Lecture 14 - The Preprocessor

Meng-Hsun Tsai
CSIE, NCKU

Source Code → Preprocessor → Compiler → Linker → Executable Program

Introduction

- Directives such as `#define` and `#include` are handled by the ***preprocessor,*** a piece of software that edits C programs just prior to compilation.
- Its reliance on a preprocessor makes C (along with C++) unique among major programming languages.
- The preprocessor is a powerful tool, but it also can be a source of hard-to-find bugs.

14.1 How the Preprocessor Works

How the Preprocessor Works

- The preprocessor looks for ***preprocessing directives***, which begin with a `#` character.
- We've encountered the `#define` and `#include` directives before.
- `#define` defines a ***macro***—a name that represents something else, such as a constant.
- The preprocessor responds to a `#define` directive by storing the name of the macro along with its definition.
- When the macro is used later, the preprocessor "expands" the macro, replacing it by its defined value.

How the Preprocessor Works (cont.)

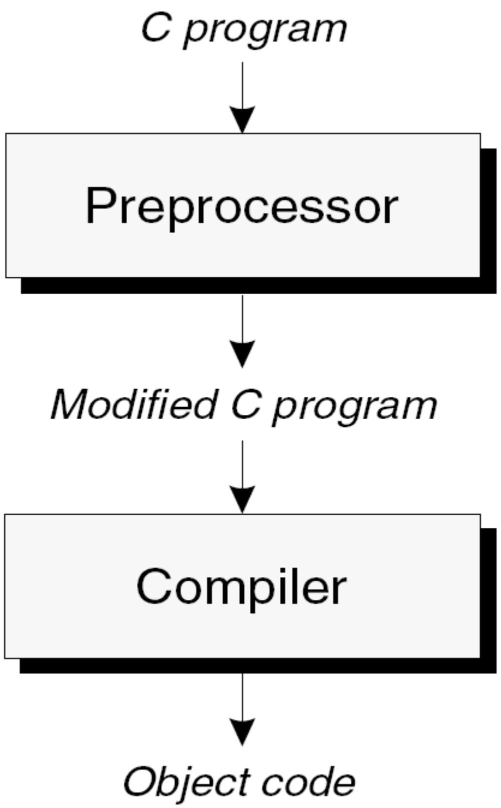
- `#include` tells the preprocessor to open a particular file and "include" its contents as part of the file being compiled.
- For example, the line

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

  instructs the preprocessor to open the file named `stdio.h` and bring its contents into the program

How the Preprocessor Works (cont.)

- The right side shows the role of the preprocessor in the compilation process.
- The input to the preprocessor is a C program, possibly containing directives.
- The preprocessor executes these directives, removing them in the process.
- The preprocessor's output goes directly into the compiler.



How the Preprocessor Works (cont.)

- The `celsius.c` program of Lecture 2:

```
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)
int main(void)
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);
    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", celsius);
    return 0;
}
```

How the Preprocessor Works (cont.)

- The program after preprocessing:

stdio.h

Lines brought in from stdio.h

```
int main(void)
{
    float fahrenheit, celsius;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);
    celsius = (fahrenheit - 32.0f) * (5.0f / 9.0f);
    printf("Celsius equivalent: %.1f\n", celsius);
    return 0;
}
```

How the Preprocessor Works (cont.)

- The preprocessor does a bit more than just execute directives.
- In particular, it replaces each comment with a single space character.

  eg. //, /*, */
- Some preprocessors go further and remove unnecessary white-space characters, including spaces and tabs at the beginning of indented lines.

How the Preprocessor Works (cont.)

- In the early days of C, the preprocessor was a separate program.
- Nowadays, the preprocessor is often part of the compiler, and some of its output may not necessarily be C code.
- Still, it's useful to think of the preprocessor as separate from the compiler.

How the Preprocessor Works (cont.)

- To view the output of the preprocessor within `gcc`, use the `-E` option.

```
gcc -E -o program program.c
```

- A word of caution: The preprocessor has only a limited knowledge of C.
- As a result, it's quite capable of creating illegal programs as it executes directives.

Preprocessing Directives

- Most preprocessing directives fall into one of three categories:
  - ***Macro definition.*** The `#define` directive defines a macro; the `#undef` directive removes a macro definition.
  - ***File inclusion.*** The `#include` directive causes the contents of a specified file to be included in a program.
  - ***Conditional compilation.*** The `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` directives allow blocks of text to be either included in or excluded from a program.

Preprocessing Directives (cont.)

