Appendix D

Additional Preprocessor Topics

NOTE D.1

Preprocessor Conditional Inclusion Directives

The C/C++ preprocessor provides a mechanism for the conditional inclusion of selected segments of code into a program, permitting increased program versatility in areas such as:

- avoiding multiple inclusion of header file contents
- debugging
- testing macro values or testing for the presence/absence of macros
- writing portable "non-portable" code

The conditional inclusion directives are shown below and may be placed anywhere in a program, nested if desired. Each conditional inclusion block must begin with one of *#if, #ifdef,* or *#ifndef* and must end with *#endif*. *#endif* is required because braces have no special significance to the preprocessor.

- #if *constant-expression* // Is *constant-expression* true?
- #ifdef *identifier* // Is *identifier* currently defined as a macro name?
- #ifndef *identifier* // Is *identifier* currently not defined as a macro name?
- #elif *constant-expression* // Is *constant-expression* true? – Conditional alternative to any of above
- #else // Unconditional alternative to any of above
- #endif // End of conditional block

#if versus if

Aside from syntactical differences there is a major functional difference between runtime **if** statements and preprocessor conditional inclusion blocks. In the runtime **if** statement below the debugging code will always get compiled and become part of the executable program (thereby increasing its size) even if that code never gets executed when the program runs. In contrast, the debugging code in the preprocessor block below will be compiled and become part of the executable program only if the preprocessor finds *DEBUG* equal to 1. Otherwise, that code will not be included in the compilation, resulting in a smaller executable file.

```
if (debug == 1)
{
        Any kind of debugging...
        ...code goes here, including...
        ...preprocessor directives.
}
```

```
#if DEBUG == 1        /* parentheses optional for #if */
        Any kind of debugging...
        ...code goes here, including...
        ...preprocessor directives.
#endif
```

#if versus #ifdef and #ifndef

For debugging and other purposes a macro such as *DEBUG* in the second example above is sometimes defined without a replacement list, as in:

```
#define DEBUG
```

This form renders as meaningless any future comparisons of that macro with actual values. However, it can still be tested using the *#ifdef* or *#ifndef* directives and undefined with a *#undef* directive. Since these directives may only test one macro per directive the preprocessor *defined* operator may be used to test the status of more than one macro per directive. The following are examples:

```
#ifdef DEBUG          #ifndef SKIPCODE          #if defined(DEBUG) || !defined(SKIPCODE)
        ...                   ...                       ...
#endif                #endif                    #endif
```

NOTE D.2

“Include Guards” to Prevent Multiple Inclusions of Header File Contents

Multiple occurrences of the same *#include* directive within a particular file cannot always be avoided. For example, if header files *abc.h* and *xyz.h* both contain the directive *#include "ijk.h"* and a programmer includes both of these files in a third file, the directive *#include "ijk.h"* will appear twice in that third file. Although some compilers may ignore the fact that the contents of file *ijk.h* occurs more than once, good programming practice dictates that such duplication never occur. This can be guaranteed by enclosing the contents of every header file (but never an implementation file) in an “include guard”. To accomplish this the first thing in every header file must be a *#ifndef* directive, where the identifier being tested is the uppercase name of the file with underbars replacing all periods and other characters that are not allowed in identifiers. In addition, if the name of the file starts with a numeric digit the include guard identifier must be preceded with an underbar (since identifiers cannot begin with a numeric digit). The include guard ends at the end of the file with a *#endif* directive. Here is an example of implementing an include guard in a header file named *TestFile1.h*:

```
#ifndef  TESTFILE1_H        /* first thing in the file - beginning of include guard */
#define  TESTFILE1_H        /* second thing in the file - continuation of include guard */
    …                       /* everything you really want in the file
#endif                      /* last thing in the file - end of include guard */
```

Here are some more examples of include guard naming, followed by the details of how include guards work:

File Name	Include Guard Name
C1A2E3_main-test2.c	*None – Implementation File*
C1A2E3_main-test2.h	C1A2E3_MAIN_TEST2_H
iostream	IOSTREAM
6*Hello&.h	_6_HELLO__H

Assume the following contents of header file *filename.h*:

```
#ifndef  FILENAME_H         /* first thing in the file - beginning of include guard */
#define  FILENAME_H         /* second thing in the file - continuation of include guard */
    extern int g_status = 0;  /* what you really want in file filename.h... */
    double total(void);       /* ...more of what you really want in file filename.h */
#endif                      /* last thing in the file - end of include guard */
```

