Informatics

Ingeniería en Electrónica y Automática Industrial

The Preprocessor

The preprocessor in C language

- Definition
- Preprocessor directives
- `#include.` Header files
- `#define`
  - Symbolic constants
  - Macros
- Conditional compilation directives
- Other directives

Definition

- The ***preprocessor*** is a text processor that performs operations on the source code.
  - It is a separate first step in compilation
  - Basically it is the inclusion in the main source code of **header files**, **macros** and **conditional compilation**
  - The preprocessor instructions are called ***directives***
  - The preprocessor main goal is to facilitate programming

Preprocessor directives (I)

- They are ***special instructions*** that are **processed before the actual compilation** which produce the final machine code
  - They are not regular C statements, so in the source code:
    - They are preceded by the symbol «#»
    - No «;» expected at the end
  - A standard set is included in ANSI C. Compilers usually include others
  - By default they just occupy one line. To continue in the next one the symbol «\» must be used
  - They can be in any part of the source code but their effect is just from the line where they are placed onwards.

Preprocessor directives (II)

- The directives included in ANSI C are

`#include`	`#define`
`#error`	`#if`
`#elif`	`#else`
`#ifdef`	`#ifndef`
`#endif`	`#undef`
`#line`	`#pragma`

`#include`. Header files

- Makes the preprocessor to substitute the directive by the ***header file*** in the point where the directive is

  `#include "headerfile.h".` Preprocessor looks for the file first in the program directory and later in the system ones (mainly for user files)

  `#include <headerfile.h>.` Preprocessor looks directly in the system directories (for standard libraries)

- Typically header files collect information that is used by different source files, as:
  - Macros and constants definitions, global variables, function declarations….

`#define`. Symbolic constants

```
#define IDENTIFIER string
```

- The preprocessor will substitute any occurrence of `IDENTIFIER` in the source code by *`string`*
  - *`string`* can be a
    - Symbolic constant
    - Macro (optional parameters)
  - To distinguish `IDENTIFIER` from variables use CAPITALS
  - Definitions can use previous definitions
- For **symbolic constants**:

```
#define PI 3.141516
#define MEMERR "Error in Memory Allocation"
```

`#define`. Macros (I)

`#define MACRONAME(parameters)` *`expression`*

- `MACRONAME` is the identifier (in CAPITALS)
- `parameters` are arguments separated by commas to be substituted when the identifier occurs in the code
- *`expression`* is any valid expression that operates with the `parameters`
- When the preprocessor finds a call to `MACRONAME` in the source code will substitute it for *`expression`* changing `parameters` by their values contained in the call.

`#define`. Macros (II)

- Example. Macro to obtain the greater of two numbers

```
#define MAX(a,b) ((a)>(b)) ? (a) : (b)
...
x = MAX(dat1, dat2);
```

  - It looks similar to a function but it is just a substitution:
    `x = ((dat1)>(dat2)) ? (dat1) : (dat2)`

- **Macros vs functions**
  - Macros generate longer code but are faster (no function call)
  - Macros can give rise easily to errors difficult to debug (always **use parenthesis** in *`expression`*)
  - Some standard *functions* are macros (`getc(), gerchar()`)
  - In general
    - Use **macros for small and easy code** that appears many times
    - Use functions for larger code

`#define`. Macros (III)

- In ANSI C there are five useful **predefined macros:**
  - `__LINE__` Writes the code line number when compiling:
    ```
    int nline = __LINE__
    ```
  - `__FILE__` Writes the name of the source code:
    ```
    printf("%s\n", __FILE_);
    ```
  - `__DATE__` Writes the date of compilation (mm dd yyyy)
    ```
    printf("%s\n", __DATE__);
    ```
  - `__TIME__` Writes the time of compilation (hh:mm:ss)
    ```
    printf("%s\n", __TIME__);
    ```
  - `__STDC__` Is substituted by 1 if all code is ANSI standard
    ```
    int ansi = __STDC__
    ```

Conditional compilation directives (I)

- The **conditional compilation directives** allow for selective compilation of parts of the source code:
  - Facilitate debugging (debug with value-check, write, etc..)
  - Make possible to personalize programs (eg. compile for different platforms)
- Types:
  - Compilation conditioned by the **value of an expression**:

    `#if` `#elif` `#else` `#endif`
  - Compilation conditioned by the definition of a **macro**

    `#ifdef` `#ifndef` `#endif`

Conditional compilation directives (II)

- Compilation conditioned by the **value of an expression**

```
#if constantexpression1
      statements1;
#elif constantexpression2
      statements2;
#elif constantexpression2
      .....
#elif constantexpressionN
      statementsN;
#else
      statementsM;
#endif
```

Conditional compilation directives (III)

- The terms *`constantexpressionX`* are evaluated in compilation time:
  - They can include logical and relational operations
  - They cannot include program variables
- `statementX` represent C code lines
- `#elif` is equivalent to `#else #if`
- `#else` and `#elif` are associated to the nearest up `#if` and are optional.

Conditional compilation directives (IV)

- Example. Program to include a different header file depending on the connected printer in that moment

```
#if DEVICE == IBM
      #include ibmdrv.h
#elif DEVICE == HP
      #include hpdrv.h
#else
      #include gendrv.h
#endif
```

Conditional compilation directives (IV)

- Compilation conditioned by the definition of a **macro**

```
#ifdef MACRONAME1
      statements1;
#endif
#ifndef MACRONAME2
      statements2;
#endif
```

  - `statements1` are compiled just if *`MACRONAME1`* is previously defined
  - `statements2` are compiled just if *`MACRONAME2`* is NOT previously C
  - `#else` can be combined with them but not `#elif`

Other directives

- **`#undef`.** Eliminates the definition of a macro or symbolic constant `#undef` *`MACRONAME`*
- **`#error`.** Stops compilation showing a message on screen:
  `#error` *`Message`*
- **`#line`.** Changes the value of predefined macros `__LINE__` and `__FILE__`

  `#line` *`linename`* "*`newfilename`*"
- **`#pragma`.** To access compiler-specific preprocessor extension; ie each pragma directive has different sintax, implementation rule and use

  `#pragma` *`compilerspecificextension`*