ECE326

PROGRAMMING LANGUAGES

Lecture 21 : C/C++ Macro Programming

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

C Preprocessor Macro

- Provides text substitution of tokens
  - Name replaced by content of macro whenever name is used
  - Faster than inline functions
    - Basically same as copy pasting code
  - Requires care as it does not understand host language
- Preprocessor
  - Done before C source code is compiled
  - Scans the source code in multiple passes until no more replacement can be made

X Macro

- Technique for maintaining list of tokens

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

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

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

HIT STAND
```

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

#undef deletes a macro.

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”;
}
```

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
```

Predefined Macros

- __FUNCTION__
  - Name of function the macro is in
  - Helpful for debugging
- __DATE__
  - A string that represents date of compilation
- __TIME__
  - A string that represents time of compilation
- Use these for serious projects (e.g. system library)

For Each

- C programmers use macros to emulate foreach loop

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

#define FOREACH(ptr, idx, array, size) \
    for ((i) = 0, (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;
}
```

- Note that ptr can temporarily have OOB address

Define Function

- Can be used to define functions with same arguments

```
#define DEFINE_COMMAND(name) \
    void name ## _command(int nargs, const char * args[])

DEFINE_COMMAND(quit) {                          // quit_command
    // exit program with exit code
    exit(atoi(args[0]));
}

DEFINE_COMMAND(get) {                           // get_command
    FILE * f = fopen(args[0], “rt”);
    int c;
    while((c = fgetc(f)) != EOF) fputc(c, stdout);
    fclose(f);
}
```

Comma in Arguments

- Only comma inside parentheses are preserved
- Brackets or braces do not prevent separating arguments

```
#define STR(EXP) #EXP
cout << STR((1, 2, 3)) << endl;     # prints (1, 2, 3)

cout << STR([1, 2, 3]) << endl;
error: macro "STR" passed 3 arguments, but takes just 1

// args preserves the comma and spacing between arguments
#define VSTR(args...) #args
cout << VSTR([1, 2, 3]) << endl;    # prints [1, 2, 3]
```

Parenthesis Detection

- Checks to see if argument is inside parenthesis

```
#define SECOND(x, n, ...) n
#define CHECK(...) CHECK_N(__VA_ARGS__, 0,)
#define PROBE(x) x, 1,
#define IS_PAREN(x) CHECK(IS_PAREN_PROBE x)
#define IS_PAREN_PROBE(...) PROBE(!)

IS_PAREN((1, 2))  →  CHECK(IS_PAREN_PROBE (1 , 2))
                  →  CHECK(PROBE(!))
                  →  CHECK(!, 1,)
                  →  SECOND(!, 1, 0,)
                  →  1
IS_PAREN(hi)      →  CHECK(IS_PAREN_PROBE hi)
                  →  SECOND(IS_PAREN_PROBE hi, 0,)
                  →  0
```

IS_PAREN_PROBE
is not a macro constant

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)
#define y (2 * x)
```

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

Self-Referential Macros

- When a macro expands, it is disabled, which prevents further expansion of same macro *in the same scan*
- Can cause another macro to not expand
- E.g. deferred expression

```
#define EMPTY()
#define DEFER(x) x EMPTY()
#define EXPAND(...) __VA_ARGS__

int A() { return 456; }          // not affected A() macro
#define A() 123
DEFER(A)()  →  A EMPTY()()       // A cannot expanded here
            →  A ()              // requires one more scan
printf("%d", DEFER(A)());        // prints 456
```

Self-Referential Macros

- Forcing another scan

```
#define EMPTY()
#define DEFER(x) x EMPTY()
#define EXPAND(...) __VA_ARGS__

int A() { return 456; }        // not affected A() macro
#define A() 123

EXPAND(DEFER(A)())      →  EXPAND(A EMPTY()())
                        →  EXPAND(A ())
                        →  A ()
                        →  123

printf("%d", EXPAND(DEFER(A)()));   // prints 123
```

- This behaviour can be used to implement recursion

Advanced Concatenation

- Can be used to create token that is another macro