- Several rules apply to all directives.
- ***Directives always begin with the # symbol.***

  The `#` symbol need not be at the beginning of a line, as long as only white space precedes it.
- ***Any number of spaces and horizontal tab characters may separate the tokens in a directive.*** Example:

```
#     define       N       100
```

Preprocessing Directives (cont.)

- ***Directives always end at the first new-line character, unless explicitly continued.***

To continue a directive to the next line, end the current line with a \ character:

```
#define DISK_CAPACITY (SIDES *              \
                       TRACKS_PER_SIDE *    \
                       SECTORS_PER_TRACK *  \
                       BYTES_PER_SECTOR)
```

Preprocessing Directives

- ***Directives can appear anywhere* in a program.**

  Although `#define` and `#include` directives usually appear at the beginning of a file, other directives are more likely to show up later.

- ***Comments may appear on the same line* as a directive.**

  It's good practice to put a comment at the end of a macro definition:

```
#define FREEZING_PT 32.0f /* freezing point of water */
```

14.2 Macro Definitions

Macro Definitions

- The macros that we've been using since Lecture 2 are known as *simple* macros, because they have no parameters.
- The preprocessor also supports *parameterized* macros.

Simple Macros

- Definition of a ***simple macro*** (or ***object-like macro***):

  `#define` *identifier replacement-list*

  replacement-list is any sequence of ***preprocessing tokens.***

- The replacement list may include identifiers, keywords, numeric constants, character constants, string literals, operators, and punctuation.

- Wherever *identifier* appears later in the file, the preprocessor replaces it with the *replacement-list*.

Simple Macros (cont.)

- Any extra symbols in a macro definition will become part of the replacement list.
- Putting the = symbol in a macro definition is a common error:

```
#define N = 100   /*** WRONG ***/
…
int a[N];         /* becomes int a[= 100]; */
```

Simple Macros (cont.)

- Putting the = symbol in a macro definition is a common error:

```
#define N = 100   /*** WRONG ***/
…
int a[N];         /* becomes int a[= 100]; */
```

- Ending a macro definition with a semicolon is another popular mistake:

```
#define N 100;   /*** WRONG ***/
…
int a[N];        /* becomes int a[100;]; */
```

- The compiler will detect most errors caused by extra symbols in a macro definition but can not identify the actual reason (i.e., the error in the macro's definition).

Simple Macros (cont.)

- Simple macros are primarily used for defining "manifest constants"—names that represent numeric, character, and string values:

```
#define STR_LEN 80
#define TRUE    1
#define FALSE   0
#define PI      3.14159
#define CR      '\r'
#define EOS     '\0'
#define MEM_ERR "Error: not enough memory"
```

Simple Macros (cont.)

- Advantages of using `#define` to create names for constants:
  - ***It makes programs easier to read.*** The name of the macro can help the reader understand the meaning of the constant.
  - ***It makes programs easier to modify.*** We can change the value of a constant throughout a program by modifying a single macro definition.
  - ***It helps avoid inconsistencies and typographical errors.*** If a numerical constant like 3.14159 appears many times in a program, chances are it will occasionally be written 3.1416 or 3.14195 by accident.

Simple Macros (cont.)

- ***Controlling conditional compilation***

  Macros play an important role in controlling conditional compilation.

  A macro that might indicate “debugging mode”:

```
#define DEBUG
…
#ifdef DEBUG
   printf("check point 1\n");
#endif
…
```

Parameterized Macros

- Definition of a ***parameterized macro*** (also known as a ***function-like macro***):

  `#define` *identifier*`(` x_1 `,` x_2 `,` … `,` x_n `)` *replacement-list*

  x_1, x_2, …, x_n are identifiers (the macro's ***parameters***).

- The parameters may appear as many times as desired in the replacement list.

- There must be *no space* between the macro name and the left parenthesis.

- If space is left, the preprocessor will treat `(`x_1, x_2, …, x_n`)` as part of the replacement list.

Parameterized Macros (cont.)

- Wherever a macro ***invocation*** of the form *identifier* (y_1, y_2, …, y_n) appears later in the program, the preprocessor replaces it with *replacement-list*, substituting y_1 for x_1, y_2 for x_2, and so forth.
- Parameterized macros often serve as simple functions.

Parameterized Macros (cont.)