So when a multiple inclusion of *filename.h* occurs,

```
#include "filename.h"       /* first inclusion */
#include "filename.h"       /* second inclusion */
```

It first expands to,

```
#ifndef  FILENAME_H         /* first inclusion: FILENAME_H not defined at this point... */
#define  FILENAME_H         /* ...so it gets defined here... */
    extern int g_status = 0;  /* ...and all information you really want... */
    double total(void);       /* ...in file filename.h gets included here */
#endif                      /* end of first inclusion of filename.h */
#ifndef  FILENAME_H         /* second inclusion: FILENAME_H got defined above... */
                            /* ...so contents of filename.h is not re-included here */
#endif                      /* end of second inclusion of filename.h */
```

And finally completely expands to:

```
    extern int g_status = 0;  /* everything you really want in file... */
    double total(void);       /* ...filename.h appears only once */
```


NOTE D.3

Commenting Out Sections of Code

It is frequently desirable, for testing or debugging reasons, to comment out one or more sections of code. Although the first thought might be to start those sections with **/*** and end them with ***/**, this will fail miserably if they contain any comments since comments cannot be nested. Placing a C++ comment token **//** in front of each line then removing them later is too cumbersome. The solution is simple: Merely begin each such section with *#if 0* and end it with *#endif*. For example,

```
    #if 0
        None of this code...
        ...gets compiled.  It...
        ...may contain comments...
        ...and other preprocessor directives.
    #endif
```

Conditional Inclusion for Multilevel Debugging

Often the simple true/false test provided by the preprocessor *assert/NDEBUG* facility (see Note D.4) is insufficient for detailed debugging. It may instead be desirable to output the actual values of expressions at various points in program execution. By using output statements in conjunction with preprocessor conditional inclusion directives, any level of debugging can be achieved. That code need not be physically removed from the finished product but can simply be not included by the preprocessor. Note:

- Never use C/C++ **if** statements in place of preprocessor conditional inclusion constructs if the debugging code will not be removed from the finished product. To do so results in an unnecessary increase in code size since the debugging code will never be used in the finished product.

- If a program's failure includes a crash, use non-buffered output statements (*fprintf(stderr, ...)*, *fputs(..., stderr)*, *cerr*, etc.) or the program crash point, as indicated by the last message output before the crash, can be misleading.

```
#define DEBUG1                                  /* define for level 1 debugging */
#define DEBUG2                                  /* define for level 2 debugging */

...program statements
#ifdef DEBUG1                                   /* level 1 debugging */
    ...debugging code such as:
#   include <math.h>
    fprintf(stderr, "x = %d, y = %f, z = %n\n", x, y, z);
    y = 3. * cos(m);
    fprintf(stderr, "m = %e, y = %f\n", a, y);
#   ifdef DEBUG2                                /* level 2 debugging */
        ...debugging code such as:
        for (pointNr = 0; pointNr < POINTS; ++pointNr)
            fprintf(stderr, "point[%d] = %d\n", pointNr, point[pointNr]);
#   endif                                       /* end level 2 debugging */
#endif                                          /* end level 1 debugging */
...program statements
```

NOTE D.4

Debugging Using the Preprocessor *assert* Macro

The preprocessor *assert* macro provides a built in rudimentary aid to program debugging. *assert* is always implemented as a macro and is defined in the standard header files <*assert.h*> (C) and <*cassert*> (C++). Its syntax is

```
void assert(int expression);
```

If the macro *NDEBUG* is defined by the programmer before the point in the source file where <*assert.h*> or <*cassert*> is included, the *assert* macro will produce no code, will do nothing, and will be defined as

```
#define assert(ignore) ((void)0)
```

If *NDEBUG* has not been defined, *assert* will be defined in an implementation-specific fashion, which if its argument is false (zero), will output a diagnostic message and call the *abort* function to terminate the program. The message includes the stringization of the *assert* argument, the file name, and the line number. This facility allows the programmer to use assertions freely during program development and then to effectively discard them later by defining the macro *NDEBUG*.

