THE PREPROCESSOR

To use C effectively you really have to master two languages: the C language proper, and the preprocessor. Before a compiler begins the usual chores of syntax checking and instruction translation, it submits your program to a preliminary phase called preprocessing that alters the very *text* of the program according to your instructions. The altered text that the compiler sees is called a *translation unit*. In particular, the preprocessor performs the following three functions for you:

1) header/source file inclusion
2) macro expansion
3) conditional compilation

In this article I will illustrate these features of the preprocessor.

THE INCLUDE DIRECTIVE

One of the first source lines any C programmer sees or composes is this:

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

Take a moment right now and jot down everything you know about this statement...

Let's see how you did. stdio.h is of course a standard library *header*, so called because such include directives usually appear near the beginning of a source file so that their definitions will be in force throughout the rest of the compilation. We commonly think of it as a header *file*, but there is no requirement that the definitions and declarations pertaining to standard input and output reside in a file. The C standard only requires that those definitions replace the include directive in the text of the program before translation. They could reside in tables internal to the preprocessor. Most implementations do supply header files for the standard library, however. MSDOS compilers install header files in a suitable subdirectory. Here is a sampling:

```
\BC4\INCLUDE             /* Borland C++ 4.0 */
\MSVC\INCLUDE                    /* Microsoft Visual C++ */
\SYMANTEC\INCLUDE        /* Symantec C++ */
\WATCOM\H                        /* Watcom C/C++ */
```

On UNIX systems you will find them in /usr/include. Since an implementation is not obliged to even supply header information in physical files, it should be no surprise that those that do needn't name the files after the include directive. How can a compiler supply a file named stdio.h on a platform whose file system doesn't allow periods in a file name? On MSDOS systems there can be no file that exactly matches the C++ header <strstream.h>, for example, because the file system only allows up to eight characters before the period. Most MSDOS implementations map header names into file names by truncating the base part (the portion before the period) to eight characters, and the extension (the portion after the period) to three (so the definitions for <strstream.h> reside in the file STRSTREA.H). A standard-conforming implementation must supply a mapping to the local file system for header names with at least six characters before the period and one character after.

Conforming compilers also support include directives with string targets:

```
#include "mydefs.h"
```

The string must represent a name recognized by the local file system. The file must be a valid C/C++ source file, and like the standard headers, usually contains function prototypes, macro definitions and other declarations. An implementation must specify the mechanism it uses to locate the requested source file. On platforms with hierarchical file systems, the compiler usually searches the current directory first, and failing that, tries the same place where the standard headers are. Because standard header names are special preprocessing tokens and not

strings, any backslashes in a header name are not escape characters:

```
#include <sys\stat.h>                    /* \, not \\ */
```

Note that you don't use double backslashes, even with the string form:

```
#include "\project\include\mydefs.h"
```

Included files may themselves contain other include directives, nested up to eight levels deep. Since some definitions (like typedefs) must only appear once during a compilation, you must guard against the possibility of a file being included more than once. The customary technique is to define a symbol associated with the file, and exclude the text of the file from the compilation if the symbol has already been seen by the compiler:

```
/* mydefs.h */
#ifndef MYDEFS_H
#define MYDEFS_H

<declarations/definitions go here>

#endif
```

OTHER PREPROCESSOR DIRECTIVES

As you can see, there's more to the #include directive than meets the eye. There are eleven other preprocessor directives you can use to alter your source text in meaningful ways (see Table 1). All begin with the '#' character, which must be the first non-space character on its source line.

The #define directive creates *macro* definitions. A macro is a name for a sequence of zero or more preprocessing tokens. (Valid preprocessing tokens include valid C language tokens like identifiers, strings, numbers, and operators, preprocessing directives, header names, and any single character). For example, the line

```
#define MAXLINES 500
```

associates the text "500" (without the quotes) with the symbol MAXLINES. The preprocessor keeps a table of all symbols created by the #define directive, along with the corresponding replacement text. Whenever it encounters the token MAXLINES outside of a quoted string or comment, it replaces it with the token 500. It is important to remember that this is *text replacement*. In later phases of compilation it appears as if you actually typed 500 instead of MAXLINES. No semantic analysis occurs during preprocessing.

