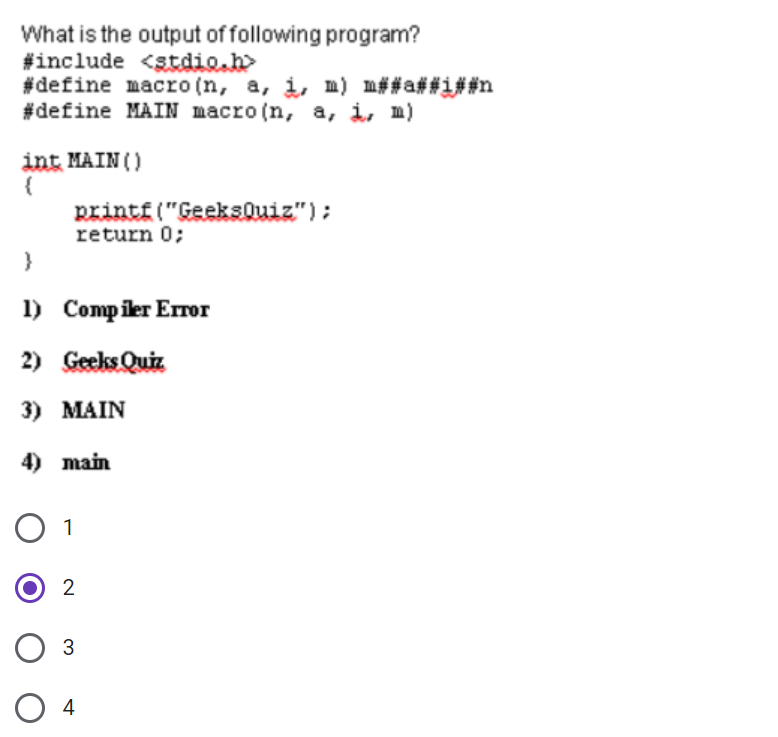
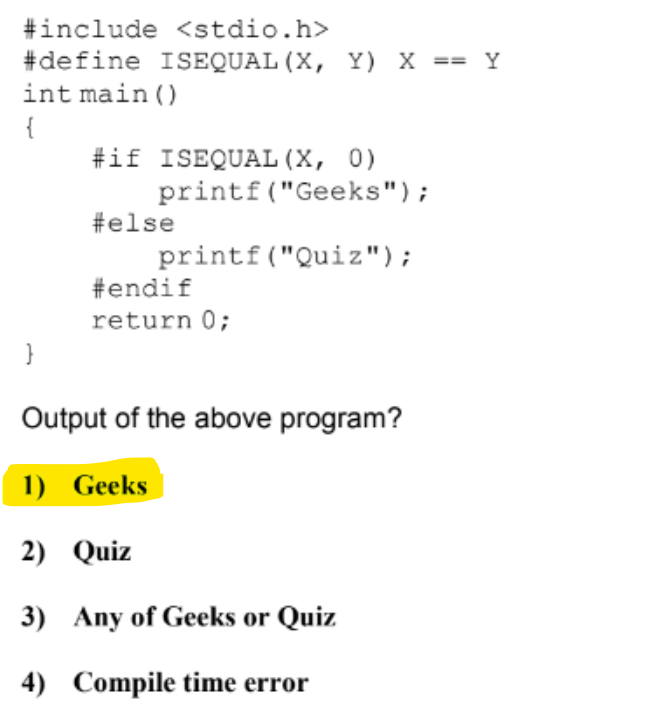
**EX**



**EX**

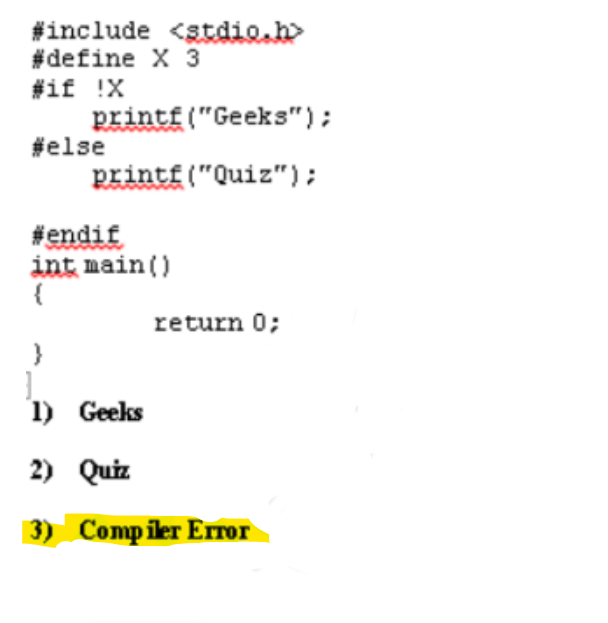






**EX**

**EX**



**EX**

The conditional macro #if ISEQUAL(X, 0) is expanded to #if X == 0. After the pre-processing is over, all the undefined macros are initialized with default value 0. Since macro X has not been defined, it is initialized with 0. So, Geeks is printed.

[](https://github.com/Abdallah-Ghazy)

[](https://www.linkedin.com/in/abdallah-ghazy/)

[](https://www.facebook.com/profile.php?id=100009485341470)

[](https://www.youtube.com/channel/UCRh59pwh7KTLgfftifu_zrQ)

" من ضيع الأصول حرم الوصول ومن ترك الدليل ضل السبيل"