Because the underlying code for one or more *assert*s can greatly increase the code size of a program, it is important that a finished, debugged program have all *assert*s either physically removed or effectively removed by defining *NDEBUG*. Defining *NDEBUG* has the advantage of permitting all debugging code to be immediately reinstated by simply commenting out the definition, where physically removing each *assert* is usually much more difficult to restore.

Assume the following program is in a file named *test.c* at the line numbers shown. The definition of *NDEBUG* is easily activated/deactivated by removing or inserting the *//* in front of it:

```
// #define NDEBUG


#include <cassert>
#include <cstddef>
#include <cstring>

int main()
{
    char *charPtr, *myString = "hello world\n";
    int x, y;

    ...
    // aborts if empty string in myString and outputs: Assertion failed: strlen(myString), file test.c, line ...
    assert(strlen(myString));
    ...
    // aborts if x is not less than y and outputs: Assertion failed: x < y, file test.c, line ...
    assert(x < y);
    ...
    // aborts if charPtr != NULL and outputs: Assertion failed: charPtr == NULL, file test.c, line ...
    assert(charPtr == NULL);

    return EXIT_SUCCESS;
}
```

NOTE D.5

Predefined Macros

The language standards require that certain macros be predefined by the compiler and not be undefinable by program code. The four that are of primary interest and that are common to both C and C++ are:

__LINE__	The presumed line number of the current source line (an **int**)
__FILE__	The presumed name of the source file (a string)
__DATE__	The date of compilation of the source file (a string)
__TIME__	The time of compilation of the source file (a string)

The values of all predefined macros except __LINE__ and __FILE__ remain constant in any file. __LINE__ and __FILE__ may be changed by the preprocessor #line directive. Other implementation-specific macros may be predefined such as __TURBOC__, THINK_C, MSDOS, AZTEC_C, etc.

The following listing is typical of some conditional inclusion directives that might be found in an arbitrary program header file named *sysio.h*. The first two directives and the last directive prevent multiple inclusions of the contents of this file. Note the use of implementation-specific predefined macros to make the code portable to several different environments without requiring code modifications. Among other things, the inclusion of this header permits the names *in* and *out* to be used when referring to a compiler's I/O port functions, even though the actual names differ between compilers. Additionally, note how different header files are included, depending on the compiler/system being used.

```
#ifndef  SYSIO_H                  /* if SYSIO_H not defined, then this file not yet included */
#define SYSIO_H                   /* define SYSIO_H */

#ifdef __TURBOC__                 /* Borland compiler predefined macro */
#       include <dos.h>           /* this compiler's I/O header file */
#       define out  outport       /* define I/O word output function name */
#       define in  inport         /* define I/O word input function name */
#elif defined(MSDOS)              /* Microsoft compiler predefined macro */
#       include <io.h>            /* this compiler's I/O header file */
#       define out  outw          /* define I/O word output function name */
#       define in  inw            /* define I/O word input function name */
#elif defined(BRANDX_C)           /* Brand-X C compiler, ROM or native versions */
#       ifdef MCH_ROM             /* ROM cross-compiler version in use */
#               undef STACK_LIMIT /* undefine a macro */
#               define out  out68 /* define I/O word output function name */
#               define in  in68   /* define I/O word input function name */
#       else                      /* native compiler version in use */
#               define out  out86 /* define I/O word output function name */
#               define in  in86   /* define I/O word input function name */
#       endif                     /* end of Brand-X version determination */
#else                             /* default to UNIX cc compiler */
#       error  Unrecognized Compiler!  /* diagnostic message including: Unrecognized Compiler! */
#endif                            /* end of system determination */

#if defined(MSDOS) || !defined(__SPECIAL__) && defined(__OPTIMIZE__)        /* optimized I/O */
#       define MODE "fastcode.h"
#else
#       define MODE "stdcode.h"
#endif
#include MODE                     /* include code type information header file */

#endif                            /* end of #ifndef SYSIO_H */
```