A macro without parameters like MAXLINES above is sometimes called an *object-like* macro because it defines a program constant that looks like an object. Because object-like macros are constants, it is customary to type them in upper case as a hint to the reader. You can also define *function-like* macros with zero or more parameters, such as

```
#define beep()   putc('\a',stderr)
#define abs(x)   ((x) >= 0 ? (x) : (-(x)))
#define max(x,y) (((x) > (y)) ? (x) : (y))
```

Thee must be no whitespace between the macro name and the first left parenthesis. The expression

```
abs(-4)
```

expands to

```
((-4) >= 0 ? (4) : (-(4)))
```

just as if you had typed it that way. It is important that macro parameters (like x above) are parenthesized in the replacement text. This avoids precedence surprises with complex argument expressions. For example, if you had used the naive mathematical definition for absolute value:

```
x >= 0 ? x : -x
```

then the expression abs(a - 1) would expand to

```
a - 1 >= 0 ? a - 1 : -a - 1
```

which is incorrect when a - 1 < 0 (it should be -(a - 1)).

Even if you put parentheses around all arguments, you should usually parenthesize the entire replacement expression as well to avoid surprises with respect to the surrounding text. To see this, define abs() without enclosing parens:

```
#define abs(x) (x) >= 0 ? (x) : (-x)
```

Then abs(a) - 1 expands to

```
(a) >= 0 ? (a) : -(a) - 1
```

which is incorrect when a is non-negative.

It is also dangerous to use expressions with side effects as macro arguments. For example, the macro call abs(i++) expands to

```
((-i++) >= 0 ? (i++) : (-(i++)))
```

No matter what the value of i happens to be, it gets incremented twice, not once, which probably isn't what you had in mind.

PRE-DEFINED MACROS

Conforming implementations supply the five built-in object-like macros in Table 2. The last three remain constant during the compilation of a source file. Any other pre-defined macros that a compiler provides must begin with a leading underscore followed by either an uppercase letter or another underscore. C++ implementations also define the symbol __cplusplus. Most compilers run in multiple modes, some of which are not standard-conforming. To guarantee that the sample program in Listing 1 will run correctly under Borland C, for example, you need to run in "ANSI mode" (via the "-A" command-line option). You may not redefine any of these five macros with the #define directive, nor remove them with the #undef directive.

Conforming compilers also provide a function-like macro, assert, which you can use to put diagnostics in programs. If its the argument evaluates to zero, assert prints the argument along with source file and line (using __FILE__ and __LINE__) to standard error and aborts the program (see Listing 2). For more information on using the assert macro, see the CODE CAPSULE "File Processing, Part 2" in the June 1993 issue of this journal.

A compiler is allowed to provide macro versions for any functions in the standard library (getc and putc usually come as macros for efficiency). With the exception of a handful of required function-like macros (assert, setjmp, va_arg, va_end, and va_start), an implementation must supply true function versions for all functions in the standard library. A macro version of a library function in effect hides its prototype from the compiler, so its arguments are not type-checked during translation. To force the true function to be called, remove the macro definition with the #undef

directive, for example

```
#undef getc
```

Alternatively, you can surround the function name in parentheses when you call it:

```
c = (getc)(stdin);
```

This cannot match the macro definition since a left parenthesis does not immediately follow the function name.

CONDITIONAL COMPILATION

You can selectively include or exclude segments of code with conditional directives. For example, you can embed the following excerpt in your code to accommodate for the difference in the syntax of the delete operator for earlier versions of C++:

```
#if VERSION < 3
   delete [strlen(p) + 1] p;
#else
   delete [] p;
#endif
```

Your compiler probably supplies a macro similar to VERSION (Borland C++ defines __BCPLUSPLUS__, Microsoft _MSCVER). The target of an #if directive must evaluate to an integer constant, and obeys the usual C rule of non-zero means true, zero false. You cannot use casts or the sizeof operator in such expressions.

