

ECE326

PROGRAMMING LANGUAGES

Lecture 20 : C Preprocessor Macro

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Metaprogramming

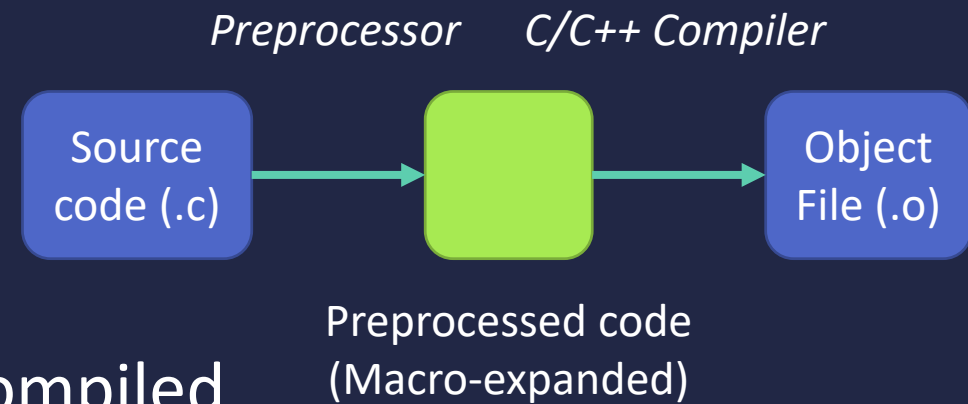
- Writing code that will generate more code
- Generic programming
 - Writing code with minimum assumption about data types
- Reflective programming
 - Access and/or modify program structure and/or behaviour
 - Has knowledge of compiler/interpreter's internals
- Different approaches
- Can have overlaps
 - E.g. C++ template programming encompasses all three

Metaprogramming

- Macro systems
 - Maps certain input sequence into replacement output
 - E.g. text-based replacement
 - `"int a = 5;".replace("int", "long long")`
- Generative programming
 - Purpose of a program is to generate code for another program
 - May be same or different target language
 - Haskell compiler can generate C code from Haskell source code
- Template programming
 - Parameterized code which can be instantiated upon use

C Preprocessor Macro

- Rudimentary support for metaprogramming in C/C++
- Provides text substitution of tokens
 - Token
 - A lexical unit, comprised of a type and value
 - E.g. `int a;`
 - `int` is a “keyword” token, of value “int”
 - `a` is an “identifier” token, of value “a”
 - `;` is a “separator” token, of value “;”
- Preprocessor
 - Done before C source code is compiled



C Preprocessor Macro

- Macro
 - A fragment of code with a name
- Macro expansion
 - Name replaced by content of macro whenever name is used
- Preprocessor
 - Scans the source code in multiple passes until no more replacement can be made
 - Has no knowledge of the C language
 - DANGER – can even use C keywords for macro names

Macro Constant

- Also known as object-like macro
- Typically used to give name to a special literal

```
#define BUFFER_SIZE 1024
```

```
foo = (char *) malloc (BUFFER_SIZE);
```

```
// becomes this
```

```
foo = (char *) malloc (1024);
```

- Macros can be used after it is defined, but not before

```
foo = X;
```

```
#define X 4
```

```
bar = X;
```



```
foo = X;
```

```
bar = 4;
```

Macro Function

- A macro that takes zero or more parameters
- Looks like a normal function using parentheses

```
#define hello() printf("hello world")  
// macro with two parameters  
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

```
// function macro can be nested  
min(min(a, b), c)
```

→

```
min(((a) < (b) ? (a) : (b)), c)
```

→

```
((((a) < (b) ? (a) : (b))) < (c) ?  
  (((a) < (b) ? (a) : (b))) : (c))
```

Macro Function

- Avoid expressions with side effects when using macro

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

```
min(a++, b)
```

→

```
((a++) < (b) ? (a++) : (b))
```

```
// a++ got called twice when macro is used
```

```
// not the same behaviour if min() is a C function
```

```
// GNU C only solution (not part of official C standard)
```

```
#define min(X, Y) \
({ typeof(X) x_ = (X); \
  typeof(Y) y_ = (Y); \
  (x_ < y_) ? x_ : y_; })
```

{ ... } acts like an expression.
Its value is the value of the
last statement. Similar to Rust

Macro Function

- Multiline macro requires use of continuation \
- Emulating void functions

```
#define print_array(array) do { \
    unsigned i; \
    for (i = 0; i < sizeof(array)/sizeof(*(array)); i++) \
        printf("%ld ", (long)*((array)+i)); \
    printf("\n"); \
} while(0)
```

```
short a[] = { 2 , 3, 5, 46, 345, 1, -3 };
print_array(a);
```

```
2 3 5 46 345 1 -3
```

do { ... } while(0) is necessary to allow natural use of semicolon at end of function. Just { ... } will cause syntax error!

Macro Function

- `do { ... } while(0)`

```
#define bad_compound( ) { \
    printf( "hello\n" ); \
    printf( "world\n" ); }
```