NOTE D.6A

"Where Am I" Using __*LINE*__ and __*FILE*__

When a message is output by a running program, whether during debugging or production operation, it is often desirable to know precisely where in the code that message is coming from. For example, a program may attempt to open a file at several points and output an error message if the open fails. But how do you know which of the opens failed? While you could hard code the line number into the message itself when you are writing the program, this is very cumbersome and error prone since line numbers change as code is added or deleted. There is a much better way! Since the __LINE__ macro represents the line number on which it appears and the __FILE__ macro represents the name of the file in which it appears, simply outputting their values as part of the message solves the problem and you will always know exactly where the message is coming from:

```
printf("Line %d in file %s\n", __LINE__, __FILE__);
```

or

```
cout << "Line " << __LINE__ << "in file " << __FILE__ << '\n';
```

The Preprocessor *#line* Directive

The values of the predefined macros __LINE__ and __FILE__ may be altered within a file using the syntax

```
#line number "filename"
```

where *"filename"* is optional. When the C/C++ compiler finds a problem during compilation, it typically outputs a message giving both the file name and the line number on which the problem occurred. The preprocessor *#line* directive lets the programmer change both the line number and the file name presumed by the compiler. To understand the purpose of this, first consider the following code:

In header file *Header.h*:

```
1.      typedef long type_t;
2.      struct node {int x, y;};
3.      ...97 more lines
100.    extern double g_answer;
101.    void Average(type_t *li);
102.    double sqr(double n);
```

In C source file *Test.c*:

```
1.      #include "Header.h"
2.
3.      void T1(void)
4.      {
5.              int cycles = 3;
6.              cycles = t2=;                   /* note the syntax error on this line (line 6) */
7.      }
```

...............CONTINUED

NOTE D.6B **...............CONTINUATION**

The Preprocessor *#line* Directive, Cont'd.

During compilation, a temporary intermediate file is typically created by the preprocessor consisting of the original C/C++ source file expanded to include the contents of all included files as shown below. Note the line numbering of the new intermediate file, which is the file that actually gets compiled:

In the preprocessor generated intermediate file, arbitrarily named *xyz.$$$*:

```
1.    typedef long type_t;
2.    struct node {int x, y;};
3.    ...97 more lines
100.  extern double g_answer;
101.  void Average(type_t *li);
102.  double sqr(double n);
103.
104.  void t1(void)
105.  {
106.      int cycles = 3;
107.      cycles = t2=;         /* note the syntax error on this line (line 107) */
108.  }
```

Upon compilation, an error message like *"syntax error in file xyz.$$$, line 107"* might be generated, which is virtually useless since the actual programming error is in file *Test.c*, which has just 7 lines! If, however, the preprocessor inserts a *#line* directive into the intermediate file each time it adds another file, the identity and structure of each of the original files can be preserved and a meaningful message like *"syntax error in file Test.c, line 6"* can be generated. The above intermediate file would then look like:

```
1.    #line 1 "Header.h"            /* next line interpreted by compiler as line 1 of file Header.h */
2.    typedef long type_t;          /* interpreted by compiler as line 1 of file Header.h */
3.    struct node {int x, y;};      /* interpreted by compiler as line 2 of file Header.h */
4.    ...97 more lines              /* ...97 more lines of file Header.h */
101.  extern double g_answer;       /* interpreted by compiler as line 101 of file Header.h */
102.  void Average(type_t *li);     /* interpreted by compiler as line 102 of file Header.h */
103.  double sqr(double n);         /* interpreted by compiler as line 103 of file Header.h */
104.  #line 2 "Test.c"              /* next line interpreted by compiler as line 2 of file Test.c */
105.                                /* interpreted by compiler as line 2 of file Test.c */
106.  void t1(void)                 /* interpreted by compiler as line 3 of file Test.c */
107.  {                             /* interpreted by compiler as line 4 of file Test.c */
108.      int cycles = 3;           /* interpreted by compiler as line 5 of file Test.c */
109.      cycles = t2=;             /* note the syntax error on this line (now line 6 of file Test.c) */
110.  }                             /* interpreted by compiler as line 7 of file Test.c */
```

In summary, *#line number "filename"* causes the line following the directive to be presumed to be line number *number* of file *filename*. If *"filename"* is omitted the *filename* currently in effect is retained. In the general case, the syntax

```
#line tokens
```

is allowed as long as it expands to the required form.

NOTE D.7

The *#error* Directive

The syntax of the preprocessor *#error* directive is

```
#error tokens
```

If reached by the preprocessor a diagnostic message is output, including the sequence of tokens, and compilation is terminated. This directive is typically used to detect inconsistencies and constraints during preprocessing rather than during compilation or runtime.