The #if directive is handy when you want to comment-out long passages of code. You can't just wrap such sections in a single, enclosing comment because there are likely to be comments in the code itself (right?), causing the outer comment to end prematurely. It is better to make the code in question the group of an #if directive that always evaluates to 0:

```
#if 0
<put code to be ignored here>
#endif
```

PREPROCESSOR OPERATORS

Sometimes you just want to know if a macro is defined, without using its value. For example, if you only support two compilers, you might have something like the following in your code:

```
#if defined _MSCVER
<put Microsoft-specific statements here>
#elif defined __BCPLUSPLUS__
<put Borland-specific statements here>
#else
#error Compiler not supported.
#endif
```

The defined operator evaluates to 1 if its target is present in the symbol table, which means that it was either the subject of a previous #define directive or the compiler provided it as a built-in macro. The #error directive prints its message to stderr and halts. It isn't necessary to assign a macro a value. For example, to insert debug trace output into your program, you can do the following:

```
#if defined DEBUG
fprintf(stderr,"x = %d\n",x);
#endif
```

To define the DEBUG macro, just insert the following statement before the first use of the macro:

```
#define DEBUG
```

The following equivalences exist:

```
#if defined X       <==>   #ifdef X
#if !defined X      <==>   #ifndef X
```

Using the defined operator is more flexible than the equivalent directives on the right because you can combine multiple tests as a single expression:

```
#if defined __cplusplus && !defined DEBUG
```

defined is one of three preprocessor operators (see Table 3).

The operator #, the "stringizing" operator, encloses a macro argument in quotes. As the program in Listing 3 illustrates, this can be useful for debugging. The trace() macro encloses its arguments in quotes so they become part of a printf format statement. For example the expression trace(i,d) becomes

```
printf("i" " = %" "d" "\n",i);
```

and, after the compiler concatenates adjacent string literals it sees this:

```
printf("i = %d\n",i);
```

There is no way to build quoted strings like this without the stringizing operator because the preprocessor ignores macros inside quoted strings.

The token-pasting operator, ##, concatenates two tokens together to form a single token. The call trace2(1) in Listing 4 is translated into

```
trace(x1,d)
```

Any space surrounding these two operators is ignored.

IMPLEMENTING assert()

Implementing assert reveals an important fact about macros. Since the action of assert depends on the result of a test, you might first try an if statement:

```
#define assert(cond) \
  if (!(cond)) __assert(#cond,__FILE__,__LINE__)
```

where the function __assert prints the message and halts the program. This causes a problem, however, when assert finds itself within an if statement:

```
if (x > 0)
   assert(x != y)
else
```

```
    /* whatever */
```

because it expands into

```
if (x > 0)
   if (!(x != y)) __assert("x != y","file.c",7);
else
   /* whatever */
```

The indentation is misleading because the second if intercepts the else:

```
if (x > 0)
   if (!(x != y))
      __assert("x != y","file.c",7);
   else       /* OOPS! New control flow! */
      /* whatever */
```

The usual fix for nested if problems like this is to use braces:

```
#define assert(cond) \
  {if (!(cond)) __assert(#cond,__FILE__,__LINE__)}
```

but this expands into

```
if (x > 0)
   {if (!(x != y)) __assert("x != y","file.c",7)};
else
   /* whatever */
```

and the combination }; in the second line creates a null statement that completes the outer if, leaving a dangling else, which is a syntax error. The correct way to define assert is in Listing 5. (Listing 6 has the implementation of the support function __assert()). In general, when a macro must make a choice, it is good practice to write it as an expression and not as a statement.

MACRO MAGIC

It's important to understand precisely what steps the preprocessor follows to expand macros, otherwise you can be in for some mysterious surprises. For example, if you insert the following line near the beginning of Listing 4

```
#define x1 SURPRISE!
```

then trace2(1) is expanded into

```
trace(x ## 1,d)
```

which in turn becomes

```
trace(x1,d)
```

But the preprocessor doesn't stop there. It *rescans* the line to see if any other macros need expanding. The final state of the program text that the compiler sees is in Listing 7.

To further illustrate, consider the text in Listing 8. It is not a complete program, by the way, but is for preprocessing only - don't try to compile it all the way. (If you have Borland C use the CPP command). The output from the

preprocessor appears in Listing 9. The str() macro just puts quotes around its argument. It might appear that xstr() is redundant, but there is an important difference between it and str(). The output of the statement str(VERSION) is of course

```
"VERSION"
```

but xstr(VERSION) expands to

```
str(2)
```

because arguments not connected with a # or ## are *fully expanded* before they replace their respective parameters. Then the statement is rescanned, giving "2". So in effect, xstr() is a version of str() that expands its argument before quoting it.

