CPNA Lecture 19 - The Preprocessor

Mridul Sankar Barik

Jadavpur University

2023

Introduction

- The preprocessor is a software that edits C source file prior to compilation
- C program ⇒ [Preprocessor] ⇒ Modified C program ⇒ [Compiler] ⇒ Object Code
- Preprocessor directives: commands that begin with a # character
  - `#define` directive defines a macro - a name that represents a constant or frequently used expressions
  - `#include` directive tells preprocessor to include content of a file as part of the file being compiled
  - `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif` directives for conditional compilation
- Additional tasks performed:
  - Replace each comment with a single space character
  - (Sometimes) Remove unnecessary white space characters
- To generate preprocessed code: `$gcc -E test.c -o test.E`

Preprocessor Directives - Rules

- Directives always begin with a # symbol
- Any number of spaces and horizontal tab characters may separate the tokens in a directive
  `# define N 100`
- Directives always end at the first new line character, unless explicitly continued

```
#define VOLUME (LENGTH * \
                WIDTH * \
                HEIGHT)
```

- Comments may appear on the same line as directives
  `#define N 10 /* Some Comment */`

Macro Definition I

- Simple Macros:
  - Format: `#define identifier replacement-list`
  - Replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators and punctuation

```
#define STR_LEN 80
#define PI 3.14159
#define CR '\r'
#define EOS '\0'
#define ERR_MSG "There was an error"
```

  - Advantages
    - It makes programs easier to read
    - It makes programs easier to modify
    - Making minor changes to C's syntax

```
#define BEGIN {
#define END }
```

    - Renaming types

Macro Definition II

```
#define BOOL char
#define BYTE unsigned char
#define SIZE int
```

- Parameterized Macros:

```
#define identifier(X1, X2, ..., Xn) replacement-list
```

  - Important: There must be no space between macro name and the left parenthesis
  - Example:

```
#define IS_EVEN(n) ((n)%2 == 0)

if(IS_EVEN(i)) i++;
```

  - Use of parameterized macro instead of a function
    Advantages:
    - The program may be slightly faster
    - Macros are generic - parameters have no type

    Disadvantages:
    - Compiled code will often be larger

Macro Definition III

- Arguments are not type checked
- Not possible to have a pointer to a macro
- A macro may evaluate its arguments more than once

Example I

```
#include<stdio.h>

#define MYMULT(x, y) ((x)*(y))
#define MMULT(x, y) x*y

int main(void){
    int i=2, j=3;

    printf("%d\n", MYMULT(i+1, j+1));
    printf("%d\n", MMULT(i+1, j+1));
    return(0);
}
/*Preprocessed Code*/

printf("%d\n", ((i+1)*(j+1)));
printf("%d\n", i+1*j+1);
```

Example II

```
/*Other examples*/

#define Average(x,y) (((x)+(y))/2.0)
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
#define PI 3.1415
#define circleArea(r) (PI*r*r)
```

- One-pass macro processor
  - Alternates between macro definition and macro expansion
  - Definition of a macro must appear before its invocation
- Two-pass macro processor
  - All macro definitions are processed during the first pass
  - All macro invocation statements are expanded during the second pass
  - Does not allow the body of one macro instruction to contain definitions of other macros

Variable Length Argument List

- C99 added **variadic macros** that may have a variable number of arguments
- To define a variadic macro, define a macro with arguments where the last argument is three periods ...
- The macro `__VA_ARGS__` expands to whatever arguments matched this ellipsis in the macro call

```
#define Warning(...) fprintf(stdout, __VA_ARGS__)

Warning("Warning: %s\n", message);
```

Conditional Compilation I

- To instruct preprocessor whether to include certain chuck of code or not
- Uses
  - use different code depending on the machine, operating system
  - to exclude certain code from the program but to keep it as reference for future purpose
- `#ifdef` directive
- Example:

```
#define DEBUG 1

#ifdef DEBUG
printf("Value of i: %d\n", i);
printf("Value of j: %d\n", j);
#endif
```

- `#if #else #elif` directives

Conditional Compilation II

- `defined` Operator: when applied to an identifier, it produces a value 1 if the identifier is a defined macro

```
#ifdef WIN32
...
#elif defined(MAC)
...
#elif defined(LINUX)
...
#endif
```

- `#ifndef` tests whether an identifier is not defined as a macro

Conditional Compilation III

- Another example:

```
int main(void){
    #ifdef TEST
        printf("Test mode\n");
    #endif
    printf("Running...\n");
}
```

  - `-D` command line option of `gcc` is used to define a preprocessor macro from the command line
  - `gcc -DTEST test.c -o test` produces output

    ```
    Test mode
    Running...
    ```

  - `gcc test.c -o test` produces output

    ```
    Running...
    ```

The ## Operator I

- Pastes two tokens (identifiers, literals etc.) together to form a single token
- Example:

```
#define MK_ID(n) i##n

int MK_ID(1), MK_ID(2), MK_ID(3);

/*After proprocessing*/

int i1, i2, i3;
```

The ## Operator II

- Another Example:

```
#define GENERIC_MAX(type) \
type type##_max(type x, type y) \
{
    return (x)>(y)?(x):(y);
}

GENERIC_MAX(float)

/*After proprocessing*/

float float_max(float x, float y)
{
    return (x)>(y)?(x):(y);
}
```

Pre-Define Macros

Name	Description
__LINE__	Line number of file being compiled
__FILE__	Name of file being compiled
__DATE__	Date of compilation
__TIME__	Time of compilation
__STDC__	1, if compiler conforms to C standard