PREPROCESSOR DIRECTIVES

- A preprocessor is a program that processes our source program before it is passed to the compiler.
- The preprocessor works on the source code and creates expanded source code.
- The preprocessor offers several features called **preprocessor directives.**
- Preprocessor starts with # symbol.
- The directives can be placed anywhere in a program but are most often placed at the beginning of the program, before the function definition.

Types of preprocessor directives.

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the program
#ifdef	Used to check whether a macro is defined by #define or not
#ifndef	Used to check whether a macro is not defined by #define or not
#if	Used to check specified condition
#elif	Alternate code that executes when #if or #elif fails
#else	This block executes when all #if & #elif fails
#endif	Used to terminate conditional statements

1. FILE INCLUSION

- File inclusion allows us to include the contents of one source code file into another.
- This is done using the #include directive, which tells the C preprocessor to insert the contents of the specified file into the current file at the point where the #include directive appears.
- File inclusion is commonly used to separate code into different files for better organization and maintainability.
- It also allows us to reuse code that is common to multiple source files without having to duplicate it.

Syntax:

For Standard Library Inclusion: #include <header_file_name.h>

For Custom Header File Inclusion: #include "header_file_name.h"

Standard Library Inclusion:

The standard library inclusion is used to include standard header files that define functions, constants, and macros used in C programming.

These header files are usually provided by the compiler or the operating system, and are included using angle brackets (<>) as follows:

Ex:

```
#include<stdio.h>
void main(){
    printf("Hello world");
}
```

Output: Hello world

Explanation:

We include the standard I/O header file stdio.h using the #include directive.

This header file defines the printf() function, which we use to print "Hello, world!" to the console.

Because we included the header file using the angle bracket notation (< >), the compiler will search for the header file in its standard library directories.

Custom Header File Inclusion:

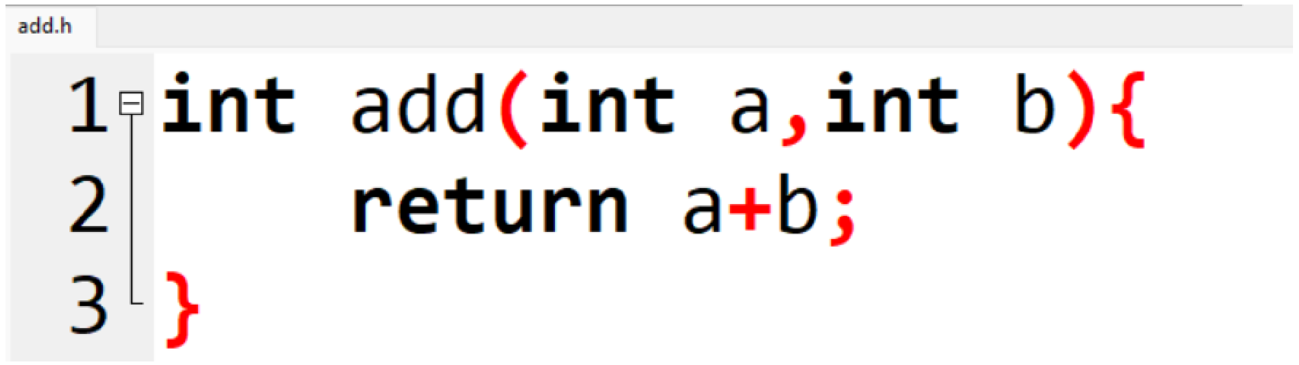
Custom header files are created by the programmer to define functions, constants, and macros that are specific to a particular program or module.

These header files are included using double quotes ("")

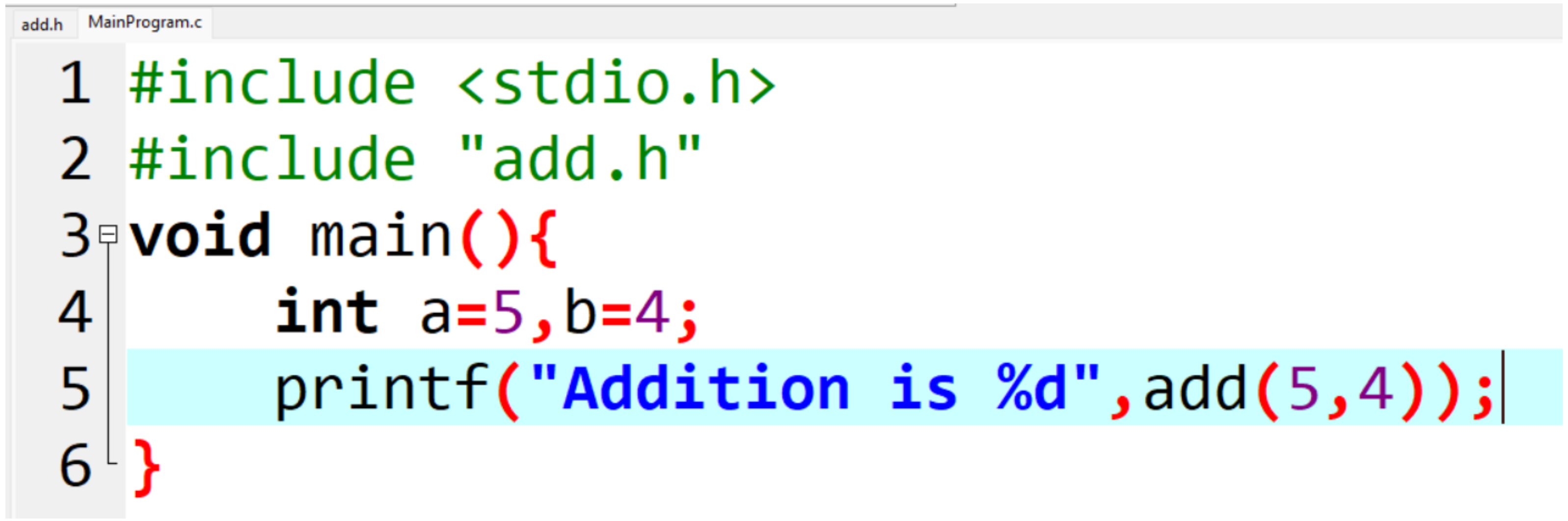
When a file is included using the #include directive, its contents are treated as if they were part of the current file.

This means that any function or macro definitions in the included file are available for use in the current file, and any global variables in the included file are visible in the current file.

Ex:

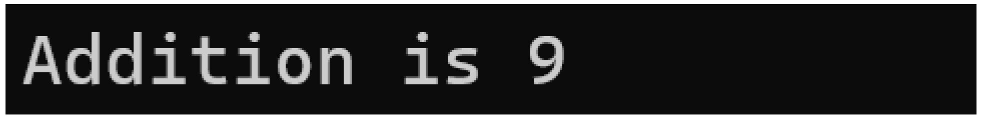


```c
int add(int a,int b){
    return a+b;
}
```



```c
#include <stdio.h>
#include "add.h"
void main(){
	int a=5,b=4;
	printf("Addition is %d",add(5,4));
}
```

Output:



```
Addition is 9
```

We include a custom header file add.h using the #include directive.

