Preprocessor Directives

A directive is a command carried out by the preprocessor. A preprocessor decides which lines to compile, it also carries out a search-and-replace before the source is complied.

`#include` directive is the most essential - it brings in a set of declarations from another file.

General Syntax

Typically directives should be placed on their own line and begin with a `#` sign.

Summary

- `#define symbol` creates a symbolic name and assigns an empty string to it useful in conjunction with the `#ifdef` directive
- `#define symbol value` creates a symbolic name and assigns replacement text to it.
- `#define symbol(args) value` creates a macro function
- `#elif condition` starts a conditional compilation block, should follow another `#elif`, `#if` or `#ifdef`
- `#endif` ends a conditional compilation block
- `#error message` ends compilation immediately and prints the message
- `#if condition` begins a conditional compilation block
- `#ifdef symbol` begins conditional block, checks to see if specified symbol is defined
- `#ifndef symbol` begins conditional block, checks to see if specified symbol is not defined
- `#include <filename>` read contents of *file_name*
- `#include "filename"` read contents of project own header files
- `#line line_number` sets next line of code to *line_number* used for reporting purposes
- `#line line_number file_name` sets current *file_name* and *line_number* to the values indicated
- `#pragma command_text` responds to compiler-vendor-specific directive
- `#undef symbol`removes a symbol declaration

Using Directives to Solve Specific Problems

Creating Symbols

`#define` keyword useful create symbols used throughout a program

Creating Macros with `#define`

Macros are a convenience but have some drawbacks - noting they carry out search-and-replace during preprocessing - using templates or functions might

be more efficient

```
#define MAX(A, B) (A > B ? A : B)
```

Conditional Compilation

Conditional compilation relies on `#if` and its variations. For example changes for specific version can be isolated and switched on/off.

```
#define VERSION 3.0

#if VERSION > 2.5
// ...
#endif
```

Preprocessor Operators

- `#macro_arg` strigifies an argument
- `token1##token2` concatenates two tokens
- `defined(args)` evaluates to true/false if the symbol has been defined

Predefined Macros

C++ preprocessor provides some predefined macros. Mainly used for printing diagnostic info during runtime.

- `assert(statement)` terminates program if statement is false, part of `<cassert>`
- `static_assert(statement, fail_msg)` similar to assert by requires no header file. Reports an error at compile time.
- `NDEBUG` turns off debugging behaviour, disabling use of `assert` macros
- `__DATE__` produces 11 char date string
- `__FILE__` produces string containing the file name
- `__LINE__` produces string containing the current lint number
- `__STDC__` defined if complier supports standard C only
- `__TIME__` produces 8 char time
- `__cplusplus` produces symbol if the C++ language is supported by the compiler

Creating Project Header Files

Header files are intended to be included by every individual source. `#define "my_proj.h"`. Should include

- prototype of each and every function intended to be shared by all modules
- `extern` declaration for all global data
- Class declarations for the project
- `enum` and `typedef` declarations

- `#define` directives

Header files should not include executable code only declarations.

```
// funciton prototypes
void do_stuff();
...

// external variables, not actually created
extern double time_left;
...
```