C-Preprocessor

1. Source code files are processed by the preprocessor before being complied. A C-Preprocessor is a separate program which can run independently. However, it is invoked automatically by the C-Compiler before the compilation stage.

2. The preprocessor converts the source code file into another source code file i.e. modify and expand the original source code file. That modified file could be stored in memory before being sent to the compiler or even exists as a real file in the file system.

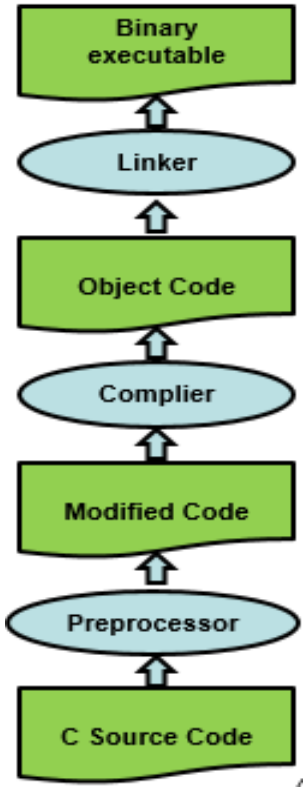
3. Preprocessor commands start with "#“. for example:

- ✓ #define: mainly used to define constants e.g. → #define MAX_ARRAY_SIZE 1000
- ✓ #include: usually used to include header files e.g. → #include <stdio.h>
  this will add the contents of <stdio.h> into the source code file at the location of the #include statement before it gets compiled. This will allow using functions such as printf and scanf, whose declarations are located in the file stdio.h. (include allows re-use of previously written code in C programs).

C-Preprocessor

- The **C Preprocessor** is a separate step in the compilation process and not considered a part of the compiler. It is no more than a text substitution tool which instructs the compiler to do some required pre-processing instructions before the actual compilation process starts.
- All preprocessor commands begin with a “#”.
- May appear in any place in the code.
- The C preprocessor provides the below activities

1. File Inclusion.
   - Allow a program to include header files.
   - Name of the header file should end with “.h”
   - Use the “#include” directive.
   - #include “abc.h” → including user-define header files.
   - #include <abc.h> →including system header files.
2. Conditional compilation.
   - Allow sharing code on deferent platforms.
   - Allow sharing code only in a specific situation.
3. Macro expansion.
   - Provide parameterized text substitution.
   - No type checking.
   - Code may run faster.
4. Constant definition.



Commonly used Directives

Directive	Description
#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.

Examples

1) include

```
#include <stdio.h>
#include "myFile.h"
```

- ✓ gets stdio.h from **System Libraries** and add the text to the current source.
- ✓ gets **myheader.h** from the local directory and add the content to the current source

2) define

```
#define ARRAY_SIZE 100
```

- ✓ replace instances of ARRAY_SIZE in the program with 100.
- ✓ commonly used for declaring constants in c to increase readability.

3) undef

```
#undef ARRAY_SIZE
#define ARRAY_SIZE 1000
```

- ✓ un-define the existing ARRAY_SIZE and re-define it with size = 1000

4) ifndef

```
#ifndef PI
#define PI 3.141593
#endif
```

- ✓ define PI only if PI is not defined

```
#ifndef NULL
#define NULL (void *)0
#endif
```

5) ifdef

```
#ifdef PI
#undef PI
#endif
```

- ✓ Un-define PI only if PI exists

The Defined() Operator

The preprocessor defined operator 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).

```
#include <stdio.h>

#if !defined (MESSAGE)
  #define MESSAGE "This is a C program"
#endif

int main(void) {
  printf("%s\n", MESSAGE);
  return 0;
}
```

Output:
This is a C program

How to avoid duplicate file include using defined directive

f1.h

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

f2.h

```
#include "f1.h"
int mult(int a, int b)
{
  return a*b;
}
```

main.c

```
#include "f2.h"
#include "f1.h"
int main()  {
   return 0;
}
```

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-
line>"
# 1 "main.c"
# 1 "f2.h" 1
# 1 "f1.h" 1
int sum(int a, int b) {
  return a+b;
}
# 2 "f2.h" 2
int mult(int a, intb) {
  return a*b;
}
# 2 "main.c" 2
# 1 "f1.h" 1
int sum(int a, int b) {
  return a+b;
}
# 3 "main.c" 2
int main() {
  return 0;
}
```

f1.h