The same relationship exists between glue() and xglue(). The statement glue(VERSION,3) concatenates its arguments into the token VERSION3, but xglue(VERSION,3) first expands VERSION giving

```
glue(2,3)
```

which in turn rescans into the token 23. The next two statements are a little trickier:

```
        glue(VERS,ION)
==      VERS ## ION
==      VERSION
==      2
```

and

```
        xglue(VERS,ION)
==      glue(VERS,ATILE)
==      VERS ## ATILE
==      VERSATILE
```

Of course, if VERSATILE were a defined macro it would be further expanded. The last four statements in listing 8 expand as follows:

```
        ID(VERSION)
==      "This is version "xstr(2)
==      "This is version "str(2)
==      "This is version ""2"

        INCFILE(VERSION)
==      xstr(glue(version,2)) ".h"
==      xstr(version2) ".h"
==      "version2" ".h"

        str(INCFILE(VERSION))
==      #INCFILE(VERSION)
==      "INCFILE(VERSION)"

        xstr(INCFILE(VERSION))
==      str("version2" ".h")
==      #"version2" ".h"
==      "\"version2\" \".h\""
```

For obvious reasons, the # operator inserts escape characters before all embedded quotes and backslashes.

The macro replacement facilities of the preprocessor clearly offer you an incredible amount of flexibility (too much, some would say). There are two limitations to keep in mind:

1) If at any time the preprocessor encounters the current macro in its own replacement text, no matter how deeply nested in the process, it does not expand it but leaves it as-is (otherwise the process would never terminate!). For example, given the definitions

```
#define F(f) f(args)
#define args a,b
```

F(g) expands to g(a,b), but what does F(F) expand to? (Answer: F(a,b)).

2) If a fully-expanded statement resembles a preprocessor directive, e.g., if after expansion the result is an #include directive, it is *not* invoked, but left verbatim in the program text. (Thank goodness!).

CHARACTER SETS AND TRIGRAPHS

The character set you use to compose your program doesn't have to be the same as the one in which the program runs. This is certainly true of non-English applications. A C translator only understands English alphanumerics, the graphics characters used for operators and punctuators (there are 29 of them), and a few control characters (newline, horizontal tab, vertical tab, and form-feed). This is the *source character set*. Any other characters may appear only in quoted strings, character constants, header names or comments. The *execution character set* is implementation-defined, but must contain characters representing alert ('\a'), backspace ('\b'), carriage return ('\r'), and newline ('\n').

Many non-U.S. environments do not support some of the elements of the source character set, making it impossible to write C programs. To overcome this obstacle, standard C defines a number of *trigraphs*, which are triplets of characters from the Invariant Code Set (ISO 646-1983) found in virtually every environment in the western world. Each trigraph corresponds to a character in the source character set which is not in ISO 646 (see Table 4). For example, whenever the preprocessor encounters the token ??= anywhere in your source text (even in strings), it replaces it with the encoding for the # character. The program in Listing 11 shows how to write the "Hello, world!" program from Listing 10 using trigraphs.

In an effort to enable more readable programs world-wide, the C++ draft standard defines a set of digraphs and new keywords for non-ASCII developers (see Table 5). Listing 12 shows what "Hello, world" looks like using these new tokens. Perhaps you agree that the symmetric look of the bracketing operators is easier on the eye.

PHASES OF TRANSLATION

The C standard defines eight distinct phases of translation. An implementation doesn't necessarily make eight separate passes through the code, of course, but the result of translation must behave as if it had. The eight phases are:

1. Physical source characters are mapped into the source character set. This includes trigraph replacement and things like mapping a carriage return/line feed to a single newline character in MSDOS. Borland separates trigraph processing into a separate executable file (TRIGRAPH.EXE).

2. All lines that end in a backslash are merged with their continuation line, and the backslash is deleted.

3. The source is parsed into preprocessing tokens and comments are replaced with a single space character. The C++ digraphs are recognized as tokens.

4. Preprocessing directives are invoked and macros are expanded. Steps 1 through 4 are repeated for any included files.

5. Escape sequences in character constants and string literals that represent characters in the execution set are converted (e.g., '\a' would be converted to a byte value of 7 in an ASCII environment).