- Examples of parameterized macros:

```
#define MAX(x,y)    ((x)>(y)?(x):(y))
#define IS_EVEN(n) ((n)%2==0)
```

- Invocations of these macros:

```
i = MAX(j+k, m-n);
if (IS_EVEN(i)) i++;
```

- The same lines after macro replacement:

```
i = ((j+k)>(m-n)?(j+k):(m-n));
if (((i)%2==0)) i++;
```

Parameterized Macros (cont.)

- A more complicated function-like macro:

```
#define TOUPPER(c) \
   ('a'<=(c)&&(c)<='z'?(c)-'a'+'A':(c))
```

- The `<ctype.h>` header provides a similar function named `toupper` that's more portable.
- A parameterized macro may have an empty parameter list:

```
#define getchar() getc(stdin)
```

- The empty parameter list isn't really needed, but it makes `getchar` resemble a function.

Parameterized Macros (cont.)

- Using a parameterized macro instead of a true function has a couple of advantages:
  - ***The program may be slightly faster.*** A function call usually requires some overhead during program execution, but a macro invocation does not.
  - ***Macros are "generic."*** A macro can accept arguments of any type, provided that the resulting program is valid.

Parameterized Macros (cont.)

- Parameterized macros also have disadvantages.
- ***The compiled code will often be larger.***

  Each macro invocation increases the size of the source program (and hence the compiled code).

  The problem is compounded when macro invocations are nested:

```
n = MAX(i, MAX(j, k));
```

  The statement after preprocessing:

```
n = ((i)>(((j)>(k)?(j):(k)))?(i):(((j)>(k)?(j):(k))));
```

Parameterized Macros (cont.)

- ***Arguments aren't type-checked.***

  When a function is called, the compiler checks each argument to see if it has the appropriate type.

  Macro arguments aren't checked by the preprocessor, nor are they converted.

Parameterized Macros (cont.)

- ***A macro may evaluate its arguments more than once.***

  Unexpected behavior may occur if an argument has side effects:

  ```
  n = MAX(i++, j);
  ```

  The same line after preprocessing:

  ```
  n = ((i++)>(j)?(i++):(j));
  ```

  If `i` is larger than `j`, then `i` will be (incorrectly) incremented twice and `n` will be assigned an unexpected value.

Parameterized Macros (cont.)

- Parameterized macros can be used as patterns for segments of code that are often repeated.
- A macro that makes it easier to display integers:

```
#define PRINT_INT(n) printf("%d\n", n)
```

- The preprocessor will turn the line

```
PRINT_INT(i/j);
```

into

```
printf("%d\n", i/j);
```

The # Operator

- Macro definitions may contain two special operators, # and ##.
- Neither operator is recognized by the compiler; instead, they're executed during preprocessing.
- The # operator converts a macro argument into a string literal; it can appear only in the replacement list of a parameterized macro.
- The operation performed by # is known as "stringization."

The # Operator (cont.)

- There are a number of uses for `#`; let’s consider just one.
- Suppose that we decide to use the `PRINT_INT` macro during debugging as a convenient way to print the values of integer variables and expressions.
- The `#` operator makes it possible for `PRINT_INT` to label each value that it prints.

The # Operator (cont.)

- Our new version of `PRINT_INT`:

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
```

- The invocation

```
PRINT_INT(i/j);
```

  will become

```
printf("i/j" " = %d\n", i/j);
```

- The compiler automatically joins adjacent string literals, so this statement is equivalent to

```
printf("i/j = %d\n", i/j);
```

The ## Operator

- The ## operator can "paste" two tokens together to form a single token.
- If one of the operands is a macro parameter, pasting occurs after the parameter has been replaced by the corresponding argument.

The ## Operator (cont.)

- A macro that uses the `##` operator:

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

- A declaration that invokes `MK_ID` three times:

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

- The declaration after preprocessing:

```
int i1, i2, i3;
```

General Properties of Macros

- Several rules apply to both simple and parameterized macros.
- ***A macro's replacement list may contain invocations of other macros.***

  Example:

  ```
  #define PI      3.14159
  #define TWO_PI (2*PI)
  ```

  When it encounters `TWO_PI` later in the program, the preprocessor replaces it by `(2*PI)`.

  The preprocessor then ***rescans*** the replacement list to see if it contains invocations of other macros.

General Properties of Macros (cont.)

- ***The preprocessor replaces only entire tokens.***

  Macro names embedded in identifiers, character constants, and string literals are ignored.

  Example:

```
#define SIZE 256

int BUFFER_SIZE;

if (BUFFER_SIZE > SIZE)
  puts("Error: SIZE exceeded");
```

  Appearance after preprocessing:

```
int BUFFER_SIZE;

if (BUFFER_SIZE > 256)
  puts("Error: SIZE exceeded");
```

General Properties of Macros (cont.)

- ***A macro definition normally remains in effect until the end of the file in which it appears.***

  Macros don't obey normal scope rules.

  A macro defined inside the body of a function isn't local to that function; it remains defined until the end of the file.

- ***Macros may be "undefined" by the `#undef` directive.***

  The `#undef` directive has the form

  `#undef` *identifier*

  where *identifier* is a macro name.

  One use of `#undef` is to remove the existing definition of a macro so that it can be given a new definition.

Parentheses in Macro Definitions

- The replacement lists in macro definitions often require parentheses in order to avoid unexpected results.
- If the macro's replacement list contains an operator, always enclose the replacement list in parentheses:

```
#define TWO_PI (2*3.14159)
```

- Also, put parentheses around each parameter every time it appears in the replacement list:

```
#define SCALE(x) ((x)*10)
```

- Without the parentheses, we can't guarantee that the compiler will treat replacement lists and arguments as whole expressions.

Parentheses in Macro Definitions (cont.)

- An example that illustrates the need to put parentheses around a macro's replacement list:

```
#define TWO_PI 2*3.14159
/* needs parentheses around replacement list */
```

- During preprocessing, the statement

```
conversion_factor = 360/TWO_PI;
```

  becomes

```
conversion_factor = 360/2*3.14159;
```

The division will be performed before the multiplication.

Parentheses in Macro Definitions (cont.)

- Each occurrence of a parameter in a macro's replacement list needs parentheses as well:

```
#define SCALE(x) (x*10)
  /* needs parentheses around x */
```

- During preprocessing, the statement

```
j = SCALE(i+1);
```

  becomes

```
j = (i+1*10);
```

  This statement is equivalent to

```
j = i+10;
```

Creating Longer Macros

- The comma operator can be useful for creating more sophisticated macros by allowing us to make the replacement list a series of expressions.
- A macro that reads a string and then prints it:

```
#define ECHO(s) (gets(s), puts(s))
```

- Calls of `gets` and `puts` are expressions, so it's perfectly legal to combine them using the comma operator.
- We can invoke `ECHO` as though it were a function:

```
ECHO(str);  /* becomes (gets(str), puts(str)); */
```

Predefined Macros

- C has several predefined macros, each of which represents an integer constant or string literal.
- The `__DATE__` and `__TIME__` macros identify when a program was compiled.
- Example of using `__DATE__` and `__TIME__`:

```
printf("Wacky Windows (c) 2010 Wacky Software, Inc.\n");
printf("Compiled on %s at %s\n", __DATE__, __TIME__);
```

- Output produced by these statements:

```
Wacky Windows (c) 2010 Wacky Software, Inc.
Compiled on Dec 23 2010 at 22:18:48
```

- This information can be helpful for distinguishing among different versions of the same program.

Predefined Macros (cont.)

- We can use the `__LINE__` and `__FILE__` macros to help locate errors.
- A macro that can help pinpoint the location of a division by zero:

```
#define CHECK_ZERO(divisor) \
  if (divisor == 0) \
    printf("*** Attempt to divide by zero on line %d " \
           "of file %s ***\n", __LINE__, __FILE__)
```

- The `CHECK_ZERO` macro is invoked prior to a division:

```
CHECK_ZERO(j);
k = i / j;
```

- If `j` happens to be zero, a message of the following form will be printed:

```
*** Attempt to divide by zero on line 9 of file foo.c ***
```

The `__func__` Identifier

- The `__func__` identifier behaves like a string variable that stores the name of the currently executing function.
- The effect is the same as if each function contains the following declaration at the beginning of its body:

  `static const char __func__[] = "`*`function-name`*`";`

  where *function-name* is the name of the function.

The `__func__` Identifier (C99)

- Debugging macros that rely on the `__func__` identifier:

```
#define FUNCTION_CALLED() printf("%s called\n", __func__);
#define FUNCTION_RETURNS() printf("%s returns\n", __func__);
```

- These macros can used to trace function calls:

```
void f(void)
{
  FUNCTION_CALLED();    /* displays "f called" */
  …
  FUNCTION_RETURNS();   /* displays "f returns" */
}
```

- Another use of `__func__`: it can be passed to a function to let it know the name of the function that called it.

14.3 Conditional Compilation

Conditional Compilation

- The C preprocessor recognizes a number of directives that support ***conditional compilation.***
- This feature permits the inclusion or exclusion of a section of program text depending on the outcome of a test performed by the preprocessor.

The `#if` and `#endif` Directives

- Suppose we're in the process of debugging a program.
- We'd like the program to print the values of certain variables, so we put calls of `printf` in critical parts of the program.
- Once we've located the bugs, it's often a good idea to let the `printf` calls remain, just in case we need them later.
- Conditional compilation allows us to leave the calls in place, but have the compiler ignore them.

The `#if` and `#endif` Directives (cont.)

- The first step is to define a macro and give it a nonzero value:

```
#define DEBUG 1
```

- Next, we'll surround each group of `printf` calls by an `#if`-`#endif` pair:

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

The **`#if`** and **`#endif`** Directives (cont.)

- During preprocessing, the `#if` directive will test the value of `DEBUG`.
- Since its value isn't zero, the preprocessor will leave the two calls of `printf` in the program.
- If we change the value of `DEBUG` to zero and recompile the program, the preprocessor will remove all four lines from the program.
- The `#if`-`#endif` blocks can be left in the final program, allowing diagnostic information to be produced later if any problems turn up.

The `#if` and `#endif` Directives (cont.)

- General form of the `#if` and `#endif` directives:

  `#if` *constant-expression*
  `#endif`

- When the preprocessor encounters the `#if` directive, it evaluates the constant expression.
- If the value of the expression is zero, the lines between `#if` and `#endif` will be removed from the program during preprocessing.
- Otherwise, the lines between `#if` and `#endif` will remain.

The **`#if`** and **`#endif`** Directives (cont.)

- The `#if` directive treats undefined identifiers as macros that have the value 0.
- If we neglect to define `DEBUG`, the test

  ```
  #if DEBUG
  ```

  will fail (but not generate an error message).
- The test

  ```
  #if !DEBUG
  ```

  will succeed.

The `defined` Operator

- The preprocessor supports three operators: `#`, `##`, and `defined`.
- When applied to an identifier, `defined` produces the value 1 if the identifier is a currently defined macro; it produces 0 otherwise.
- The `defined` operator is normally used in conjunction with the `#if` directive.

The `defined` Operator (cont.)

- Example:

```
#if defined(DEBUG)
…
#endif
```

- The lines between `#if` and `#endif` will be included only if `DEBUG` is defined as a macro.
- The parentheses around `DEBUG` aren't required:

```
#if defined DEBUG
```

- It's not necessary to give `DEBUG` a value:

```
#define DEBUG
```

The `#ifdef` and `#ifndef` Directives

- The `#ifdef` directive tests whether an identifier is currently defined as a macro:

  `#ifdef` *identifier*

- The effect is the same as

  `#if defined(`*identifier*`)`

- The `#ifndef` directive tests whether an identifier is *not* currently defined as a macro:

  `#ifndef` *identifier*

- The effect is the same as

  `#if !defined(`*identifier*`)`

The `#elif` and `#else` Directives

- `#if`, `#ifdef`, and `#ifndef` blocks can be nested just like ordinary `if` statements.
- When nesting occurs, it's a good idea to use an increasing amount of indentation as the level of nesting grows.
- Some programmers put a comment on each closing `#endif` to indicate what condition the matching `#if` tests:

```
#if DEBUG
…
#endif /* DEBUG */
```

The `#elif` and `#else` Directives (cont.)

- `#elif` and `#else` can be used in conjunction with `#if`, `#ifdef`, or `#ifndef` to test a series of conditions:

  ```
  #if expr1
  Lines to be included if expr1 is nonzero
  #elif expr2
  Lines to be included if expr1 is zero but expr2 is nonzero
  #else
  Lines to be included otherwise
  #endif
  ```

- Any number of `#elif` directives—but at most one `#else`—may appear between `#if` and `#endif`.

Uses of Conditional Compilation

- Conditional compilation has other uses besides debugging.
- ***Writing programs that are portable to several machines or operating systems.***

  Example:

```
#if defined(WIN32)
…
#elif defined(MAC_OS)
…
#elif defined(LINUX)
…
#endif
```

Uses of Conditional Compilation

- ***Providing a default definition for a macro.***

  Conditional compilation makes it possible to check whether a macro is currently defined and, if not, give it a default definition:

```
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 256
#endif
```

- ***Temporarily disabling code that contains comments.***

  A `/*…*/` comment can't be used to "comment out" code that already contains `/*…*/` comments.

  An `#if` directive can be used instead:

```
#if 0
Lines containing comments
#endif
```