The C Pre–Processor

ECE2036

December 4, 2012

The C Pre–Processor

1. The C *pre–processor* is the very first step in any C or C++ program compilation.
2. It is a very simple program that makes the compiler's job easier.
3. It processes *Directives*, *Identifier Definitions*, and *Macros*.
4. All pre–processor commands start with the hash symbol `#`
5. It strips out all comments from the source code.
6. The output of the pre–processor is then the input to the compiler.
7. The pre–processor program `cpp` can be called directly if desired, and is in fact often useful.

The #include Directive

1. We have already seen and used the `#include` directive in the lab assignments.
2. It simply tells the pre-processor to locate the specified file, and insert all of the code in that file directly into the file being compiled.
3. This is used primarily to get function prototypes for the various C and C++ run–time library functions.
   - `printf`
   - `sin`
   - `drand48`
4. Include files can, and very often do, include other files.
5. If this is done, we must be careful to avoid infinite recursion
6. We specify the name of the file to be include enclosed in either angle brackets `< >` or in double quotes.
   - When the angle brackets are used, this indicates the include file is provided by the compiler or operating system, and the pre–processor will search for the file in the “Usual Places”, depending on where the compiler is installed.
     - `#include <stdio.h>`
     - `#include <math.h>`
   - When the double quotes are used, this indicates the include file is provided by the programmer. In this case the search path to find the file is the current directory, or any directories specified in the `-I` compiler directive.
     - `#include "cs1372.h"`
     - `#include "gthread.h"`

Pre–processor Identifier Definitions

1. Another common use of the pre–processor is defining *Identifiers*
2. An identifier is defined using the `#define` pre–processor directive.
3. An identifier in this context simply means an text string substitution.
   - `#define TWO 2`
   - `int i = TWO;`
   - `#define MAXFLOAT 3.40282346638528860e+38`
   - `#define M_PI 3.14159265358979323846264338`
   - `float radians = (degrees/360.0) * 2 * M_PI;`
4. Many identifiers are defined by the system provide include files, such as those from `math.h` shown above.
5. Also, it is very common for your programs to define and use this type of identifier.

Conditional Compilation

1. The pre–processor directive `#ifdef MY_IDENT` tells the pre–processor to only include the following lines of code if the identifier `MY_IDENT` is defined.
2. The conditional compilation sequence is terminated by the directive `#endif`
   - `#ifdef LINUX`
   - `// Some linux specific code here.`
   - `#endif`
   - `#ifdef WIN32`
   - `// Some windows 32-bit specific code here.`
   - `#endif`
3. There is an equivalent *if not defined* directive:
   - `#ifndef TABLE_SIZE`
   - `#define TABLE_SIZE 100`
   - `#endif`
   - `int table[TABLE_SIZE]`
4. There is also an `#else` directive that reverses the result of the prior conditional:
   - `#ifdef USING_GRAPHICS`
   - `// Code for graphical output here`
   - `#else`
   - `// Code for non-graphical output here`
   - `#endif`

Conditional Compilation (continued)

1. The prior examples showed testing whether or not an indentifier is defined to control the conditional compilation.
2. The *value* of an identifier can also be tested, giving quite a bit more flexibility.
3. The following code is directly out of `math.h`

```
#if __FLT_EVAL_METHOD__ == 0
typedef float float_t;
typedef double double_t;
#elif __FLT_EVAL_METHOD__ == 1
typedef double float_t;
typedef double double_t;
#elif __FLT_EVAL_METHOD__ == 2
typedef long double float_t;
typedef long double double_t;
#else
#error "Unsupported value of __FLT_EVAL_METHOD__."
#endif
```

4. The above code snippet shows the use of the `#error` directive and the `#elif` (else - if) directive.

Some Useful Pre–Defined Identifiers

1. There are several identifiers that are pre–defined and managed by the pre–processor that are quite useful.
   - `__LINE__` is defined to be the current line number in the file being compiled.
   - `__FILE__` is defined to be the name of the file being compiled.
   - `__DATE__` is defined to be the date the program is being compiled.
   - `__TIME__` is defined to be the time the program is being compiled.
2. Here is an example of a program using the above identifiers.

```
int main()
{
  cout << "This is the line number " << __LINE__;
  cout << " of file " << __FILE__ << ".\n";
  cout << "Its compilation began " << __DATE__;
  cout << " at " << __TIME__ << ".\n";
}
```

Include Guards

1. As mentioned previously, a file being included (with `#include` can and often does include other files.
2. Care must be taken to prevent infinite recursion.
3. This is commonly done with *include guards*.
4. It is simply an indentifer that is defined by the include file, and uses an `#ifdef` to prevent a recursive inclusion of the same file.
5. Here is an example of how this might be used

```
// This is include file tcp.h
#ifndef __TCP_H__
// This part is compiled only if the above
// symbol is not yet defined If not, define
// the symbol to prevent the recursing
#define __TCP_H__
// The remainder of tcp.h here
#endif
```

Pre–processor Macros

1. Another extremely useful of the pre–processor is the definition and use of *macros*
2. A macro is nearly identical to an identifier, and is defined in the same way.
3. However, a macro has *arguments*
4. Remember however that all of the pre–processor directives and processing is done at compile time, before the actual compiler is called.
5. Macro arguments do not have types or values. They are just a string of characters that is substituted for the paramenter name.
6. Suppose we have a one–dimensional array that represents the data in a two–dimensional structure of width w, (such as pixels on the display screen)
7. It is convenient to define a macro that calculates the correct index for a given x and y coordinate.
8. The following is a flawed attempt at doing this. The reason for the flaw will become apparent.
   - `// Define a macro to compute the array index`
   - `// for the given x, y position in a 2-D`
   - `// structure of width w.`
   - `#define INDEX(x,y,w) y * w + x`
9. We can then "call" the macro by simply typing the macro name and providing the three arguments
   - `int i = INDEX(10, 20, 30);`

Macros, Continued

1. The expansion of a macro is simply a text substitution, substuting the value of the argument specified for the text string of the parameter name.
2. In our previous example, the expansion of the `INDEX` macro is:
   - `int i = 20 * 30 + 10;`
3. However, our macro definition is incorrect, as shown by the following' example:
   - `int i = INDEX(x-1, y-1, 20);`
4. This will expand to:
   - `int i = y-1 * 30 + x-1;`
5. The above expansion does not produce the desired results due to the precedence of operators.
6. The correct defininition of the `INDEX` macro uses parenthesis in the definition to prevent the above problem.
   - `#define INDEX(x,y,w) (y) * (w) + (x)`
7. This results in the following, correct, expansion:
   - `int i = (y-1) * (30) + (x-1);`