P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #14

C–style Preprocessing

Directives, Pragmas, Conditional Compilation, Macros

Overview

- Some Application Scenarios
- A little Terminology
- Approaches to Preprocessing in Fortran
- C–style Preprocessing

Scenario #1: Portability/Experimenting

- Take the neotectonic simulation code SHELLS
  - In each iteration it solves a large, sparse linear system of equations using some LAPACK routine.
  - Assume we want to replace this by some specialised sparse direct solver like PARDISO from the MKL
- Problem:
  - We want to stay portable,
  - but MKL will not be available everywhere
- How to handle this?

Scenario #1 (cont.)

- One possibility:
  - Put everything related to LAPACK into a file `lapack.f` and everything related to PARDISO into `pardiso.f`.
  - In both cases have common driver/wrapper functions, e.g. `LINSYS_SETUP()` and `LINSYS_SOLVE()`
  - Depending on system (availability of MKL) edit makefile to compile and link either `lapack.f` or `pardiso.f`
- Another approach:
  - Mark specialised source code parts with preprocessor directives
  - Define macro while compiling `ifort -DWITH_MKL` to select MKL and PARDISO solver
  - This is called conditional compilation

Scenario #2: Debugging

- During program development we often like to insert specialised code sections for debugging, such as e.g.
  - print statements for reporting key values
  - assertions that check assumptions on values and indices
  - ...
- Sometimes we would like to keep these (especially sophisticated assert checks) for future debugging.
- However, we do not want to slow down production code!
- Again this can be done with conditional compilation.

Scenario #3: Global Settings

- Sometimes we want to use symbolic constants during compilation that should be identical in all sub-programs
  - Think of the special number π,
  - or the discretisation parameters `nt`, `mt` and `nr` in Terra.
- For FORTRAN77 we
  - wouldn't want to pass these around as variables via sub-program interfaces or common blocks
  - so Terra `includes` a file `size.h` where necessary
- Preprocessing allows to handle this via macro definitions
- Also important for architecture dependent features (e.g. array padding)

Overview

- Some Application Scenarios
- A little Terminology
- Approaches to Preprocessing in Fortran
- C–style Preprocessing

Definitions

Preprocessor

A preprocessor is a computer program that takes an input text (often source code) and converts it to an output text.

- typically conversion involves textual substitutions and macro expansions
- but might also include conditional actions $\longrightarrow$ conditional compilation

Definitions (cont.)

Macro

A macro is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure.

Macro Expansion

The mapping process that instantiates a macro into a specific output sequence is known as macro expansion.

Examples

- under Unix/Linux the m4 macro preprocessor is typically available
  - processing the input file

```
define(AUTHOR, William Shakespeare)dnl
A Midsummer Night's Dream
by AUTHOR
```

  - with m4 yields

```
A Midsummer Night's Dream
by William Shakespeare
```

- anybody who used \newcommand in LaTeX defined a macro and relied on preprocessing
- any C/C++ programmer using an #include directive employs preprocessing

Fine Details

Lexical Preprocessor

Lexical Preprocessor perform simple substitutions on the source code prior to any parsing; requires no understanding of language itself.

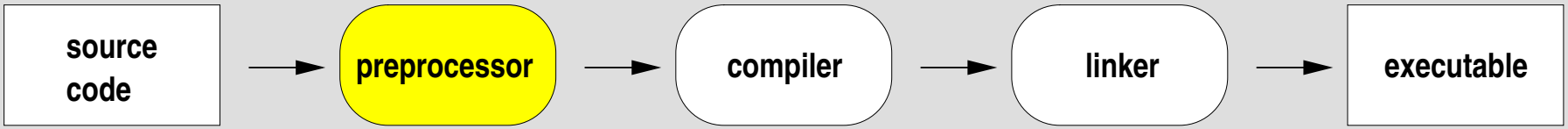
Syntactic Preprocessor / Precompiler

Precompilers are used to perform syntactical transformations before actual parsing; requires syntactical analysis and understanding of language constructs.

- The C–preprocessor and the general-purpose m4 preprocessor are lexical preprocessors.
- Precompilers are often used to extend a given language with new constructs (see e.g. embedded SQL)

Compilation Stages

- Preprocessing adds a third stage to the build process, i.e.
  - Before compilation source code is passed to a preprocessor.
  - The preprocessor performs (purely) textual changes to the source code based on the directives.
  - Resulting code is passed on to compiler.



Pragmas

Pragma

A pragma is a directive intended for the compiler providing some practical information.

- Pragmas are not intended for the preprocessor.
- Term is common in languages like e.g. C, C++ or Ada.
- An example from HPC is thread-based parallelisation using OpenMP compiler pragmas
  - C/C++: `#pragma omp parallel`
  - Fortran: `!$OMP PARALLEL`
- C languange standard allows `#pragma` preprocessing directive; behaviour (what is supported and done) is compiler dependent.

Overview

- Some Application Scenarios
- A little Terminology
- Approaches to Preprocessing in Fortran
- C–style Preprocessing

Preprocessing

The preprocessor handles the first four (of eight) phases of translation specified in the C Standard:

1. Trigraph Replacement – The preprocessor replaces trigraph sequences with the characters they represent.
2. Line Splicing – Physical source lines that are continued with escaped newline sequences are spliced to form logical lines.
3. Tokenization – The preprocessor breaks the result into preprocessing tokens and whitespace. It replaces comments with whitespace.
4. Macro Expansion and Directive Handling – Preprocessing directive lines, including file inclusion and conditional compilation, are executed.