This header file contains a function add() that takes two integers as input and returns their sum.

We use this function to calculate the sum of a and b, and then print the result to the console using printf().

Because we included the header file using the double quote notation (" "), the compiler will search for the header file in the current directory or in any directories specified by the -I flag at compilation time.

2. MACROS

- A macro is a piece of code in a program that is replaced by the value of the macro.
- Macro is defined by **#define** directive.
- Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.

```c
#include <stdio.h>
#define pi 3.14
void main(){
    int r;
    float area;
    printf("Enter radius of circle: ");
    scanf("%d",&r);
    area = pi*r*r;
    printf("Area of circle is %.2f",area);
}
```

Types of macros in C

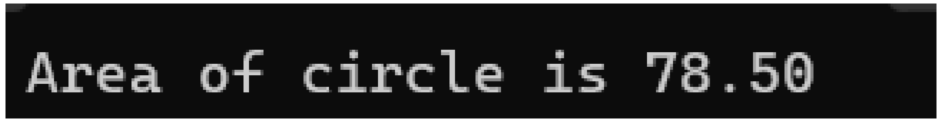
1. **Object-Like Macros:**

An object-like macro is a simple identifier that will be replaced by a code fragment. It is called object-like because it looks like an object in code that uses it. It is popularly used to replace a symbolic name with a numerical/variable represented as a constant.

Ex:

```c
#include<stdio.h>
#define pi 3.14
void main(){
    int r=5;
    float area = pi*r*r;
    printf("Area of circle is %.2f",area);
}
```

Output:



```
Area of circle is 78.50
```

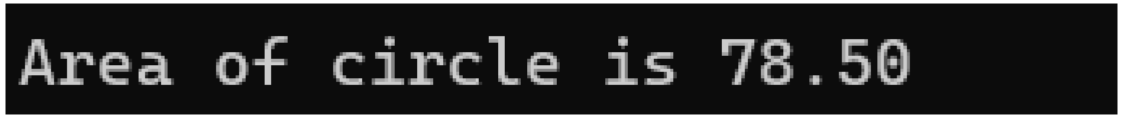
2. Chain Macros:

Macros inside macros are termed chain macros. In chain macros first of all parent macro is expanded then the child macro is expanded.

Ex:

```c
#include<stdio.h>
#define pi_value pi
#define pi 3.14
void main(){
    int r=5;
    float area = pi*r*r;
    printf("Area of circle is %.2f",area);
}
```

Output:

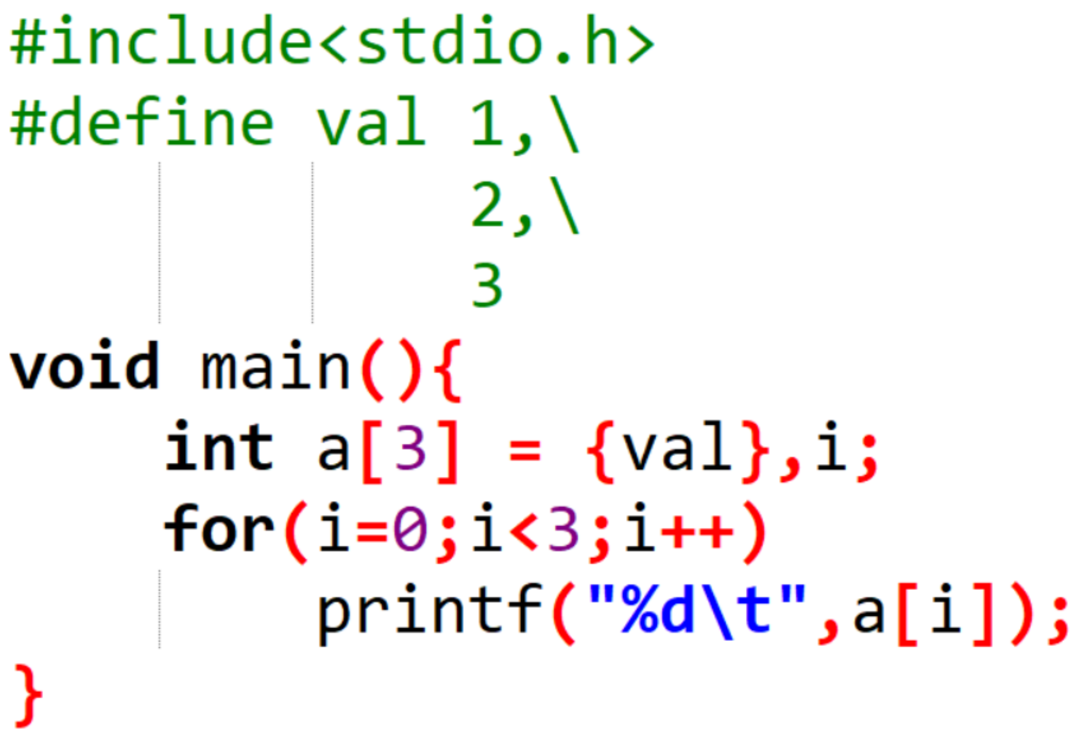


```
Area of circle is 78.50
```

3. Multiline Macros:

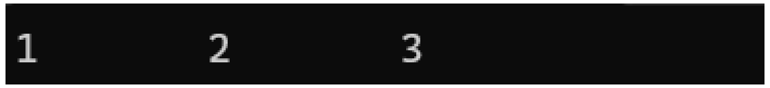
An object-like macro could have a multi-line. So to create a multi-line macro you have to use backslash-newline.

Ex:



```
#include<stdio.h>
#define val 1,\
            2,\
            3
void main(){
    int a[3] = {val},i;
    for(i=0;i<3;i++)
        printf("%d\t",a[i]);
}
```

Output:



```
1       2       3
```

4. **Function-Like Macros:**

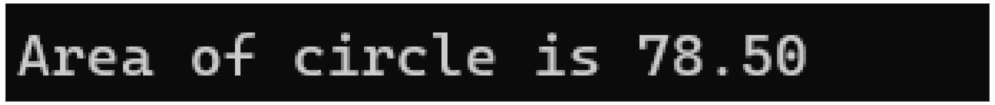
These macros are the same as a function call. It replaces the entire code instead of a function name. A pair of parentheses immediately after the macro name is necessary.

Note: If we put a space between the macro name and the parentheses in the macro definition, then the macro will not work.

Ex:

```c
#include<stdio.h>
#define pi 3.14
#define area(r) (pi*r*r)
void main(){
    int r=5;
    printf("Area of circle is %.2f",area(r));
}
```

Output:



```
Area of circle is 78.50
```

3. CONDITIONAL COMPILATION

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

- #if
- #ifdef
- #ifndef

Optionally, an alternative block of text can be set aside with one of two directives:

- #else
- #elif

The end of the block or alternative block is marked by the #endif directive.