```
#define CAT(a, args...) a ## args
#define IFF(c) CAT(IFF_, c)
#define IFF_0(t, ...) __VA_ARGS__
#define IFF_1(t, ...) t
#define FALSE 0
#define CAN_DO() 1

IFF(FALSE)(5, 9)          →    IFF(0)(5, 9)
                          →    IFF_0(5, 9)
                          →    9

IFF(CAN_DO())(5, 9)       →    IFF(1)(5, 9)
                          →    IFF_1(5, 9)
                          →    5
```

When Statement

- Previous example
  - Only works if macro expands to 1 or 0
  - We want a generalized when statement

```
WHEN(cond, true-expression, false-expression)
```

- Idea:
- !! operator (double negation)
  - Converts a number to 1 or 0, E.g. !!12 = !0 = 1
  - Can be achieved using macro's pattern matching

When Statement

- Try 1:

```
#define SECOND(a, b, ...) b
#define CHECK(...) SECOND(__VA_ARGS__, 0)
#define PROBE() ~, 1
#define NOT(x) CHECK(_NOT_ ## x)
#define _NOT_0 PROBE()
#define BOOL(x) NOT(NOT(x))

BOOL(123)   →  NOT(NOT(123))
            →  CHECK(_NOT_ ## NOT(123))
            →  CHECK(_NOT_NOT(123))
            →  SECOND(_NOT_NOT(123), 0)
            →  0
```

/facepalm

When Statement

- Try 2:

```
#define CAT(a, args...) a ## args
#define SECOND(a, b, ...) b
#define CHECK(...) SECOND(__VA_ARGS__, 0)
#define PROBE() ~, 1
#define NOT(x) CHECK(CAT(_NOT_, x))
#define _NOT_0 PROBE()
#define BOOL(x) NOT(NOT(x))

BOOL(123)   → NOT(NOT(123)) →  NOT(CHECK(CAT(_NOT_,123)))
   →  NOT(CHECK(_NOT_123))) →  NOT(SECOND(_NOT_123, 0))
   →  NOT(0)                →  CHECK(CAT(_NOT_, 0))
   →  CHECK(_NOT_0)         →  CHECK(PROBE())
   →  CHECK(~, 1)           →  SECOND(~, 1, 0)
   →  1
```

When Statement

- Joining with previous example

```
#define CAT(a, args...) a ## args
#define SECOND(a, b, ...) b
#define CHECK(...) SECOND(__VA_ARGS__, 0)
#define PROBE() ~, 1
#define NOT(x) CHECK(CAT(_NOT_, x))
#define _NOT_0 PROBE()
#define BOOL(x) NOT(NOT(x))
#define IFF(c) CAT(IFF_, c)
#define IFF_0(t, ...) __VA_ARGS__
#define IFF_1(t, ...) t
#define WHEN(cond, t, f) IFF(BOOL(cond))((t), (f))

int a = WHEN(12, 5, 7), b = WHEN(0, 3, 8);

a = 5, b = 8
```

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
```

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

__cplusplus

- A predefined macro
- Used if mixing C and C++ code
  - This requires lots of care, because C is not a subset of C++
- Extern “C”
  - Code within this block are C code, not C++

```
#ifdef __cplusplus
extern “C” {
#endif
      struct hand * alloc_hand(struct shoe * new);
#ifdef __cplusplus
}
#endif
```

Version Control

- Integer, comparison, relational operators are supported

```
#if EASYDB_VERSION > 1
#define ALLOW_SURRENDER
#endif

int foo() {
#if VERBOSE >= 2
      printf(“entering foo");
#endif
      …
}

#if !(defined __LP64__ || defined __LLP64__) || \
      defined _WIN32 && !defined _WIN64
      // we are compiling for a 32-bit system
#endif
```

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