ECE326

PROGRAMMING LANGUAGES

Lecture 22 : C Preprocessor Macro

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

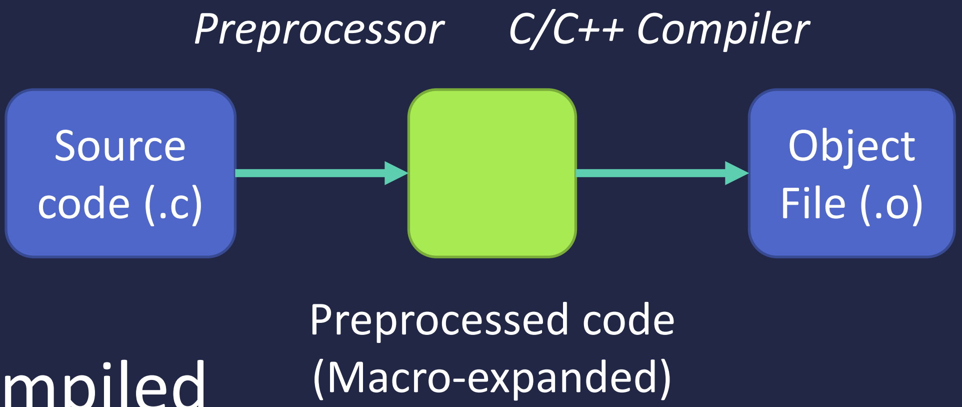
Metaprogramming

- Template programming
  - Parameterized templates are instantiated upon use
  - Enables generic programming and compile-time computation
- Generative programming
  - Purpose of a program is to generate code for another program
    - May be same or different target language
- Macro systems
  - Maps certain input sequence into replacement output
  - E.g. text-based replacement

```
"int a = 5;".replace("int", "long long") # long long a = 5;
```

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

- **#define** MACRO_NAME *macro_content*
- 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

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

// this will not expand the macro function
hello

// this will expand the macro function
hello()
      →
printf("hello world")
```

Macro Function

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

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

```
// macro function 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_macro(X, Y) ((X) < (Y) ? (X) : (Y))

// a++ happens twice when macro is used
// not the same behaviour if min() were a C function
min_macro(a++, b)
       →
       ((a++) < (b) ? (a++) : (b))

a = 5;
r = min_macro(a++, 6);    // r = 6, a = 7

a = 5;
r = min_function(a++, 6); // r = 5, a = 6
```

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);
```

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

Macro Function

- Rationale for 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

- 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
```

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 macro parameter

```
#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 concatenated

Concatenation

- Text-based join of macro argument with another token
  - Use ## operator between parameter and another token

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

Command commands[] = {
    COMMAND (quit),
    COMMAND (help),
    …
};
    →
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
  - If ## is placed in front of vargs and *vargs is empty*, the preprocessor will delete a comma in front of vargs

```
#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)
```

If you don't give … a name, the default name is __VA_ARGS__

C Macro Trick

- Count number of arguments and pass to first argument

```
#define PP_NARG(...) PP_NARG2(__VA_ARGS__, PP_RSEQ_N())
#define PP_NARG2(...) PP_ARG_N(__VA_ARGS__)
/* PP_ARG_N() returns the 10th argument! */
#define PP_ARG_N( _1, _2, _3, _4, _5, _6, _7, _8, _9, N, ...) N
/* PP_RSEQ_N() counts from 9 down to 0 */
#define PP_RSEQ_N() 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

// variadic function from lecture 21
int find_max(int nargs, ...);
#define max(args...) find_max(PP_NARG(args), ##args)

max(702, 422, 631, 834, 892, 104, 772)
```

C Macro Trick

```
#define PP_NARG(...) PP_NARG2(__VA_ARGS__, PP_RSEQ_N())
#define PP_NARG2(...) PP_ARG_N(__VA_ARGS__)
/* PP_ARG_N() returns the 10th argument! */
#define PP_ARG_N(_1, _2, _3, _4, _5, _6, _7, _8, _9, N, ...) N
/* PP_RSEQ_N() counts from 9 down to 0 */
#define PP_RSEQ_N() 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

#define max(args...) find_max(PP_NARG(args), ##args)

max(70, 42, 63, 83, 89, 10, 52)
→ find_max(PP_NARG(…), 70, 42, 63, 83, 89, 10, 52)
→ find_max(PP_NARG2(70, 42, …, 52, PP_RSEQ_N()), 70, 42, …)
                 /* _1, _2, …, _7, _8,_9, N, … */
→ find_max(PP_ARG_N(70, 42, …, 52,  9, 8, 7, 6 , …, 1, 0), 70, 42, …)
→ find_max(7, 70, 42, 63, 83, 89, 10, 52)
```

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, 0.)
X(HIT, 1.)
X(SURRENDER, -0.5)
X(DOUBLE, 2.)
X(SPLIT, 2.)
```

```
float action_value(Action e)
#define X(N, V) if (e == N) \
    return V ; else
#include "action.xmc"
#undef X
    {} // the last else uses this
    return -1;
}
```

```
float action_value(Action e)
{
    if (e == STAND)
        return 0.;
    else if (e == HIT)
        return 1.;
    …
    else if (e == SPLIT)
        return 2.;
    else
        {}
    return -1.;
}
```

For Each

- C programmers use macros to emulate foreach loop

```
struct point { int x, y; };

#define FOREACH(ptr, i, array, size) \
    (i) = 0; \
    for ((ptr) = &array[i]; (i) < (size); (ptr) = &array[++(i)])

unsigned i;
struct point * p;
struct point arr[10] = { … };

FOREACH(p, i, arr, 10) {
    cout << "(" << p->x << ", " << p->y << ")" << endl;
}
```

Predefined Macros

- __FILE__
  - The current input file name (where macro is used)
- __LINE__
  - The current line number (where macro is used)
- Can be used to generate descriptive error messages

```
#define error(fmt, args...) \
    fprintf(stderr, "%s:%d – " fmt, __FILE__, __LINE__, \
    ##args)

error("hello %s", "world");

macro.c:24 – hello world
```

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

```
#ifndef SHOE_H  // if SHOE_H is not defined
#define SHOE_H

/* declaration of functions and definition of classes */

#endif
```

```
// in main.cpp
#include "shoe.h"   // OK – SHOE_H not defined
#include "shoe.h"   // nothing included this time
```

- Some compilers support `#pragma once`
  - Same effect, shorter to write, but requires compiler support

Self-Referential Macros

- Not possible
- Prevents infinite recursion during macro expansion

```
#define foo (4 + foo)
foo
    →
(4 + foo)    // expansion stops here
```

- This includes indirect self reference

```
#define x (4 + y)           x     →     (4 + y)
#define y (2 * x)                 →     (4 + (2 * x))
                            y     →     (2 * x)
                                  →     (2 * (4 + y))
```