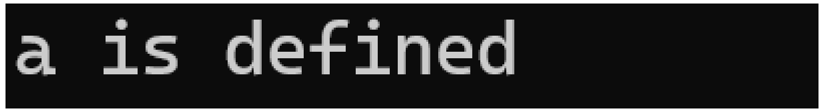
#ifdef:

This directive checks whether the identifier is defied or not. Identifiers can be defined by a #define directive or on the command line. If such identifiers have not been subsequently undefined, they are considered currently defined.

Ex:

```c
#include<stdio.h>
#define a 5
void main(){
    #ifdef a
        printf("a is defined");
    #else
        printf("a is not defined");
    #endif
}
```

Output:

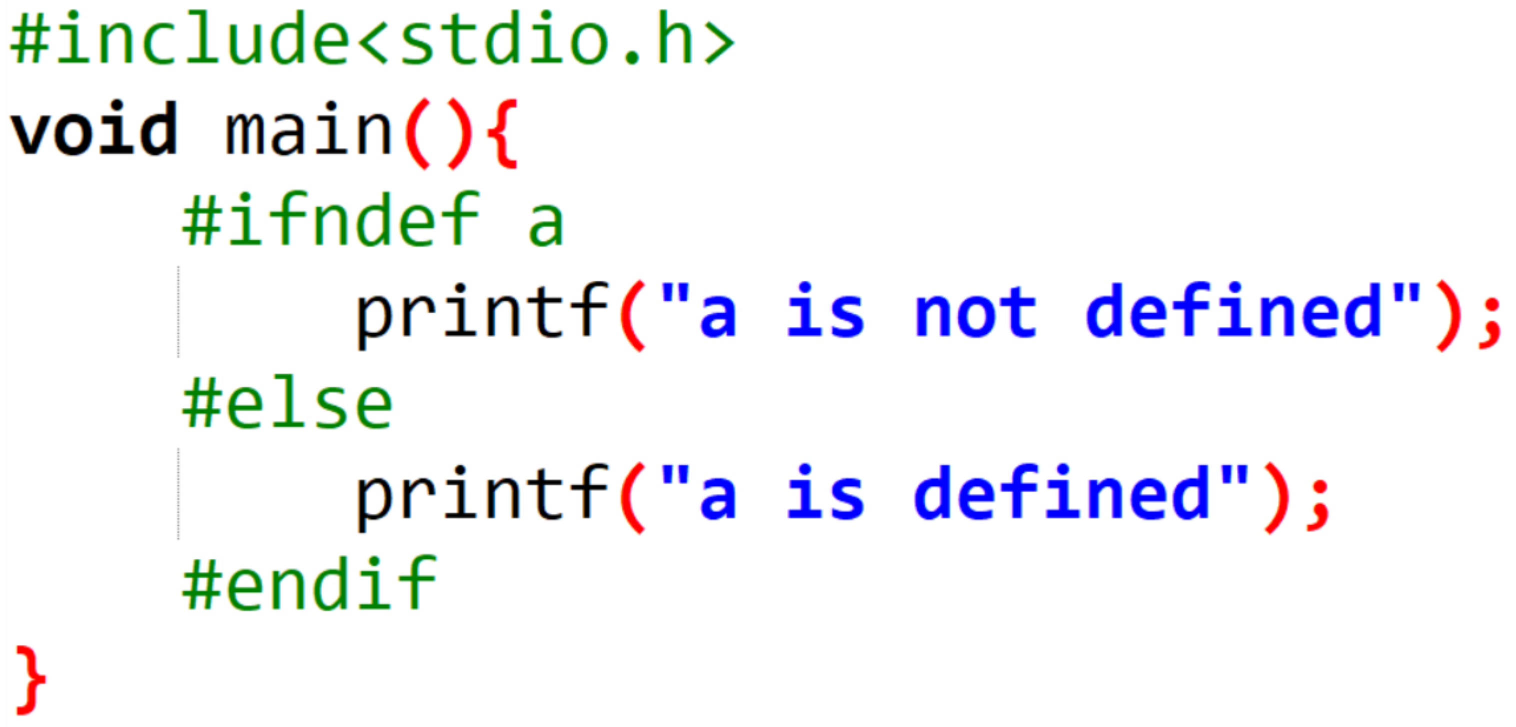


```
a is defined
```

#ifndef

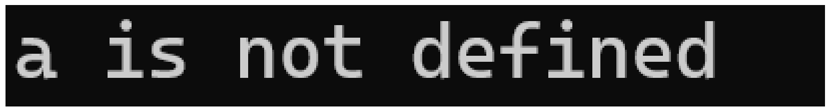
This directive checks to see if the identifier is not currently defined.

Ex:



```c
#include<stdio.h>
void main(){
    #ifndef a
        printf("a is not defined");
    #else
        printf("a is defined");
    #endif
}
```

Output:



```
a is not defined
```

#if:

This directive checks whether the constant-expression is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++), decrement (- -), sizeof , pointer (*), address (&), and cast operators.

#elif:

The #elif directive performs a task similar to the combined use of the else-if statements in C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding #if , #ifdef , #ifndef , or another #elif directive is false and if the additional constant expression presented in the #elif line is true. An #elif directive is optional

#else

This directive delimits alternative source text to be compiled if the condition tested for in the corresponding #if, #ifdef, or #ifndef directive is false. An #else directive is optional.

#endif

This directive ends the scope of the #if, #ifdef, #ifndef, #else, or #elif directive.

Ex:

```c
#include<stdio.h>
#define a 5
void main(){
    #if a>0
        printf("%d is positive number",a);
    #elif a<0
        printf("%d is negative number",a);
    #else
        printf("%d is zero");
    #endif
}
```

Output:

```
5 is positive number
```

4. PRAGMAS

- **#pragma** is a special purpose directive that is used to turn on or off some features. #pragma allows us to provide some additional information to the compiler**.**
- Pragmas are compiler specific i.e., the behavior of pragma varies from compiler to compiler.