6. Adjacent string literals are concatenated.

7. Traditional compilation: lexical and semantic analysis, and translation to assembly or machine code.

8. Linking: external references are resolved and a program image is made ready for execution.

The preprocessor consists of steps 1 through 4.

C++ AND THE PREPROCESSOR

C++ preprocessing formally differs from that of C only in the tokens it recognizes. A C++ preprocessor must recognize the tokens in Table 5 as well as .*, ->*, and ::. It must also recognize //-style comments and replace them with a single space. You should find a bigger difference in the way you *use* the C++ preprocessor, however. For example, as far as I can tell, there is little reason to define object-like macros anymore. You should use const variable definitions instead. The statement

```
const int MAXLINES = 500;
```

has a couple of advantages over

```
#define MAXLINES 500
```

Since the compiler knows the semantics of the object, you get stronger compile-time type checking. You can also inspect const objects with a symbolic debugger. Global const objects have internal linkage unless you explicitly declare them extern, so you can safely replace all your object-like macros with const definitions.

Function-like macros are *almost* unnecessary in C++. You can replace most function-like macros with inline functions. For example, replace the max macro above with

```
inline int max(int x, int y)
{
   return x >= y ? x : y;
}
```

You don't have to worry about parenthesizing to avoid precedence surprises, because this is a real function, with scope and type checking. You also don't have to worry about side effects like you do with macros, such as in the call

```
max(x++,y++)
```

But the macro version has the advantage of being able to accept arguments of any type, you say. No problem. Define max as a template instead:

```
template<class T>
inline int max(const T& x, const T& y)
{
   return x > y ? x : y;
}
```

Do keep in mind, however, that inline is a only *hint* to the compiler. Not all functions are amenable to inlining, especially those with loops and complicated control structures. Your compiler may tell you when it can't inline a function. In many cases, it is still better to define a function out-of-line than it is to define it as a macro and lose the type safety that a real function affords.

There is still room for function-like macros that use the stringizing or token-pasting operators. The program in Listing 13 uses stringizing and an inline function to test the new string class available with Borland C++ 4.0.

CONCLUSION

The preprocessor doesn't know C or C++. It is a language all its own. Many library vendors use it in intelligent ways to simplify the installation and use of their products. I encourage you to use it, but to use it prudently. It has some dark corners, which I've purposely avoided here. (Did you know there are statements that will hang some conforming preprocessors?) It is good practice, especially with C++, to do as much as you can in the programming language, and use the preprocessor only when you need to.

TABLE 1 - Preprocessor Directives

#include	Includes text of header or source file.
#define	Enters a symbol into the symbol table for the current compilation unit, with an optional value.
#undef	Removes a symbol from the symbol table.
#if #elif #else #endif	Control flow directives for conditional compilation.
#ifdef #ifndef	Symbol table query directives. (Also used for conditional compilation).
#error	Prints a message to stderr and aborts.
#line	Renumbers the current source line. Utilities like code generators use this to synchronize generated lines with original source lines in error messages.

```
#pragma          Compiler-dependent actions.
```

TABLE 2 - Pre-defined Macros

Macro	Value
__LINE__	The number of the current source line (equal to the number of newline characters read so far).
__FILE__	The name of the source file.
__DATE__	The date of translation in the form "Mmm dd yyyy".
__TIME__	The time of translation in the form "hh:mm:ss".
__STDC__	1, if the compilation is in "standard" mode.

LISTING 1 - Prints the Pre-defined Macros

```
/* sysmac.c:    Print system macros */

#include <stdio.h>

main()
{
    printf("__DATE__ == %s\n",__DATE__);
    printf("__FILE__ == %s\n",__FILE__);
    printf("__LINE__ == %d\n",__LINE__);
    printf("__TIME__ == %s\n",__TIME__);
    printf("__STDC__ == %d\n",__STDC__);
    return 0;
}

/* Output:
__DATE__ == Dec 18 1993
__FILE__ == sysmac.c
__LINE__ == 9
__TIME__ == 19:05:06
__STDC__ == 1
*/
```

LISTING 2 - Illustrates an assertion failure

```
/* fail.c */
#include <stdio.h>
#include <assert.h>

main()
{
    int i = 0;

    assert(i > 0);
    return 0;
}

/* Sample Execution:
C:>assert
Assertion failed: i > 0, file assert.c, line 9
Abnormal program termination
*/
```

