12 Macro

Programming in C

Teacher: Po-Wen Chi

neokent@gapps.ntnu.edu.tw

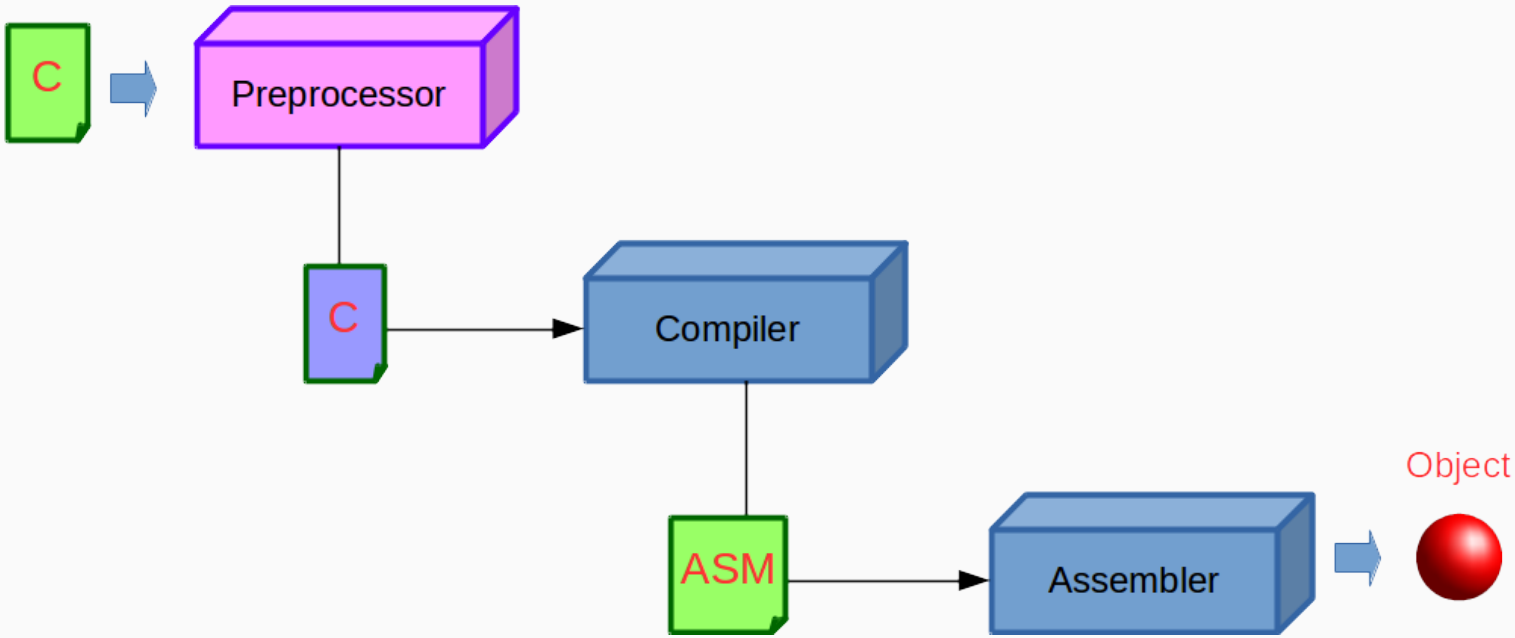
January 4, 2019

Department of Computer Science and Information Engineering,
National Taiwan Normal University

Preprocessor

Preprocessor

- Before the compiler compiles your source code into assembly codes, there is an additional stage called **Preprocessing**.
- Objective: Produce a source code file with preprocessing commands properly sorted out.



Preprocessor Directives

- Preprocessor commands are known as **Directives**.
- Preprocessor provides certain **features**, which are called **preprocessor directives**.
- Preprocessor directive starts with #.

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

Preprocessor Directives

- Preprocessor directives can be places any where in the source program.
- Note: **Place it at the start of your program**.
- Each directive must be on its own line.

Please see oneline.c.

After Preprocessor

```
#include <stdio.h>

#define MACRO_DEMO 123

int main()
{
    printf( "demo: %d\n", MACRO_DEMO );
    return 0;
}
```

gcc -E macro_demo.c.

Preprocessor Directives

- Macro Expansion.
- File Inclusion.
- Conditional Compilation.

Macro

Macro Expansion

- **#define** is known as **macro expansion**.
- Example:
  - #define PI 3.14
- General Form:
  - **#define macro_template macro_expansion**

Macro Expansion

- Preprocessor searches for macro definition.
- After finding macro definition, it searches the whole program for **macro_template**.
- **Replace** every **macro_template** with **macro_expansion**.
  - Replacement will not occur if the template is in a quoted string.
- Notes:
  - In practice, we use all capital letters for macro_template.
  - Do not use semicolon ;

Macros, Why??

- To write efficient programs.
- To increase readability of your program.
- Defined macro name can be used as part of definition of other macros.

Please see macro_example.c .