1. #pragma startup and #pragma exit

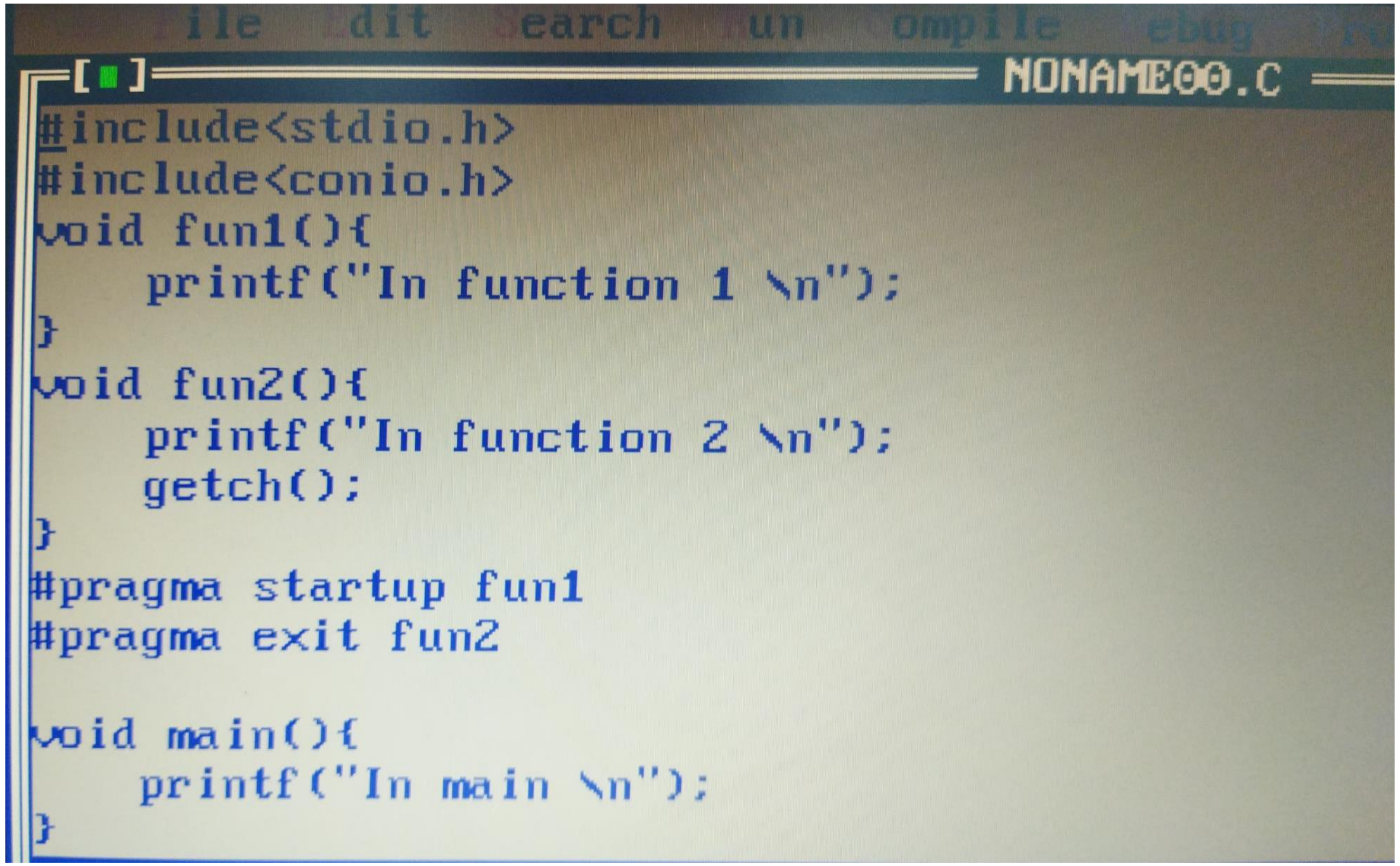
These directives help us to specify the functions that are needed to run before the program starts and just before the program exits i.e., before the control passes to the main() and just before the control returns from the main().

Syntax:

#pragma startup function_name

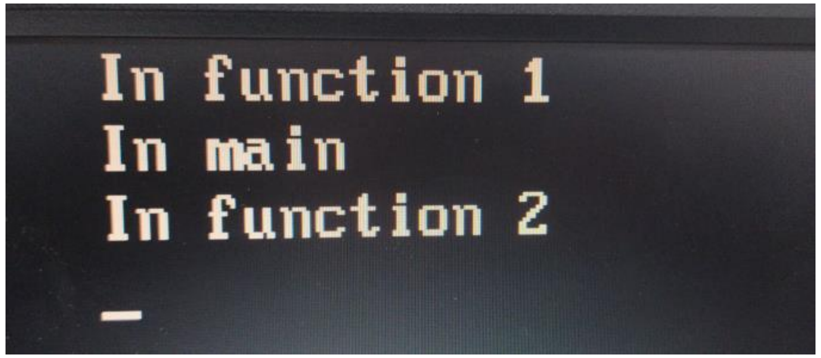
#pragma exit function_name

Example:



```
#include<stdio.h>
#include<conio.h>
void fun1(){
    printf("In function 1 \n");
}
void fun2(){
    printf("In function 2 \n");
    getch();
}
#pragma startup fun1
#pragma exit fun2

void main(){
    printf("In main \n");
}
```

Output:



```
In function 1
In main
In function 2
_
```

2. #pragma warn

This directive is used to hide the warning messages which are displayed during compilation. This may be useful for us when we have a large program and we want to solve all the errors before looking on warnings then by using it we can focus on errors by hiding all warnings. we can again let the warnings be visible by making slight changes in syntax.

Syntax:

To hide the warning: #pragma warn –name

To show the warning: #pragma warn +name

To toggle between hide and show: #pragma warn .name

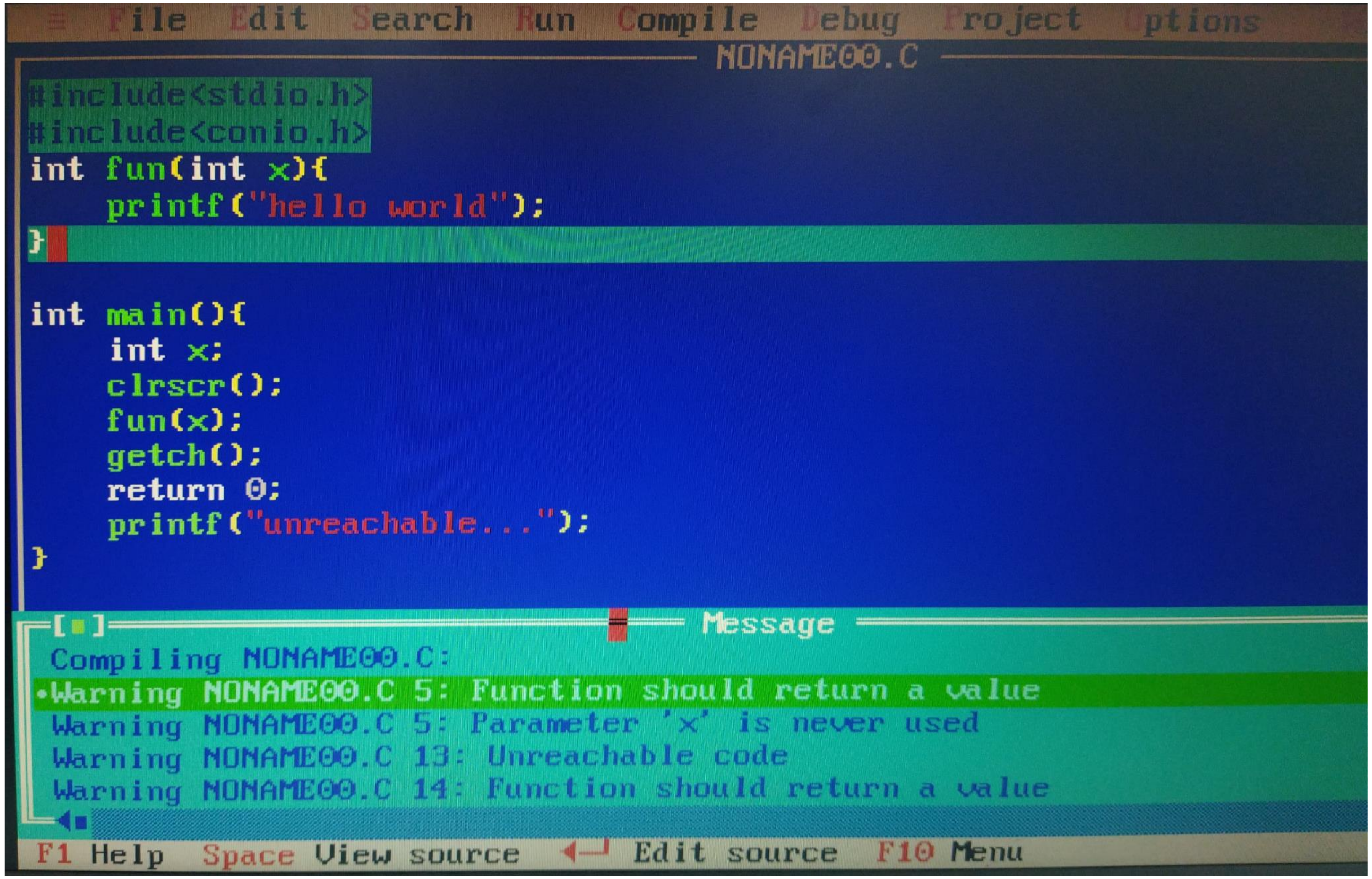
There are several types of warnings shown below:

#pragma warn -rvl: This directive hides those warning which are raised when a function which is supposed to return a value does not return a value.

#pragma warn -par: This directive hides those warning which are raised when a function does not uses the parameters passed to it.

#pragma warn -rch: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the return statement in a function is unreachable.

- **Example:**


File Edit Search Run Compile Debug Project Options
NONAME00.C

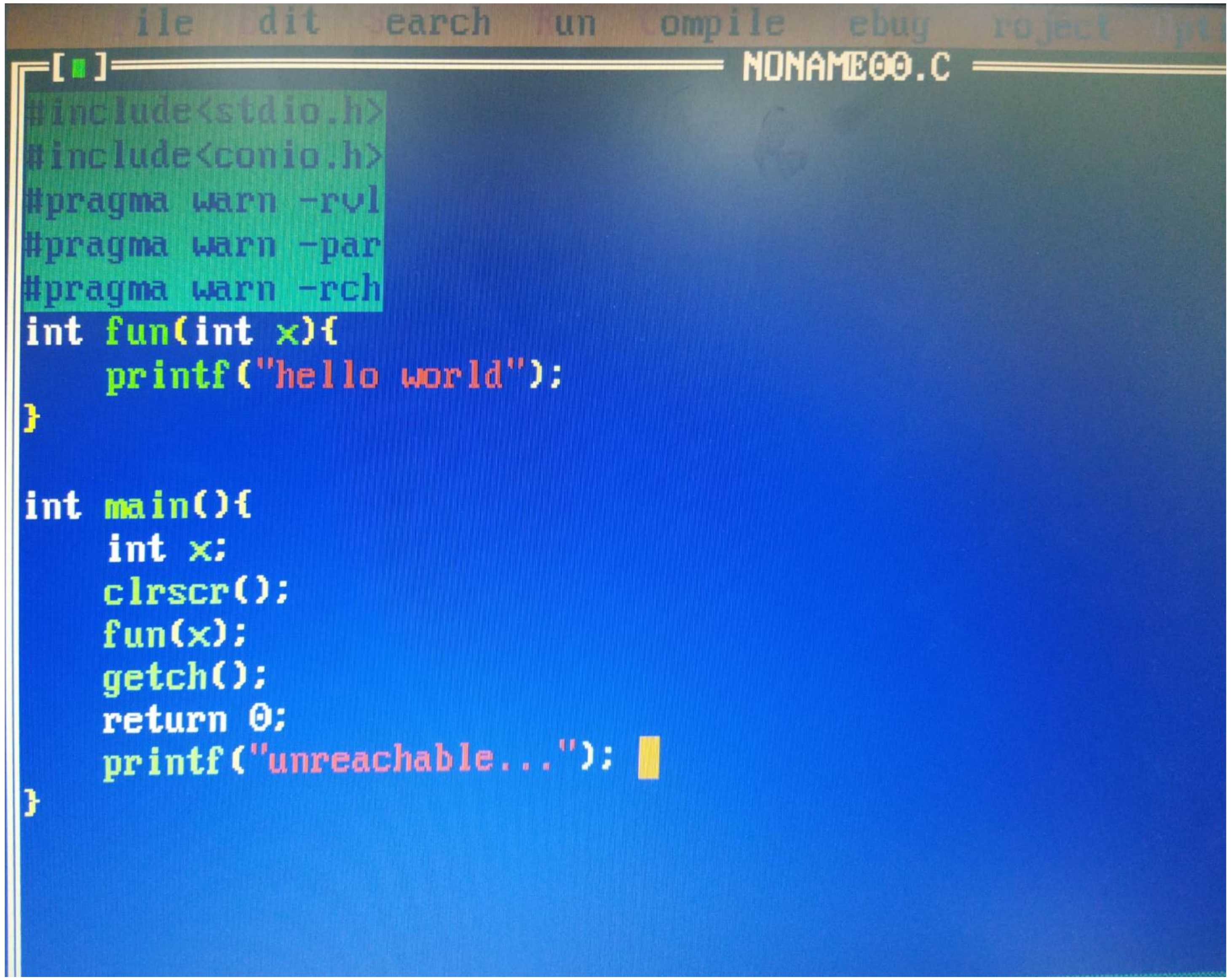
```
#include<stdio.h>
#include<conio.h>
int fun(int x){
    printf("hello world");
}

int main(){
    int x;
    clrscr();
    fun(x);
    getch();
    return 0;
    printf("unreachable...");
}
```

```
Message
Compiling NONAME00.C:
•Warning NONAME00.C 5: Function should return a value
 Warning NONAME00.C 5: Parameter 'x' is never used
 Warning NONAME00.C 13: Unreachable code
 Warning NONAME00.C 14: Function should return a value
```

F1 Help Space View source ← Edit source F10 Menu

Explanation: In this program, warnings are visible. Now those can be hidden by using #pragma warn directive.

