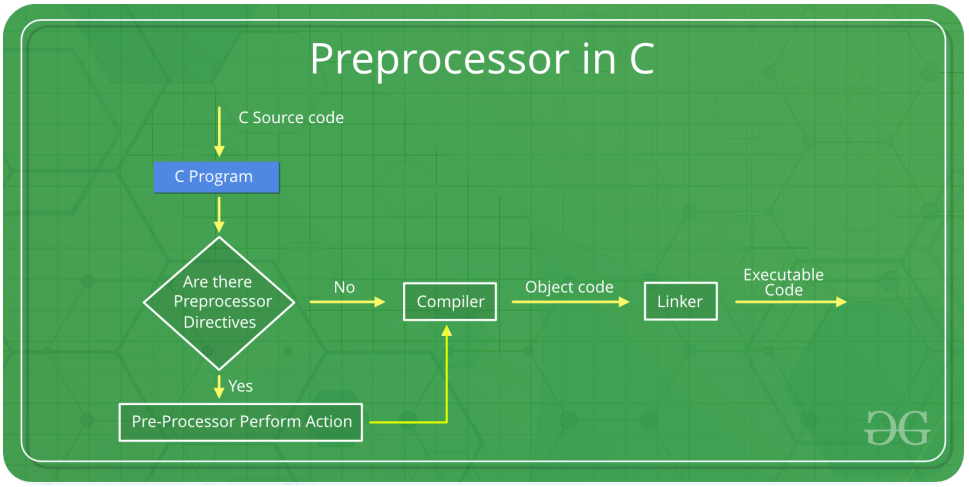
# C/C++ Preprocessors

As the name suggests, Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps before we actually start learning about Preprocessors.




You can see the intermediate steps in the above diagram. The source code written by programmers is first stored in a file, let the name be "**program.c**". This file is then processed by preprocessors and an expanded source code file is generated named "program.i". This expanded file is compiled by the compiler and an object code file is generated named "program.obj". Finally, the linker links this object code file to the object code of the library functions to generate the executable file "program.exe".

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that whatever statement starts with a '#' will go to the preprocessor program to get executed. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc. Remember that the # symbol only provides a path to the preprocessor, and a command such as include is processed by the preprocessor program. For example, #include will include extra code in your program. We can place these preprocessor directives anywhere in our program.

**There are 4 Main Types of Preprocessor Directives:**

1. Macros
2. File Inclusion

3. Conditional Compilation
4. Other directives

Let us now learn about each of these directives in detail.

## 1. Macros

Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro. Let us now understand the macro definition with the help of a program:

```
#include <iostream> // macro definition #define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        std::cout << i << "\n";
    }
    return 0;
}
```

**Output:**

```
0
1
2
3
4
```

In the above program, when the compiler executes the word LIMIT, it replaces it with 5. The word 'LIMIT' in the macro definition is called a macro template and '5' is macro expansion.

**Note**: There is no semi-colon (;) at the end of the macro definition. Macro definitions do not need a semi-colon to end.

**Macros With Arguments**: We can also pass arguments to macros. Macros defined with arguments work similarly to functions. Let us understand this with a program:

```
#include <iostream> // macro with parameter*#define AREA(l, b
int main()
{
```

```
    int l1 = 10, l2 = 5, area;

    area = AREA(l1, l2);

    std::cout << "Area of rectangle is: " << area;

    **return** 0;
}
```

**Output:**

```
Area of rectangle is: 50
```

We can see from the above program that whenever the compiler finds AREA(l, b) in the program, it replaces it with the statement (l*b). Not only this, but the values passed to the macro template AREA(l, b) will also be replaced in the statement (l*b). Therefore AREA(10, 5) will be equal to 10*5.

## 2. File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files that can be included by the user in the program:

**Header files or Standard files**: These files contain definitions of pre-defined functions like **printf(), scanf(),** etc. These files must be included to work with these functions. Different functions are declared in different header files. For example, standard I/O functions are in the 'iostream' file whereas functions that perform string operations are in the 'string' file.

**Syntax**:

```
#include<file_name>
```

where *file_name* is the name of the file to be included. The '<' and '>' brackets tell the compiler to look for the file in the standard directory.

**User-defined files**: When a program becomes very large, it is a good practice to divide it into smaller files and include them whenever needed. These types of files are user-defined files. These files can be included as:

```
#include"filename"
```

## 3. Conditional Compilation

Conditional Compilation directives are a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. This can be done with the help of the two preprocessing commands '**ifdef**' and '**endif**'.

**Syntax**:

```
#ifdef macro_name
    statement1;
    statement2;
    statement3;
    .
    .
    .
    statementN;
#endif
```

If the macro with the name '*macro_name*' is defined, then the block of statements will execute normally, but if it is not defined, the compiler will simply skip this block of statements.

## 4. Other Directives

Apart from the above directives, there are two more directives that are not commonly used. These are:

**#undef Directive**: The #undef directive is used to undefine an existing macro. This directive works as:

```
#undef LIMIT
```

Using this statement will undefine the existing macro LIMIT. After this statement, every "#ifdef LIMIT" statement will evaluate as false.

**#pragma Directive**: This directive is a special purpose directive and is used to turn on or off some features. This type of directives are compiler-specific, i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:

- **#pragma startup** and **#pragma exit**: These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the control returns from main()).

**Note:** Below program will not work with GCC compilers.

```
#include <bits/stdc++.h>
using namespace std;
void func1();
void func2();
#pragma startup func1
#pragma exit func2

void func1()
{
    cout << "Inside func1() \n";
}
void func2()
{
    cout << "Inside func2()\n";
}

int main()
{
    void func1();
    void func2();
    cout << "Inside main()\n";

    return 0;
}
```

**Output:**

```
Inside func1()
Inside main()
Inside func2()
```

The above code will produce the output as given below when run on GCC compilers:

```
Inside main()
```

This happens because GCC does not support #pragma startup or exit. However, you can use the below code for a similar output on GCC compilers.

```
#include <iostream>
using namespace std;

void func1();
void func2();

void __attribute__((constructor)) func1();
void __attribute__((destructor)) func2();

void func1()
{
    printf("Inside func1() \n");
}

void func2()
{
    printf("Inside func2() \n");
}

// Driver code*int main()
{
    printf("Inside main()**\n**");

    return 0;
```

**#pragma warn Directive:** This directive is used to hide the warning message which is displayed during compilation. We can hide the warnings as shown below:

- **#pragma warn -rvl**: This directive hides those warnings which are raised when a function that is supposed to return a value does not return a value.
- **#pragma warn -par**: This directive hides those warnings which are raised when a function does not use the parameters passed to it.
- **#pragma warn -rch**: This directive hides those warnings which are raised when a code is unreachable. For example, any code written after the *return* statement in a function is unreachable.