The C Preprocessor

Akash Hegde

Seventh Sense Talent Solutions

Vivekananda Institute of Technology

27 March 2021

Introduction

- 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 substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.
- All preprocessor commands begin with a hash symbol (#).
- It must be the first non-blank character, and for readability, a preprocessor directive should begin in the first column.

Preprocessor Directives

- **#define** – Substitutes a preprocessor macro.
- **#include** - Inserts a particular header from another file.
- **#undef** - Undefines a preprocessor macro.
- **#ifdef** - Returns true if this macro is defined.
- **#ifndef** - Returns true if this macro is not defined.
- **#if** - Tests if a compile time condition is true.
- **#else** - The alternative for #if.
- **#elif** - #else and #if in one statement.
- **#endif** - Ends preprocessor conditional.
- **#error** - Prints error message on stderr.
- **#pragma** - Issues special commands to the compiler, using a standardized method.

Preprocessor Examples

- Example 1:
  #define MAX_ARRAY_LENGTH 20
  This directive tells the preprocessor to replace instances of MAX_ARRAY_LENGTH with 20.
  Use #define for constants to increase readability.
- Example 2:
  #include <stdio.h>
  #include "myheader.h"
  These directives tell the preprocessor to get stdio.h from **System Libraries** and add the text to the current source file.
  The next line tells preprocessor to get **myheader.h** from the local directory and add the content to the current source file.

Preprocessor Examples

- Example 3:
  #undef FILE_SIZE
  #define FILE_SIZE 42
  It tells the preprocessor to undefine existing FILE_SIZE and define it as 42.
- Example 4:
  #ifndef MESSAGE
  #define MESSAGE “You wish!”
  #endif
  It tells the preprocessor to define MESSAGE only if MESSAGE isn't already defined.

Preprocessor Examples

- Example 5:

```
#ifdef DEBUG
      /* debugging statements here */
#endif
```

It tells the preprocessor to process the statements enclosed if DEBUG is defined.
This is useful if you pass the *-DDEBUG* flag to the *gcc* compiler at the time of compilation.
This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

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.
- **__DATE__** - The current date as a character literal in "MMM DD YYYY" format.
- **__TIME__** - The current time as a character literal in "HH:MM:SS" format.
- **__FILE__** - This contains the current filename as a string literal.
- **__LINE__** - This contains the current line number as a decimal constant.
- **__STDC__** - Defined as 1 when the compiler complies with the ANSI standard.

Predefined Macros

- Example:

```
#include <stdio.h>
int main() {
	printf("File :%s\n", __FILE__ );
	printf("Date :%s\n", __DATE__ );
	printf("Time :%s\n", __TIME__ );
	printf("Line :%d\n", __LINE__ );
	printf("ANSI :%d\n", __STDC__ );
	return 0;
}
```

Preprocessor Operators

- **The macro continuation operator -**
- A macro is normally confined to a single line.
- The macro continuation operator (\) is used to continue a macro that is too long for a single line.
- For example –

```
#define message_for(a, b) \
    printf(#a " and " #b ": Hello world!\n")
```

Preprocessor Operators

- **The stringize operator -**
- The stringize or number-sign operator ('#'), when used within a macro definition, converts a macro parameter into a string constant.
- This operator may be used only in a macro having a specified argument or parameter list.
- For example –

```
#include <stdio.h>
#define message_for(a, b) \
     printf(#a " and " #b ": Hello world!\n")
int main(void) {
     message_for(Java, Python);
     return 0;
}
```

Preprocessor Operators

- **The token pasting operator -**
- The token-pasting operator (##) within a macro definition combines two arguments.
- It permits two separate tokens in the macro definition to be joined into a single token.
- For example –

```
#include <stdio.h>
#define tokenpaster(n)
      printf ("token" #n " = %d", token##n)
int main(void) {
      int token34 = 40;
      tokenpaster(34);
      return 0;
}
```

Preprocessor Operators

- **The defined operator -**
- It is used in constant expressions to determine if an identifier is defined using #define.
- If the specified identifier is defined, the value is true (non-zero).
- If the symbol is not defined, the value is false (zero).
- For example –

```
#include <stdio.h>
#if !defined (MESSAGE)
      #define MESSAGE "You wish!“
#endif
int main(void) {
      printf("Here is the message: %s\n", MESSAGE);
      return 0;
}
```

Parameterized Macros

- One of the powerful functions of the preprocessor is the ability to simulate functions using parameterized macros.
- For example, we might have some code to square a number as follows –

```
int square(int x) {
      return x * x;
}
```

- We can rewrite above the code using a macro as follows –

```
#define square(x) ((x) * (x))
```

Parameterized Macros

- Macros with arguments must be defined using the **#define** directive before they can be used.
- The argument list is enclosed in parentheses and must immediately follow the macro name.
- Spaces are not allowed between the macro name and open parenthesis.
- Example:

```
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(void) {
      printf("Max b/w 20 and 10 is %d\n", MAX(10, 20));
      return 0;
}
```

Header Files

- A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files.
- There are two types of header files: the files that the programmer writes and the files that comes with your compiler.
- You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Header Files

- Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.
- A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Header Files

- **Include -** Both the user and the system header files are included using the preprocessing directive **#include**.
- It has the following two forms –
- **#include <file>**
  This form is used for system header files.
  It searches for a file named 'file' in a standard list of system directories.
  You can prepend directories to this list with the -I option while compiling your source code.
- **#include “file”**
  This form is used for header files of your own program.
  It searches for a file named 'file' in the directory containing the current file.
  You can prepend directories to this list with the -I option while compiling your source code.

Header Files

- 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.
- For example, if you have a header file header.h as follows:
  char *test (void);
  …

Header Files

- …and a main program called *program.c* that uses the header file, like this –

```
int x;
#include "header.h“
int main (void) {
      puts (test ());
}
```

- the compiler will see the same token stream as it would if program.c read.

```
int x;
char *test (void);
int main (void) {
      puts (test ());
}
```

Header Files

- **Once-only headers** – If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error.
- The standard way to prevent this is to enclose the entire real contents of the file in a conditional.
- Example:
  #ifndef HEADER_FILE
  #define HEADER_FILE
  the entire header file file
  #endif
- This construct is commonly known as a wrapper **#ifndef**.
- When the header is included again, the conditional will be false, because HEADER_FILE is defined.
- The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Header Files

- **Computed includes** – Sometimes it is necessary to select one of the several different header files to be included into your program.
- For instance, they might specify configuration parameters to be used on different sorts of operating systems.
- You could do this with a series of conditionals as follows –

```
#if SYSTEM_1
      # include "system_1.h“
#elif SYSTEM_2
      # include "system_2.h“
#elif SYSTEM_3
...
#endif
```

Header Files

- But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name.
- This is called a **computed include**.
- Instead of writing a header name as the direct argument of **#include**, you simply put a macro name there –
  #define SYSTEM_H "system_1.h"
  …
  #include SYSTEM_H
- SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the **#include** had been written that way originally.
- SYSTEM_H could be defined by your Makefile with a -D option.

Thank you!