Example:



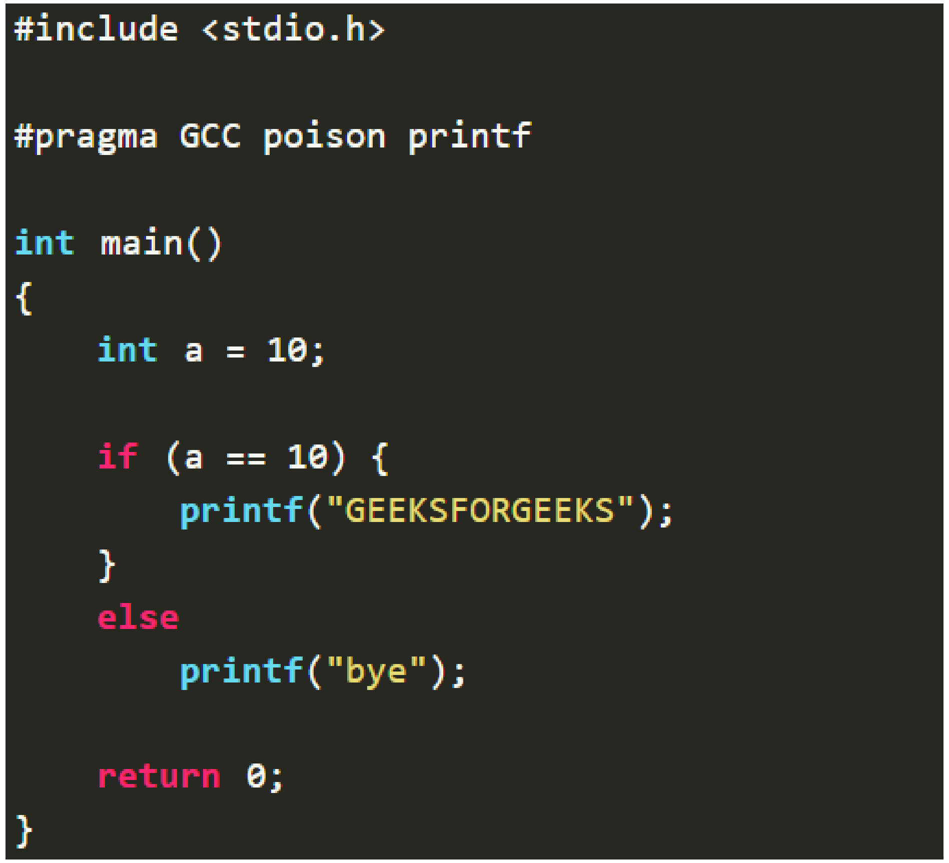
```
#include<stdio.h>
#include<conio.h>
#pragma warn -rvl
#pragma warn -par
#pragma warn -rch
int fun(int x){
    printf("hello world");
}

int main(){
    int x;
    clrscr();
    fun(x);
    getch();
    return 0;
    printf("unreachable...");
}
```

Explanation: In this program, no warning will show as they are hidden by using #pragma warn

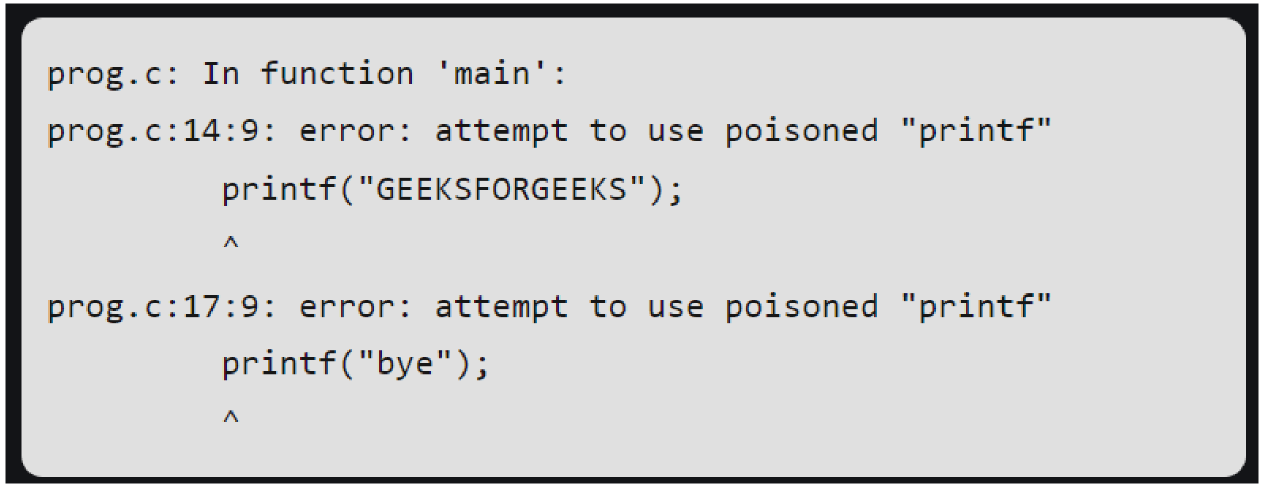
3. #pragma GCC poison

This directive is supported by the GCC compiler and is used to remove an identifier completely from the program. If we want to block an identifier then we can use the #pragma GCC poison directive.

Syntax: #pragma GCC poison identifier



```c
#include <stdio.h>

#pragma GCC poison printf

int main()
{
    int a = 10;

    if (a == 10) {
        printf("GEEKSFORGEEKS");
    }
    else
        printf("bye");

    return 0;
}
```

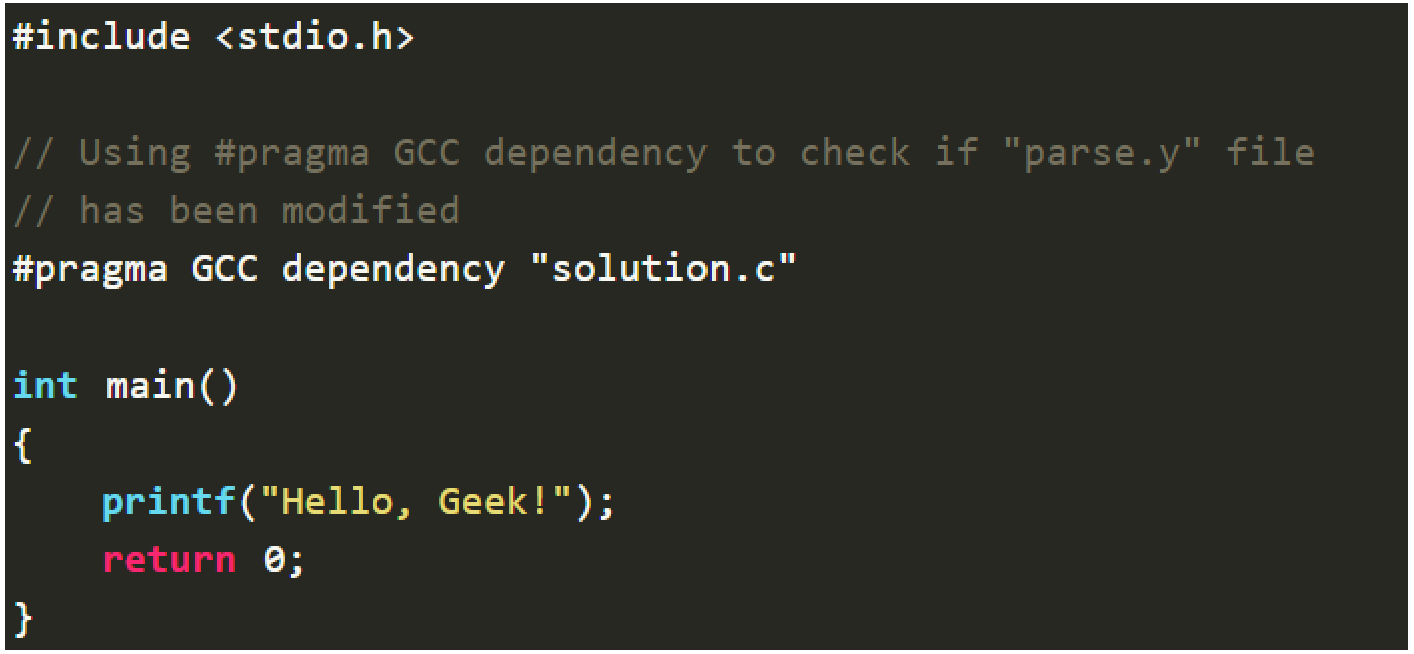


```
prog.c: In function 'main':
prog.c:14:9: error: attempt to use poisoned "printf"
         printf("GEEKSFORGEEKS");
         ^
prog.c:17:9: error: attempt to use poisoned "printf"
         printf("bye");
         ^
```

