CST8152 – Compilers
Article #3
The C Language Preprocessor

1. File inclusion

   #include <*filename*> /* standard libraries header files */
   #include "*filename*" /* user defined header files */

2. Macro Definitions and Expansions (Macro Substitutions)

   #define *NAME* /* defines name */
   #undef *NAME* /* undefines name */
   #define *NAME Replacemen*t /* replaces NAME with replacement */
   #define *name(list) expression* /* inline functions or macro */

3. Conditional processing (compilation)

   #if *constant-expression*
   #ifdef *NAME*
   #if defined(*NAME*)
   #ifndef *NAME*
   #if !defined(*NAME*)

   #elif *constant-expression* /* optional */
   #else /* optional */
   #endif /* must be present */

4. Pragmas (Pragmatics)

   #pragma *directive* /* sets different compiler options */

5. Other
   # /* null directive – no any action or "" enclosure */
   ## /* token passing – concatenation */
   #line *constant* /* changes the line number of the source */
   #error *text* /* compiler displays error message including text */

6. Predefined macros in ANSI C:
   __LINE__ , __FILE__ , __TIME__ , __DATE__ , __STDC__

Examples

1. File inclusion

```
#include <limits.h>
```

The header file must be in the *include* directory of the compiler

```
#include "buffer.h"
```

The header file must be in the same directory as the source file containing the `include` preprocessor directive

2. Protecting header files from multiple inclusions

```
#ifndef BUFFER_H_
#define BUFFER_H_

… declaration

#endif
```

When the header file is included for the first time BUFFER_H_ is not defined and the contents of the file (the declarations) is included in the source file which contains the include statement. If the header file is included for the second time (this happens when you work with multiple source file programs), BUFFER_H_ has already been defined and the declarations will not be included. This prevents from duplicate declarations syntax errors.

3. Inserting debug statements in a program

The following line must be placed in your header file or on the top of your source file

```
#define DEBUG
```

If you want to print some debugging information in your program, you include the following lines

```
#ifdef DEBUG
  printf("The size is negative: %d", size)
#endif
```

When the testing of your program is finished you must the define statement from the header or place somewhere under the define

```
#undef DEBUG
```

The #undef directive is ofen used to suppress macros when a routine or procedure is implemented both as a macro and as a function.

```
#undef getchar
int getchar(void){…}
```

4. Defining constants in C

```
#define FAIL -1
```

The word `FAIL` will be replaced with `-1` everywhere in the program starting from the line following the define directive to the end of the source file.
Warning: If `FAIL` is enclosed in double quotes it will not be replaced. If, however, a parameter name is preceded by a # symbol in the replacement text, the combination will be replaced by the actual argument enclosed in quotes.

```
#define reprint(fexp) printf(#fexp " = %9.3f\n",fexp)
```

When the macro (see below) is invoked, as in

```
reprint(x+y);
```

it will be expanded to

```
printf("x + y" " = %9.3f\n",x+y);
```

5. Defining macros

It is possible to define macros, which resembles functions. These pseudo functions are truly generic because data of any type can be used with them as long as the macro contains the proper statement. If the #define value can not fit on one line and must be carried over another line, then the backslash (\) character should be the last thing on the line. For example:

```
#define WCALC(y,x) \
((y + x) / (x –y) * \
(INT_MAX – 77x))
```

Once defined, you can use them as normal functions.

```
#define sum(x,y) x+y

a = a + sum(b,c);
```

The preprocessor will replace the function-like call with the replacement text. The `sum()` in the line above will be replaced with `b+c`

```
a = a + b+c;
```

This looks like inline functions in C++, but it is not the same. When writing macros in C, you must remember that the preprocessor simply replaces the macro. This could lead to serious mistakes. For example, if you write the following

```
a = a * sum(b+c,d+e) / m;
```

The result from the substitution is

```
a = a * b+c + d+e / m;
```

As you see, this is wrong. To avoid these kind of mistakes always use parentheses

```
#define sum(x,y)  ((x) + (y))
```

Now the result is correct

```
a = a * ((x) + (y)) / m;
```

The directive ## can be used to create variable names. If the following is used:

```
#define STR(a,b)  a ## b
……
STR(str,1) = strcat(STR(str,2), STR(my,name));
```

The result after the preprocessor will be

```
str1 = strcat(str2,myname);
```

6. Including a proper file (header)

The following sequence will check the type of the application and will include the proper header file.

```
#define PLATFORM 10
…
#if  PLATFORM == __MSDOS__ * 10
#define HEADER  "dos.h"
#elif PLATFORM == __WIN32__ * 9
#define HEADER  "win.h"
#else
#define HEADER  "sys.h"
#endif
#include HEADER
```

Note: `__MSDOS__` and `__WIN32__` are Borland specific (non ANSI) manifest constants

7. Enforcing the same data type size across different platforms

```
#define BORLAND_16
#ifdef BORLAND_16
  typedef int  int_2b;
  typedef long long_4b;
#endif
#ifdef MSVS_16
  typedef short int_2b;
  typedef int   long_4b;
#endif
```

```
printf("Int size: %d Long size: %d\n",sizeof(int_2b),sizeof(long_4b));
```

8. Pragmas (Pragmatics)

```
#pragma inline
```

This line will force the compiler (Borland C/C++) to generate assembler output and call the assembler.

```
#pragma warning(disable : 4996)
```

This line will suppress the warning by the Visual Studio C/C++ compiler which is generated when a deprecated function is used in the code. For example, if the string.h declared functions are used in a C/C++ code compiled by MS Visual Studio, C4996 warning will be generated. C4996 is generated for the line on which the function is declared and for the line on which the function is used.

If a compiler does not recognize the `pragma` directive, it simply ignores it.
This convention allows different compilers to use different `pragma` directives.

If you want to learn more about the C Language, you can make a trip to:

http://www.lysator.liu.se/c/

You can learn there about different C Language Standards and much more.