```
if (x > 0)
    bad_compound( );
else
    printf( "x too small" );
```

→

```
if (x > 0)
    { printf( "hello\n" ); printf( "world\n" ); } ;
else
    printf( "x too small" );
```

Stray semicolon

Macro Function

- Wrap all arguments that can be an expression
 - To avoid problems with operator precedence

```
// round up an integer division: divroundup(11, 5) = 3  
#define divroundup(x, y) (x + y - 1) / y
```

```
a = divroundup(b & c, sizeof (int));
```

→

```
a = (b & c + sizeof (int) - 1) / sizeof (int);  
/* C's operator precedence works like this */  
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

```
// better version (also wraps the entire expression)  
#define divroundup(x, y) (((x) + (y) - 1) / (y))
```

Stringification

- Macro functions can turn arguments into a string
 - Use # operator in front of the expression

```
#define WARN_IF(EXP) do { \  
    if (EXP) fprintf (stderr, "Warning: " #EXP "\n"); \  
} while (0)
```

```
WARN_IF(x == 0);
```

→

```
do { if (x == 0)  
    /* C automatically joins string literals */  
    fprintf (stderr, "Warning: " "x == 0" "\n");  
} while (0);
```

Stringification

- To stringify the value of a macro, use a helper

```
#define stringify_value(s) stringify(s)
#define stringify(s) #s
#define FOO 4
```

stringify_value(FOO)

→

stringify_value(4)

→

stringify(4)

→

"4"

stringify(FOO)

→

"FOO"

- Macro arguments are expanded before substitution, UNLESS they are stringified or “pasted”

Concatenation

- Pasting macro argument with another token
 - Use `##` operator between macro and another token

```
#define COMMAND(NAME) { #NAME, NAME ## _command }
```

```
struct command commands[] = {  
    COMMAND (quit),  
    COMMAND (help),  
    ...  
};
```

→

```
struct command commands[] = {  
    { "quit", quit_command },  
    { "help", help_command },  
    ...  
};
```

```
struct command {  
    const char *name;  
    void (*function)();  
};
```

Variadic Macro

- Macro function that takes any number of arguments
 - When you put `##` in front of `vargs`, it deletes a comma in front of it if `vargs` is empty

```
#define eprintf(format, vargs...) \
    fprintf(stderr, format, ##vargs)
```

```
eprintf("success!\n")
```

→

```
fprintf(stderr, "success!\n");
```

```
eprintf("%s:%d: ", input_file, lineno)
```

→

```
fprintf(stderr, "%s:%d: ", input_file, lineno)
```

X Macro

- Technique for maintaining list of tokens

```
#define ACTIONS \
    X(STAND) \
    X(HIT) \
    X(SURRENDER) \
    X(DOUBLE) \
    X(SPLIT)
```

```
#define X(e) e,
enum Action {
    ACTIONS
};
#undef X
```

```
#define X(e) #e,
const char * action_str[] = { ACTIONS };
#undef X
```

#undef
deletes a
macro.

```
printf("%s %s\n", action_str[HIT], action_str[STAND]);
```

HIT STAND

Include Directive

- Adds content of file to current file
 - E.g. action.xmc

```
X(STAND)
X(HIT)
X(SURRENDER)
X(DOUBLE)
X(SPLIT)
```

```
void action_str2(Action e)
#define X(name) if (e == name) \
    return #name ; else
#include "action.xmc"
#undef X
    {} // the last else uses this
    return "ERROR";
}
```

```
void action_str2(Action e)
{
    if (e == STAND)
        return "STAND";
    else if (e == HIT)
        return "HIT";
    ...
    else if (e == SPLIT)
        return "SPLIT";
    else
        {}
    return "ERROR";
}
```

Optional Compilation

- Enable or disable parts of the code
 - Not even compiled at all, won't make it to final executable

```
int take_action(Hand hand, Action a) {  
    if (a == SURRENDER) {  
#ifdef ALLOW_SURRENDER  
        hand.profit = hand.bet / 2.0;  
        hand.state = COMPLETE;  
        return ERR_OK;           // action accepted  
#else  
        return ERR_INVALID;      // action rejected  
#endif  
    }  
    ...  
    return ERR_INVALID;  
}
```

Optional Compilation

- Used in header to avoid being included more than once

```
/* if SHOE_H is not defined */  
#ifndef SHOE_H  
#define SHOE_H  
  
/* declaration of functions and definition of classes */  
  
#endif  
  
#include "shoe.h" // OK - SHOE_H not defined  
#include "shoe.h" // nothing included this time
```

Conclusion

- C macros provide some metaprogramming capability
 - Uses token based substitution
 - Invoked by compiler as first part of translation
 - Inherently unsafe, requires care
 - Reasonably powerful, when coupled with existing C constructs
- C preprocessor
 - Helps manage code into files
 - Allows for optional compilation
 - Can be abused – code will become very difficult to read
 - If executable size not a concern, should use inheritance instead