4. #pragma GCC dependency:

The #pragma GCC dependency 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. This is useful if the current file is derived from the other file, and should be regenerated.

Syntax: #pragma GCC dependency "main.c"



```c
#include <stdio.h>

// Using #pragma GCC dependency to check if "parse.y" file
// has been modified
#pragma GCC dependency "solution.c"

int main()
{
    printf("Hello, Geek!");
    return 0;
}
```

Explanation: At the compile time compiler will check whether the “parse.y” file is more recent than the current source file. If it is, a warning will be issued during compilation.

5. #pragma GCC system_header

The #pragma GCC system_header directive in GCC is used to tell the compiler that the given file should be treated as if it is a system header file. warnings that will be generated by the compiler for code in that file will be suppressed hence it is extremely useful when the programmer doesn't want to see warnings or errors. This pragma takes no arguments. It causes the rest of the code in the current file to be treated as if it came from a system header.

Syntax: #pragma GCC system_header

Note:

The above code is for external library

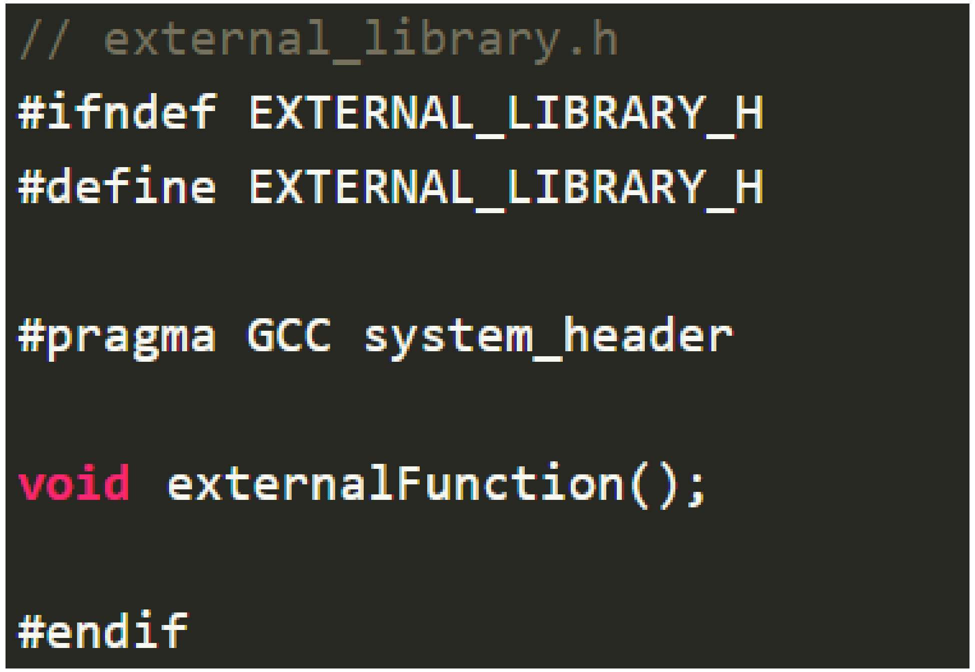
that we want to include in our main code.

So to inform the compiler that

external_library.h is a system header,

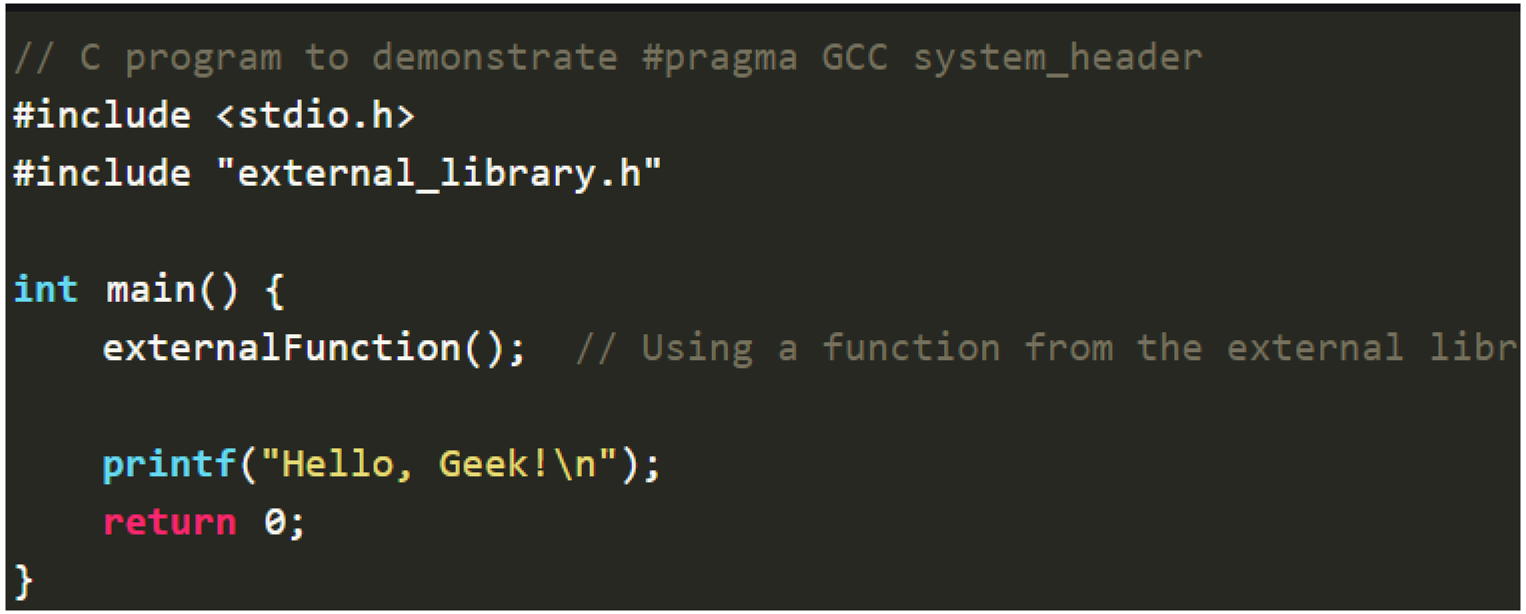
and we don't want to see warnings from it

the #pragma GCC system_header is used.



