This course has already ended.

« 10. Advanced Features Course materials 2 Variable length argument lists »

ELEC-A7100 / 10. Advanced Features / 1 Preprocessor

Tenth round tasks as a `ZIP file.`

Preprocessor¶

The preprocessor processes the C code before passing it to the actual C compiler that produces the binary object code. The preprocessor, for example, removes the comments from the code and executes preprocessor directives, such as **#include** that includes definitions from another header file as part of the compilations. The output of preprocessor is still text-format source code. The preprocessor output can be checked using the **gcc** compiler with the **-E** flag.

In the following we take a look at some of the most common preprocessor directives.

Basics¶

The preprocessor instructions begin with a hash character (#) and usually contain some parameters. So far we have seen mainly one preprocessor instruction, **#include**, that fetches another file as part of the C source file. In principle, `#include` could be used with any other file, but they are supposed to be used with header files that contain only definitions of data types, constants and function prototypes, and do not produce program code themselves. After the preprocessor phase, all preprocessor directives have been replaced by C code that can be compiled by the actual compiler.

A preprocessor directive begins from the start of the hash-marked line and ends at the end of line. Each instruction takes exactly one line, and there is no trailing semicolon as in normal C statements. However, for long instructions, a line can be split with backslash (`\`) character at the end of the line. This means that the preprocessor directive continues on the next line

Constants and Macros¶

One of the most common preprocessor directives is the `#define` directive that defines a constant that will be replaced in source code with the text given as part of the `#define` instruction.

The format of the #define declaration is:

```
#define NAME some text
```

Every *NAME* in the program code will be now replaced with the given text during the precompilation. It is worth noting that the precompiler handles text-based content and "some text" can be any sequence of characters, numbers or statement. Precompiler doesn't have any syntax checks and can thus produce sequences that are completely incomprehensible to the C compiler. This will though lead to a failure during compilation. Thus, *#define* can be used to write code that is really difficult to comprihend while it may still work. As an example, say, a chess game, which won the "Internatinal Obfuscated C Code" competition a few years ago.

Here is an example of simple *#define* in the following program:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAXSTRING 80
5
6  int main(void) {
7      char str[MAXSTRING];  // allocate memory for 80 characters
8
9      // copy 79 characters max
10     strncpy(str, "string", MAXSTRING - 1);
11     printf("str: %s\n", str);
12 }
```

In the above example, preprocessor replaces the MAXSTRING labels on line 6 and 7 by number 80, before passing the code to the actual C compiler. It is a common convention (but not mandatory) that constants and macros defined using #define use upper case names, to distinguish them from other variables in the code. It is a fairly common practice to use capital letters to define macros though, it is not mandatory to do. In some other case it would be possible that a defined macro is transformed into a string, or a small piece of C code. The preprocessor just does the text replacement, and is not concerned with the types of the variables.

Alternatively, in the above case MAXSTRING could have been defined as constant global variable: `const int MAXSTRING = 80;`. The difference is, that then the type of the value is clearly defined, and the operation is done by the C compiler, not by the preprocessor).

The *#define* declaration can be removed by *#undef* NAME. Following the #undef declaration, the NAME replacement cannot be used in code.

#define macros can also contain parameters. When such macro is used and expanded in code, the parameters are used as part of the expanded code.

For example, we could have the following **CHECK** macro definition:

```
1  #include <stdio.h>
2  #define CHECK(cond, msg) if (!(cond)) { printf("%s", msg); }
3
4  int main(void)
5  {
6      CHECK(5 > 10, "5 > 10 failed\n");
7      CHECK(10 > 5, "Something strange happened\n");
8  }
```

CHECK macro will be therefore replaced by an if-else statement and a print statement which will be print only if the condition is false.

After precompilation, main function lines would look like this:

```
if(!(5 > 10)) { printf("%s", "5 > 10 failed\n"); }
```

and

```
if(!(10 > 5)) { printf("%s", "Something strange happened\n"); }
```

Other features¶

The preprocessor supports conditional statement `#if` that contains a section of code until `#endif`. The if conditions and logical operations work as normally in C. In addition, there is `#elif` declaration for "else if", and `#else`. The behavior of these conditions is much like before with normal C conditional statements, but these are evaluated in preprocessing phase, and not visible during actual compilation.

Below is an example of using these if condition directives.