Example 1:
Ensure that the value of SIZE is an integral multiple of 1024 (for memory alignment?)

```
#if SIZE % 1024 != 0                              /* is size mod 1024? */
#     error SIZE must be a multiple of 1024!      /* if not, output an error message */
#endif                                            /* end of SIZE check */
```

Example 2:
Ensure that the buffer is of sufficient size (to prevent overflow or truncated data?)

```
#define BUFFER_SIZE 255

#if BUFFER_SIZE < 256
#   error BUFFER_SIZE is too small.
#endif
```

generates the error message:
BUFFER_SIZE is too small.

Example 3:
Ensure that a C++ compiler is being used (to prevent bogus compiler error messages?)

```
#if !defined(__cplusplus)
#   error C++ compiler required!
#endif
```

Example 4:
Ensure that the UNIX operating system is being targeted (because part of the program code is UNIX dependent?)

```
#ifndef __unix__
#   error Only UNIX is supported!
#endif
```

NOTE D.8A

The # Stringization Operator

The preprocessor unary stringization operator is used in conjunction with macro arguments and causes them to be expanded into string literals. An example of its practical usage is in the standard library macro *assert* (header file <*assert.h*>) where it is used to convert the macro's integer expression into a string that gets output.

```
#include <stdio.h>
#include <stdlib.h>

#define CatPrint(a, b)   puts(#a  #b)                        /* concatenate & output arguments as strings */
#define PrintFalse(expr)  if (!(expr)) puts(#expr)            /* test an expression and output it if false */
#define PrintIvar(z) printf("The value of %s is %d\n", #z, (z))  /* output value of int variable */

int main(void)
{
    int x = 5, y = 0;

    CatPrint(alpha, bet);              /* expands to:   puts("alpha" "bet"); */
    PrintFalse(x < y);                 /* expands to:   if (!(x < y)) puts("x < y"); */
    PrintIvar(x);                      /* expands to:   printf("The value of %s is %d\n", "x", x) */

    return EXIT_SUCCESS;
}
```

PROGRAM OUTPUT:

```
    alphabet
    x < y
    The value of x is 5
```

The ## Token Merge Operator

The preprocessor token merge operator is used in conjunction with a macro to merge together two operator-separated tokens when the macro is expanded. Whitespace on either side of the operator is ignored.

Example 1:

```
#include <stdio.h>

#define   Printx(a)   printf("%i\n", x ## a);       /* literal x and arg a merged after expansion */

void MergeSomething(void)
{
    int x13 = 45;

    Printx(13);                        /* expands to:   printf("%i\n", x13); */
}
```

...............CONTINUED

NOTE D.8B **...............CONTINUATION**

The ## Token Merge Operator, cont'd.

Example 2:

This cryptic example uses the token merge operator to create multiple specific identifiers using a generic macro. This occurs in real programs when identifiers differ only in part, such as changing sequence numbers. Both versions of the following program use identifiers that differ only in their numeric suffixes. The token merge operator in version 2 permits the use of macros that will merge the different numeric suffixes with the unchanging prefixes, thereby forming the desired identifiers. This means less typing and arguably more maintainable code.

```
/* Sizes of 8 arrays -- For both versions of the program */
#define SZ00 125
#define SZ01 200
#define SZ02 399
#define SZ03 488
#define SZ04 574
#define SZ05 661
#define SZ06 752
#define SZ07 843

/* Version 1 -- Without the token merge operator */

/* Declare arrays of ints */
int w00[SZ00], w01[SZ01], w02[SZ02], w03[SZ03], w04[SZ04], w05[SZ05], w06[SZ06], w07[SZ07];

struct                                  /* define, declare, and initialize an array of structures */
{
    int *start, *current, *end;         /* pointers into an array of ints */
} infoData[] =
{
    { w00, w00, w00 + sizeof(w00) }, { w01, w01, w01 + sizeof(w01) },
    { w02, w02, w02 + sizeof(w02) }, { w03, w03, w03 + sizeof(w03) },
    { w04, w04, w04 + sizeof(w04) }, { w05, w05, w05 + sizeof(w05) },
    { w06, w06, w06 + sizeof(w06) }, { w07, w07, w07 + sizeof(w07) }
};


/* Version 2 -- With the token merge operator */

/* Use this macro to declare an array */
#define ay(arrayNbr) w##arrayNbr[SZ##arrayNbr]

/* Use this macro as an array element initializer */
#define el(arrayNbr) { w##arrayNbr, w##arrayNbr, w##arrayNbr + sizeof(w##arrayNbr) }

/* Declare arrays of ints  --- Expands to version 1 */
int ay(00), ay(01), ay(02), ay(03), ay(04), ay(05), ay(06), ay(07);

struct                                  /* define, declare, and initialize an array of structures */
{
    int *start, *current, *end;         /* pointers into an array of ints */
} infoData[] =
{
    el(00), el(01), el(02), el(03), el(04), el(05), el(06), el(07)        /* Expands to version 1 */
};
```