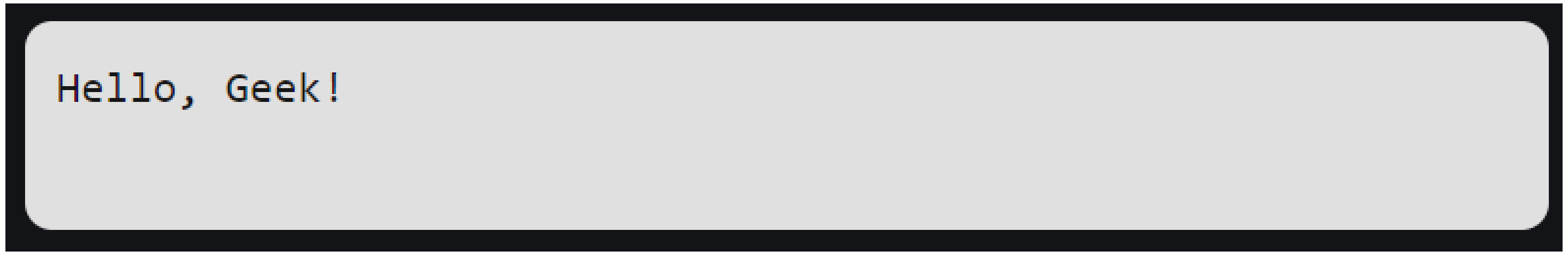
```
// external_library.h
#ifndef EXTERNAL_LIBRARY_H
#define EXTERNAL_LIBRARY_H

#pragma GCC system_header

void externalFunction();

#endif
```

Example:



```c
// C program to demonstrate #pragma GCC system_header
#include <stdio.h>
#include "external_library.h"

int main() {
    externalFunction();  // Using a function from the external libr

    printf("Hello, Geek!\n");
    return 0;
}
```

Output:



```
Hello, Geek!
```

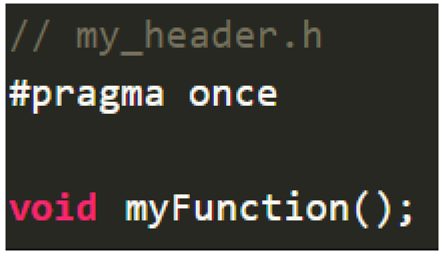
6. #pragma once

The #pragma once directive has a very simple concept. The header file containing this directive is included only once even if the programmer includes it multiple times during a compilation. This directive works similar to the #include guard idiom. Use of #pragma once saves the program from multiple inclusion optimisation.

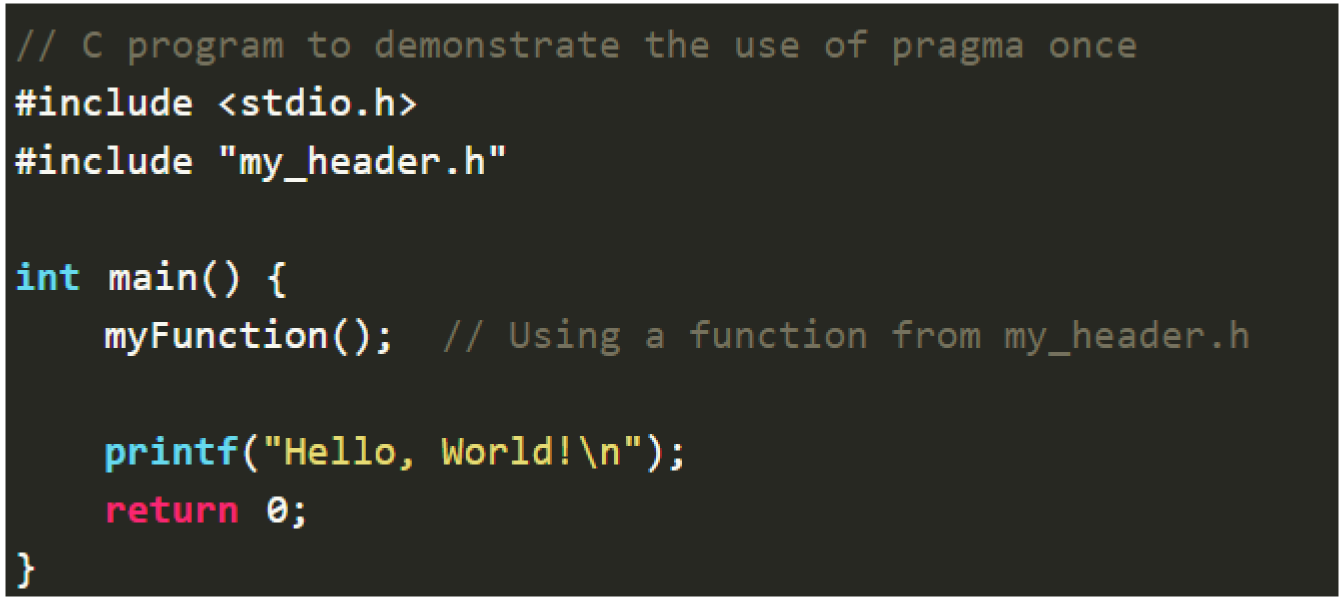
Syntax: #pragma once

Example:

#pragma once is used to avoid multiple inclusions.

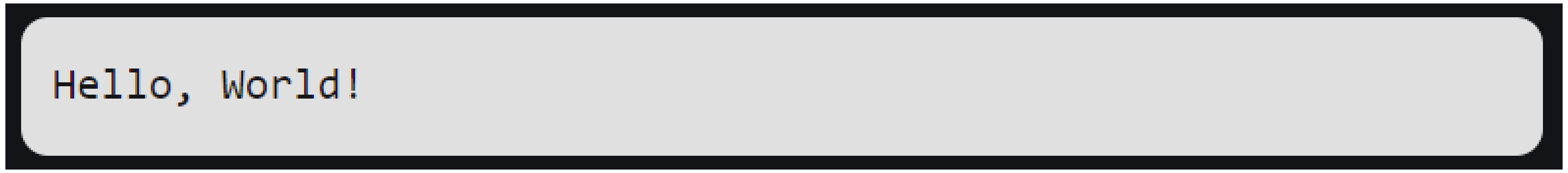


```
// my_header.h
#pragma once

void myFunction();
```



```
// C program to demonstrate the use of pragma once
#include <stdio.h>
#include "my_header.h"

int main() {
    myFunction();  // Using a function from my_header.h

    printf("Hello, World!\n");
    return 0;
}
```

Output:



```
Hello, World!
```