```
1  #if (VERSION == 1)
2      #include "hdr_ver1.h"
3  #elif (VERSION == 2)
4      #include "hdr_ver2.h"
5  #else
6      #error "Unknown version"
7  #endif
```

The #error declaration shown above raises a (compile) error with given message, and the compilation fails at this point. Note that the error is a compile-time error, and the conditions are evaluated before compilation. If we happened to have the right version above, the error will never appear in compiled code.

#define declarations for a name can also be given without a value, just to tell the preprocessor that a particular condition exists. This is commonly used with include guards. The purpose of include guard is to ensure that a particular C source file does not include the same header definitions multiple times, which would cause compile errors. This can sometimes happen, when there are nested include dependencies between multiple header files. #ifdef declaration can be used to test whether a particular name has been defined, regardless of its value.

#ifndef is for the opposite test, and is true if a name has not been declared.

Here is an example:

```
1  #ifndef SOME_HEADER_H  // at the beginning of file
2  #define SOME_HEADER_H
3
4  // some header content
5
6  #endif // at the end of the file
```

The above #ifndef condition is true for the first time a particular header file is included as part of the C source (by #include directive). If the same header is included another time, SOME_HEADER_H is already defined, and the header content is not re-evaluated. In large software projects is not unusual that a ` header is included multiple times, because there can be nested header definitions that cause complicated dependencies between them.

The preprocessor also has some readily defined macros that can be used in the C code. These can be especially useful for debugging purposes:

- `__DATE__` is substituted with the current (compile time) date. This will be evaluated at compile time: if compilation is successful, the date will not change before the next time the program is recompiled.
- `__TIME__` is substituted with the compile time time, with behavior as above.
- `__FILE__` is substituted with the name of the C source file where the macro is located. This could be used, for example, in implementing a common debugging macro (such as assert).
- `__LINE__` is substituted with the line number of the location of the macro. Again, this is useful in conjunction with some debugging macro.

A simple test program could use these in the following way:

```
1  #include <stdio.h>
2
3  #ifdef DEBUG
4  #define MYDEBUG(Msg) fprintf(stderr, "File: %s, Line: %d: %s", \
5                          __FILE__, __LINE__, Msg)
6  #else
7  #define MYDEBUG(Msg)
8  #endif
9
10 int main(void) {
11     MYDEBUG("Starting\n");
12     for (int a = 0; a < 10; ) { a++; }
13     MYDEBUG("At the end\n");
14 }
```

Try to execute the above program. The **FILE** and **LINE** will not be printed as DEBUG macro is not defined anywhere. Copy the program to your machine and try to compile it with the following command gcc -DDEBUG testi.c.

The following lines must be printed on the screen.

```
File: testi.c, Line: 11: Starting
File: testi.c, Line: 13: At the end
```

Task 10.1: Macros¶

Objective: Practice the use of parametrized macros.

This exercise does not contain other *.c files than **main.c**. Instead, the relevant code you'll need to implement is in **macros.h** header, where you need to place the following two macros:

Exercise (a): EQ3(a,b,c) takes three parameters and evaluates their equality. Evaluates to 1 if all parameters are equal (==) to each other. Evaluates to 0 otherwise. May evaluate any parameter more than once.

Exercise (b): MIN3(a,b,c) that evaluates which one of the parameters is smallest. Returns the smallest one.

Hint: Use the *ternary* operator instead of if-else.

Points 20 / 20 My submissions 1 Deadline Friday, 13 August 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Macros

Select your files for grading

macros.h

Choose File No file chosen

Submit