Macro with Arguments

- Macros can have arguments, like functions.
  - #define IS_GREAT(x) (x >= GOOD_THRESHOLD)
- Notes:
  - When defining macros, space between arguments and name is not allowed.
  - Macro expansions should be enclosed between parenthesis.
  - Use '\' to split a macro into multiple lines.

Please see macro_example_2.c and macro_example_3.c.

Macro vs. Function

- **Macro**:
  - Just replacement.
  - Faster than function, though you may not be aware.
- **Function**:
  - Passing arguments, doing works and returning the result.
  - Support **recursive** call.

No Recursive Macro

Spec

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's pre-processing tokens), **it is not replaced**. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

inline Functions

- inline is a CPP keyword.
- C includes this keyword from C99. Though GNU C (and some other compilers) had inline functions long before standard C.
- Comparison:
  1. inline is just a suggestion.
  2. inline will check types.
- I will not talk too much about this keyword since it is different from CPP. I do not want to mislead you.
- If you are interested in this topic, please read the following url. `http://www.greenend.org.uk/rjk/tech/inline.html`

Macro vs. Constant vs. Enum

```
#define UNKNOWN  0
#define SUNDAY   1
#define MONDAY   2

typedef enum {
    UNKNOWN,
    SUNDAY,
    MONDAY
} Weekday;

const int32_t UNKNOWN=0;
const int32_t SUNDAY=1;
const int32_t MONDAY=2;
```

Which one do you like?

Up to You

It depends. You can choose what you like.

I know others may tell you their choices and reasons, but I think this is a style questions instead of right or wrong.

File Inclusion

Why File Inclusion?

- Divide a program in multiple files.
  - Each file contains related functions.
  - How to classify functions. Up to you.
- Some functions or macros are required in lot of programs.
  - Put them in a file. Make them a library.
  - Include them when you need them.

What does Inclusion Mean?

Nothing but simply **copy and paste**.

Nested Inclusion is supported.

What does Inclusion Mean?

Nothing but simply **copy and paste**.

Nested Inclusion is supported.

What if I include stdio.h twice?

What does Inclusion Mean?

Nothing but simply **copy and paste**.

Nested Inclusion is supported.

What if I include stdio.h twice?

I will talk this latter

Is This a Valid Code?

```c
#include <stdio.h>
#include <stdint.h>

int32_t max( int32_t, int32_t );

int main()
{
    int32_t a = 0;
    int32_t b = 1;

    printf( "max: %d\n", max( a, b ) );

    return 0;
}
```

gcc -c include_test01.c

gcc include_test01.c

Conditional Compilation

Conditional Compilation

- Write one code to run on different environments.
  - **#if**: if.
  - **#else**: else.
  - **#elif**: else if.
  - **#endif**: end if.
  - **#ifdef**: if defined.
  - **#ifndef**: if not defined.

How to Comment the Following Codes?

```
for( int i = 0 ; i < 5 ; i++ )
{
    score[i] += 10; /* Curve */
}
```

How about This?

```
/*
for( int i = 0 ; i < 5 ; i++ )
{
    score[i] += 10; /* Curve */
}
*/
```

What You Should Do

```
#if 0
    for( int i = 0 ; i < 5 ; i++ )
    {
        score[i] += 10; /* Curve */
    }
#endif
```

ifdef and endif

General Form

```
#ifdef macroname
    statement sequence.
#endif
```

If **macroname** is defined, then the code between #ifdef and #endif will be executed.

Why do I need this? Please see os_dependent.c.

Wait a Minute!!

- Nothing happens!

Wait a Minute!!

- Nothing happens!
- How about this?
  - gcc **-D** OS_LINUX os_dependent.c
  - gcc **-D** OS_WIN os_dependent.c

Wait a Minute!!

- Nothing happens!
- How about this?
  - gcc **-D** OS_LINUX os_dependent.c
  - gcc **-D** OS_WIN os_dependent.c
- **-D**: Predefine name as a macro, with definition 1.

Wait a Minute!!

- Nothing happens!
- How about this?
  - gcc **-D** OS_LINUX os_dependent.c
  - gcc **-D** OS_WIN os_dependent.c
- **-D**: Predefine name as a macro, with definition 1.
- So one code can be executed on different OSs. What you need to do is to **build your code with different definitions**.

Wait a Minute!!

- Nothing happens!
- How about this?
  - gcc **-D** OS_LINUX os_dependent.c
  - gcc **-D** OS_WIN os_dependent.c
- **-D**: Predefine name as a macro, with definition 1.
- So one code can be executed on different OSs. What you need to do is to **build your code with different definitions**.
- This is a very useful technique. Let's see a real case.
  `https://github.com/DaveGamble/cJSON/blob/master/cJSON.h`

Another Scenario

Please see debug.c.

In this case, when releasing your program to customers, you only need to rebuild your code without change anything.

Inclusion

- We also use this technique to avoid paste header files twice.
- Again, you should use #pragma once.

Practice

Please see the debug example. Rewrite this example and add a feature about debug_level.