The Preprocessor

Introduction

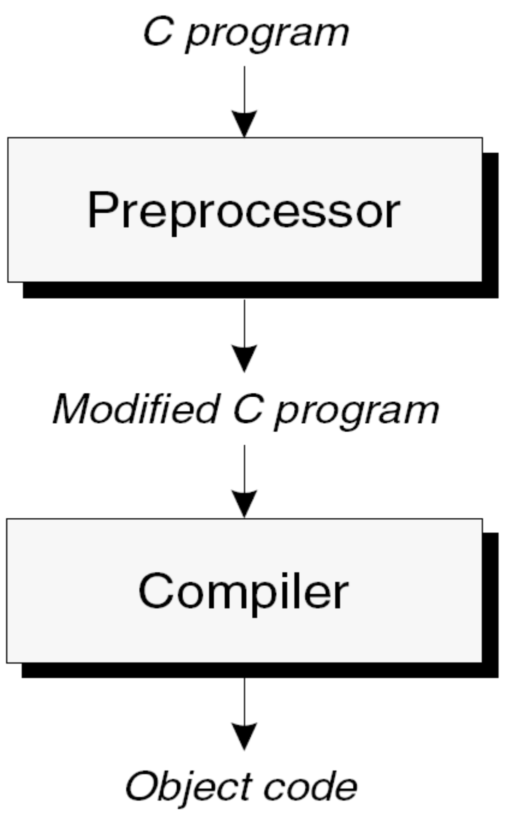
- Directives such as

  ```
  #define
  ```

  ```
  #include
  ```

  are handled by the ***preprocessor* (*前置處理器*),**
  a piece of software that edits C programs just prior
  to compilation.

How the Preprocessor Works



Introduction

- 前置處理器的 input 是 C 的程式碼，
  其中可能包含待處理的 directives 命令
- 前置處理器會執行這些 directives、將它們從程式碼中移除、並且對程式碼做相對應的修改.
- 之後，修改過的程式碼就直接成為 C compiler 的 input.

- 前置處理器是一個很有用的工具，
  但同時也可能是很難以發現的 bug 的來源

How the Preprocessor Works

- 在 C 發展早期，前置處理器是一隻獨立的程式在編譯前用來對程式碼做前置處理
- 現今，前置處理器多半已成為編譯器的一部份，甚至，在編譯器最佳化的設計下，許多已經與編譯器結合。
- 然而，使用時把它當成是一隻獨立的程式，有助於理解前置處理器的行為與功用。

Preprocessing Directives

- 前置處理器的命令 (directives) 主要分成三類：
  - *定義 Macro.*
    The **`#define`** directive defines a macro;
    the **`#undef`** directive removes a macro definition.
  - *引入檔案 File inclusion.*
    The **`#include`** directive causes the contents of a specified file to be included in a program.
  - *條件判斷 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.

#include

- `#include` 告訴前置處理器去開啟特定的檔案，並且將它的內容包含至目前的程式碼裡。

- 舉例來說，

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

告訴前置處理器去開啟 stdio.h 這個檔案、並且將它的內容替換至此 directive 命令的位置

#define

- `#define` 用來定義 ***macro*** —

  一個用來代表自定值的名字(前置處理器的識別字)

- 前置處理器會將 `#define` 所定義的 macro 儲存下來

- 當定義的 macro 被使用時， 前置處理器會把它“展開”, 將它替換成所定義的值.

Simple Macros

Simple Macros

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

  `#define` *identifier* *replacement-list*

  replacement-list 為任意的 ***preprocessing tokens.***

- Replacement list 可以包含任何的 identifiers, keywords, 數值常數, 字元常數, 字串, 運算子, 或標點

Simple Macros

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

  `#define` *identifier* *replacement-list*

  replacement-list 為任意的 ***preprocessing tokens.***

- 當 *identifier* 之後再出現於程式檔中，
  前置處理器會將它置換為 *replacement-list*.

Simple Macros

- 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

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

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

Simple Macros

- 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

- 定義 Simple Macros 的優點:
  - *讓程式碼更容易閱讀.*
    The name of the macro can help the reader understand the meaning of the constant.
  - *讓程式碼容易維護、修改.*
    We can change the value of a constant throughout a program by modifying a single macro definition.

    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.

Parameterized Macros

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

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

  $x_1, x_2, \ldots, x_n$ 為此 macro 的**參數**

- Macro 的識別字與 左括號 '(' 之間不可以有空格.

  否則的話, ($x_1, x_2, \ldots, x_n$) 會被視為 replacement list 的一部份.

Parameterized Macros

- 當前置處理器碰到 Function Macro 的定義時，會將它的定義儲存下來.
- 在程式碼裡，當 function macro 被使用時，(形式必須為 *identifier* ($y_1, y_2, \ldots, y_n$))
前置處理器會將它替為成定義的內容，
並且將各個參數代入.
- 可以把 Function Macro 視為簡化版的函式.

 不同的地方在於, Function Macro 做的是 “代入”
 沒有實際的函式呼叫過程

Parameterized Macros

- 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

- 使用 Function Macro 的優點:
  - *程式碼執行上, 會稍微快一點點.*
    A function call usually requires some overhead during program execution, but a macro invocation does not.
  - ***Function Macro** 的參數可以是任何型態.*
    A macro can accept arguments of any type, provided that the resulting program is valid.

Parameterized Macros

- Function Macro 的缺點：
- *編譯後的程式碼會較為肥大:*

  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

- ***前置處理器只做“替換”，
  無法檢查Macro 的參數型態.***

  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

- ***過度使用Macro 的替換，
  可能帶來邏輯上難以發現的bug.***

  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.