NOTE D.9

The *#pragma* Directive

The syntax of the preprocessor *#pragma* directive is

```
#pragma tokens
```

It was designed as an implementation-defined catch-all for adding new preprocessor functionality or providing implementation-defined information to the compiler. Pragmas are not standardized and implementations should ignore any pragma information they don't understand. Some compiler-specific examples of pragmas are:

#**pragma startup** *function-name [priority]*
#**pragma exit** *function-name [priority]*

…allow the program to specify function(s) that should be called either upon startup (before the *main* function is called), or program exit (just before the program terminates through *_exit*. The specified *function-name* must be a previously declared function taking no arguments and returning **void**. The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. Unspecified priorities default to 0.

#pragma option *[options...]*

…used to include command line options in the program source file(s). Most command line options can be used.

#pragma saveregs

…guarantees that a **huge** function will not change the values of any machine registers when it is entered.

#pragma GCC dependency *[text...]*

…allows you to check the relative dates of the current file and another file. If the other file is more recent than the current file a warning is issued. This is useful if the current file is derived from the other file, and should be regenerated. The other file is searched for using the normal include search path. Optional trailing text can be used to give more information in the warning message. For example:

#pragma GCC dependency *"parse.y"*
#pragma GCC dependency *"/usr/include/time.h" rerun fixincludes*

#pragma GCC poison *identifier(s)*

Sometimes there is an identifier that you want to remove completely from your program and make sure that it never creeps back in. To enforce this, you can *poison* the identifier with this pragma. **#pragma GCC poison** is followed by a list of identifiers to poison. If any of those identifiers appears anywhere in the source after the directive it is a hard error. For example,

#pragma GCC poison *printf sprintf fprintf*
sprintf(some_string, "hello");

Appendix D Practice Exercises (not for submission or grading)

D-1. A careless programmer has attempted to temporarily comment out everything inside the body of a **for** loop using the tokens indicated. Explain why this won't work and make a simple modification that does work.

```
void AverageUserValues(void)                /* average numbers input by the user */
{
    int loopCount, quantity, value, average;

    printf("Enter the number of values to average: ");
    scanf("%d", &quantity);
    if (quantity > 0) {                                         /* if there are numbers to average */
        average = 0;                                            /* initialize average */
        for (loopCount = 1; loopCount <= quantity; ++loopCount) {   /* one loop for each value */
→→  /*
            printf("Enter value #%d: ", loopCount);             /* prompt user for a value */
            scanf("%d", &value);                                /* get the value */
            average += value;                                   /* add value to running total */
→→  */
        }
        average /= quantity;                                    /* calculate the average */
        printf("The average is %d: ", average);                 /* output the average */
    }
}
```

D-2. What is wrong with the following header file that prevents it from being standards compliant and what problems can this cause? Modify it to correct the problem.

```
#define VERSION 2.01
#define IsNull(ptr) ((ptr) == NULL)
typedef unsigned int CTLREG;
size_t CountBytes(char *cp);
```

D-3. All standard C compilers have at least five predefined macros that cannot be undefined and most have several more. Search your compiler documentation and list all of its predefined macros.

D-4. Write some code that uses the preprocessor *assert* facility to ensure that a program does not continue if the expression *scanf("%d%d", &x, &y)* does not assign values to both *x* and *y*. Your code must not actually check the values of *x* or *y* or use any other variables.

D-5. Using no variables or #defines, write a program that displays the date and time the program was compiled, the name of the file, and the line number on which the output statement is located.

D-6. Using preprocessor directives only, detect if the macro *VERSION* has been defined and displays an error message at compile time if not. Test your code by compiling both with and without *VERSION* defined.