(Wikipedia)

C–style Preprocessing

- Preprocessing is an integral part of the C language standard.
- C's preprocessing language/directives
  - are independent of the C language itself
  - can be used for other applications, too.
- Most Fortran compilers understand and preprocess C-style directives.

Directives: Syntax

- Preprocessing directives always start with the # symbol
- General format is `#<directive name> <text>`
- Standard allows for white-space in front of the #
  - no good style
  - not always supported (e.g. not by ifort/fpp)
- Directive may be followed by comment in same line
- Multi-line directives are possible using the continuation symbol \.

Directives: Overview

directive	purpose
`#include`	insert contents of another file
`#define`	define a (parameterised) macro
`#undef`	undefine a macro
`#warning`	issue a warning message (common non-standard extension)
`#error`	issue message and stop compilation
`#line`	related to line number generation for compiler messages

C preprocessor directives not related to conditional compilation

Directives: Overview (cont.)

- for conditional compilation there is an *if–then-else* construct:

```
#if <condition>
#elif <condition>
#else
#endif
```

- `<condition>` must be a constant expression
- if its value = 0 `<condition>` is false, otherwise true
- preprocessor will remove all code not in the valid branch

Directives: Overview (cont.)

- the operator `defined` allows to test, whether a certain macro is defined (value unimportant)
- can be abbreviated as

```
#ifdef <name>   ⟷  #if defined(<name>)
#ifndef <name>  ⟷  #if !defined(<name>)
```

- by defining a certain macro we may select/de-select certain parts of the source code

Example 1: Conditional Compilation

- Define π if not already defined

```
#ifndef PI
#define PI 3.14159265358979
#endif
```

- Avoid multiple type definitions from inclusion of same header file (top of file `mytypes.h`)

```
#ifndef MY_TYPES_INCLUDED
#define MY_TYPES_INCLUDED
typedef ...
#endif
```

Example 2: File Inclusion

- Consider a file `hello77.f` with contents

```
      PROGRAM HELLO
#include "inc.h"
      END
```

- and a file `inc.h` containing

```
      PRINT *,'Hello World!'
```

- Executing `ifort -fpp -save-temps hello77.f` results in an executable
- and a file `hello77.i` with the preprocessed source code

```
# 1 "hello77.f"
      PROGRAM HELLO
# 1 "./inc.h" 1
      PRINT *,'Hello World!'
# 3 "hello77.f" 2
      END
```

Macros

- C allows parameterised and non-parameterised macros.
- A non-parameterised macro gets defined by

  ```
  #define <macro name>
  ```

  or

  ```
  #define <macro name> <value>
  ```

- During macro expansion preprecessor replaces each occurence of `<macro name>` in the source code by `<value>`.
- Valueless macros are legal; typically used as flags for conditional compilation.

Example 3: Non-parameterised Macros

- The file `limits.h` contains macros showing limits of ranges of integral types, e.g. `INT_MIN`. The line

  ```
  printf( "INT_MAX = %d\n", INT_MAX );
  ```

  will report that value to stdout.

  Note that `INT_MAX` does not get replaced within string constant!

- Function `int rand(void)` returns an integer between 0 and `RAND_MAX`, so

  ```
  j = 1 + (int) (10.0 * (rand() / (RAND_MAX + 1.0)));
  ```

  gives a random value $j \in \{1, 2, \ldots, 10\}$.

  (Caveat: Random number generation is a highly complex business. See e.g. Numerical Recipes, 3rd ed. for an excellent discussion of practical issues.)

Parameterised Macros

- Parameterised macros accept values to be included into the replacement text.
- They are defined by

```
#define <macro name> (parameters) <replacement text>
```

- Simple example

```
#define SQR(a) (a)*(a)
```

- Note the round brackets for avoiding side effects! Without them `SQR(x+y)` expands to `x+y*x+y` instead of `(x+y)*(x+y)` !

Example 4: Parameterised Macros

- Parameterised macros share some similarities with arithmetic statement functions in Fortran.
- Can be used to guarantee inlining of codelets!
- The following macro takes care of interpreting entries in Fortran array

  ```
  REAL mat(2,3)
  ```

  correctly when passed as 1D array `fmat` to C

  ```
  /* Macro for computing index into matrix */
  #define IDX(i,j) ( ((j) - 1) * 2 + (i) - 1 )
  ...
  cmat[i][j] = fmat[IDX(i,j)];
  ```

Pre-Defined Macros

- Compilers automatically set pre-defined macros like
  - type of operating system on which we compile
  - info on compiler and compiler version
  - info on language standard for which we compile
  - ...
- This is important for a language that is close to the OS, since things can be very different between e.g. Linux and Windows

```
#if defined(__LINUX__)
...
#elif defined(__WINXP__)
...
#endif
```

Example 5: Error Localisation

- The automatic macros `__LINE__` and `__FILE__` can e.g. be used to help locate run-time errors

```
void myFunc( int myParam ) {
  /* Check if value is positive */
  if ( myParam <= 0 ) {
    printf( "ERROR at line %d in file '%s'!!\n",
            __LINE__, __FILE__ );
  }
}
```

- Calling the routine as `myFunc(-2)` results in

```
ERROR at line 9 in file 'error.c'!!
```