Notes 9.0: The preprocessor

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2013 session 1

Generalities

The preprocessor processes preprocessor commands, which are lines of the source file that begin with #, possibly preceded by spaces.

Running the preprocessor results in all preprocessor commands being removed from the source file, some lines in the source file being modified, and some lines being added to the source file, making up a valid C program that is ready for compilation.

Giving the option `-E` to `gcc` sends to standard output the preprocessed file, without running the compiler proper.

The #include command

The `#include` preprocessor command allows one to include the contents of a source file, usually a header file, in place of the command.

The argument to `#include` can be in one of 3 possible forms.

- It can be a file name preceded by `<` possibly followed by spaces, and followed by `>` possibly preceded by spaces, in which case the file will be searched for in implementation-defined directories.
- It can be a file name preceded by `"` possibly followed by spaces, and followed by `"` possibly preceded by spaces, in which case the file name can be absolute or relative. If it is relative, it will be searched for in the current directory, before being searched in the implementation-defined directories in case it is not found in the latter.
- It can be a macro that, after expansion, must match one of the first two forms.

Use of the three forms is illustrated in include.c

The #define and #undef commands (1)

The `#define` preprocessor command allows one to define a name *name* as a macro for an expression, the macro body. When it processes this command, the preprocessor replaces all occurrences of *name* by the macro body, till the preprocessor command `#undef` *name* is found, if ever.

The program predefined_macros_and_identifier.c shows that a few predefined macros are ready for use, together with some predefined identifiers.

A macro can be redefined, whether it has been previously undefined or not. The preprocessor replaces any occurrence of a macro that has not been defined or that has been undefined by the empty expression.

A macro can accept arguments, in which case the actual arguments following the macro name are substituted for formal parameters in the macro body.

The #define and #undef commands (2)

The preprocessor expands macro names iteratively, allowing for a macro's body to itself contain a macro, as illustrated in macro_expansion.c.

There is no risk of infinite expansion as macros are not recursively expanded, as illustrated in non_recursive_macro_expansion.c.

Macro bodies are often surrounded by parentheses, as well as macro arguments, to make sure that the intended meaning of the macro is preserved in any context after macro expansion, as illustrated in parentheses_in_macro_definition.c.

A macro can have as its last or only formal parameter an ellipsis to indicate a variable number of arguments, which in the macro body are referred to by the identifier `__VA_ARGS__`; the preprocessor replaces `__VA_ARGS__` with the (possibly empty) sequence of arguments after the designated arguments, including their comma separators. This is illustrated in variable_arg_list_in_macro_definition.c.

The #if, #else, #endif, #ifdef and #ifndef commands (1)

Lines of source text can be included or excluded from the compilation depending on the value of constant expressions that evaluate to a constant arithmetic value, not to be equal to 0 iff the source text is to be included.

The syntax is of the form

```
#if constant_expression_1
    source_code_1
#elif constant_expression_2
    source_code_2
...
#elif constant_expression_n
    source_code_n
#else
    source_code_{n+1}
#endif
```

where the `#elif` and `#else` preprocessor commands are optional.

The #if, #else, #endif, #ifdef and #ifndef commands (2)

`#ifdef` *`macro`*, `#if defined` *`macro`* and `#if defined (`*`macro`*`)` offer three alternatives to the `#if` *`constant_expression`* command, with *`constant_expression`* evaluating to `1` in case *`macro`* is being defined, and to `0` otherwise.

`#ifndef` *`macro`* offers an alternative to the `#if` *`constant_expression`* command, with *`constant_expression`* evaluating to `1` in case *`macro`* has not been defined or is being undefined, and to `0` otherwise.

The use of all these preprocessor commands is illustrated in conditional_compilation.c.

The #error command

A line of the form

```
#error sequence_of_tokens
```

causes *sequence_of_tokens* to be output as a compile-time error message, as illustrated in error_directive.c.

The stringization and merging operators, # and

In a macro definition, # can occur, followed by the name of a formal parameter *parameter*. When the macro is expanded, #*parameter* is replaced by the actual argument enclosed in double quotes, with each sequence of whitespace in the actual argument being replaced by a single space.

In a macro definition, ## can occur, preceded and followed by tokens. When the macro is expanded, the tokens on both sides of ## are combined into a single token. One token or both tokens can be formal parameters to the macro, that will, as usual, be replaced by an actual argument or two actual arguments, respectively, during macro expansion.

The use of both operators is illustrated in number_sign_in_macro_definition.c.