```
#if !defined(__F1__HEADER__)
#define __F1__HEADER__
int sum(int a, int b) {
  return a+b;
}
#endif
```

f2.h

```
#include "f1.h"
int mult(int a, int b)
{
  return a*b;
}
```

main.c

```
#include "f2.h"
#include "f1.h"
int main()  {
   return 0;
}
```

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-
line>"
# 1 "main.c"
# 1 "f2.h" 1
# 1 "f1.h" 1
int sum(int a, int b) {
  return a+b;
}
# 2 "f2.h" 2
int mult(int a, intb) {
  return a*b;
}
# 2 "main.c" 2

int main() {
  return 0;
}
```

Predefined Macros

Macro	Description
__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.

test.c

```c
#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;
}
```

Output:

File :test.c
Date :Jan 21 2017
Time :08:31:57
Line :7
ANSI :1

Example 1

Macros

test.c

```
#include <stdio.h>
#define addOne(a) a+1
int main() {

  int i = 5*addOne(2);

  printf("%d",i);

  return 0;
}
```

version 1

```
int main() {
  int i = 5*2 +1;
  printf("%d",i);
  return 0;
}
```

Output: 11

After manual invoke of GNU preprocessor

gcc –E test.c

test.c

```
#include <stdio.h>
#define addOne(a) ((a)+1)
int main() {

  int i = 5*addOne(2);

  printf("%d",i);

  return 0;
}
```

version 2

```
int main() {
int i = 5*((2)+1);
printf("%d",i);
  return 0;
}
```

Output: 15

Example 2

```c
#include <stdio.h>
#define max(a,b) a>b? a:b
int main() {
  int a=1,b=2;

  int x=max(a, b);
  printf("X:%d\n", x);

  int y=max(a+1,b+1);
  printf("Y:%d\n", y);

  int z=max(a+5,b+1);
  printf("Z:%d\n", z);
 return 0;
}
```

```c
int main() {
  int a=1,b=2;

  int x=a>b? a:b;
  printf("X:%d\n", x);

  int y=a+1>b+1? a+1:b+1;
  printf("Y:%d\n", y);

  int z=a+5>b+1? a+5:b+1;
  printf("Z:%d\n", z);
 return 0;
}
```

Output:

X:2→ 1>2?1:2 → 2

Y:3→ 1+1>2+1?1+1:2+1 → 2+1 → 3

Z:6→ 1+5>2+1?1+5:2+1 → 5+1 → 6

Example 3

```c
#include <stdio.h>
#define MULT(a, b) a*b
int main(void) {
  int x = MULT(3,4);
  printf("X:%d\n", x);
  int y = MULT(3+2,4+2);
  printf("Y:%d\n", y);
  return 0;
}
```

version 1

Output:

X:12

Y:13 → 3+2*4+2 →13;

```c
#include <stdio.h>
#define MULT(a, b) (a)*(b)
int main(void) {
  int x = MULT(3,4);
  printf("X:%d\n", x);
  int y = MULT(3+2,4+2);
  printf("Y:%d\n", y);
  return 0;
}
```

version 2

Output:

X:12

Y:30 → (3+2)*(4+2) → 30

Macro for sum array elements of ant type

```
#include <stdio.h>

#define SUM_ARRAY( ARRAY, NUMBER_OF_ELEMENTS ) \
   { \
      double total = 0; \
      int i; \
      for (i=0 ; i < NUMBER_OF_ELEMENTS ; i++ ) \
      { \
         total += ARRAY[ i ]; \
      } \
      printf( "%f\n", total ); \
   }

int main( void ) {
   char a1[ 5 ] = { 1, 2, 3, 4, 5 };
   int a2[ 5 ] = { 1, 2, 3, 4, 5 };
   short a3[ 5 ] = { 1, 2, 3, 4, 5 };
   SUM_ARRAY( a1, 5 );
   SUM_ARRAY( a2, 5 );
   SUM_ARRAY( a3, 5 );
   return 0;
}
```

END