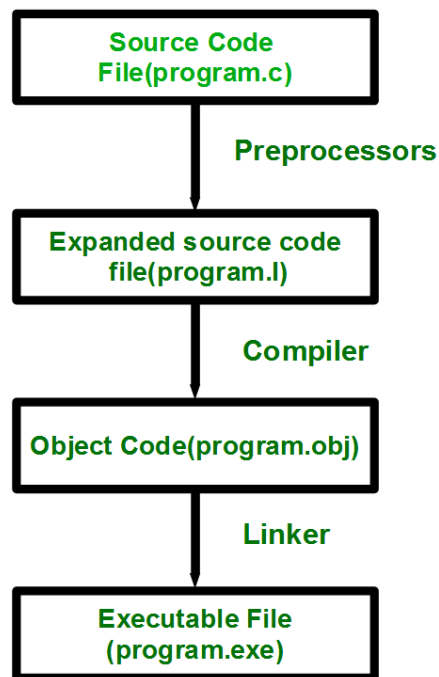


C/C++ Preprocessors

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 stored in the file `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 preprocessors directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. This '#' symbol at the beginning of a statement in a C/C++ program indicates that it is a pre-processor directive. We can place these pre-processor directives anywhere in our program. Examples of some preprocessor directives are: `#include`, `#define`, `#ifndef` etc.

- **Macros:** Macros are piece of code in a program which 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 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;
}
```

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

Note: There is no semi-colon(';') at the end of 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 works similarly as functions. Let us understand this with a program:

```
#include <iostream>

// macro with parameter
#define AREA(l, b) (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
```

- **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 which can be included by the user in the program:

1. **Header File or Standard files:** These files contains definition of pre-defined functions like `printf()`, `scanf()` etc. These files must be included for working with these functions. Different function are declared in different header files. For example standard I/O fununctions are in 'iostream' file whereas functions which perform string operations are in 'string' file.

Syntax:

```
#include< file_name >
```

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

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

```
#include"filename"
```

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

Syntax:

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

If the macro with name as '*macroname*' 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.

- **Other directives:** Apart from the above directives there are two more directives which are not commonly used. These are:

1. **#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 to false.


2. **#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 helps 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.

Look at the below program:

```
 #include <stdio.h>

 void func1();
void func2();

 #pragma startup func1
 #pragma exit func2

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

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

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





    return 0;
}
```

```
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 supports #pragma startup or exit. However you can use the below code for a similar output on GCC compilers.

```
 #include <stdio.h>  
 void func1();  
void func2();  
 void __attribute__((constructor)) func1();  
 void __attribute__((destructor)) func2();  
  
void func1()  
{  
    printf("Inside func1()\n");  
}  
  
void func2()  
{  
    printf("Inside func2()\n");  
}  
  
int main()  
{  
    printf("Inside main()\n");  
  
    return 0;  
}
```