TABLE 3 - Preprocessor Operators

Operator	Usage
#	Stringizing
##	Token pasting
defined	Symbol table query

Listing 3 - Illustrates the Stringizing Operator

```
/* trace.c: Illustrate a trace macro for debugging */

#include <stdio.h>

#define trace(x,format) \
    printf(#x " = %" #format "\n",x)

main()
{
    int i = 1;
    float x = 2.0;
    char *s = "three";

    trace(i,d);
    trace(x,f);
    trace(s,s);
    return 0;
}

/* Output:
i = 1
x = 2.000000
s = three
*/
```

LISTING 4 - Illustrates the Token-pasting Operator

```
/* trace2.c: Illustrate a trace macro for debugging */

#include <stdio.h>

#define trace(x,format) \
    printf(#x " = %" #format "\n",x)
#define trace2(i) trace(x ## i,d)

main()
{
    int x1 = 1, x2 = 2, x3 = 3;
    trace2(1);
    trace2(2);
    trace2(3);
    return 0;
}

/* Output:
x1 = 1
x2 = 2
x3 = 3
*/
```

LISTING 5 - Implementation of the assert macro

```
/* assert.h */
#ifndef ASSERT_H
#define ASSERT_H

extern void __assert(char *, char *, long);

#define assert(cond) \
    ((cond)          \
       ? (void) 0    \
       : __assert(#cond,__FILE__,__LINE__))

#endif
```

LISTING 6 - The __assert support function

```
/* xassert.c */
#include <stdio.h>
#include <stdlib.h>

void __assert(char *cond, char *fname, long lineno)
{
    fprintf(stderr,
            "Assertion failed: %s, file %s, line %ld\n",
            cond,fname,lineno);
    abort();
}
```

LISTING 7 - Preprocessed source with a surprise

```
main()
{
int SURPRISE! = 1, x2 = 2, x3 = 3;
printf("x1" " = %" "d" "\n",SURPRISE!);
printf("x2" " = %" "d" "\n",x2);
printf("x3" " = %" "d" "\n",x3);
return 0;
}
```

LISTING 8 - Illustrates macro rescanning

```
/* preproc.c:  Test # and ## preprocessing operators
 *
 *  NOTE:   DO NOT COMPILE! Preprocess only!
 */

/* Handy stringizing macros */
#define str(s) #s
#define xstr(s) str(s)

/* Handy token-pasting macors */
#define glue(a,b) a##b
#define xglue(a,b) glue(a,b)

/* Some definitions */
#define ID(x) "This is version " ## xstr(x)
#define INCFILE(x) xstr(glue(version,x)) ".h"
#define VERSION 2
#define ION ATILE

/* Expand some macros */
str(VERSION)
xstr(VERSION)
glue(VERSION,3)
xglue(VERSION,3)
glue(VERS,ION)
xglue(VERS,ION)

/* Expand some more */
ID(VERSION)
INCFILE(VERSION)
str(INCFILE(VERSION))
xstr(INCFILE(VERSION))
```

LISTING 9 - Preprocessed results from Listing 8

```
"VERSION"
"2"
VERSION3
23
2
VERSATILE

"This is version ""2"
"version2" ".h"
"INCFILE(VERSION)"
"\"version2\" \".h\""
```

LISTING 10 - A "Hello, world!" program

```
/* hello.c: Greet either the user or the world */
#include <stdio.h>

main(int argc, char *argv[])
{
    if (argc > 1 && argv[1] != NULL)
        printf("Hello, %s!\n",argv[1]);
    else
        printf("Hello, world!\n");
    return 0;
}
```

TABLE 4 - Trigraph Sequences

Trigraph	C Source Character
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

LISTING 11 - "Hello, World!" using Trigraphs

```
/* thello.c:    Greeting program using trigraphs */
#include <stdio.h>

main(int argc, char *argv??(??))
??<
    if (argc > 1 && argv??(1??) != NULL)
        printf("Hello, %s!??/n",argv??(1??));
    else
        printf("Hello, world!??/n");
    return 0;
??>
```

TABLE 5 - New C++ digraphs and identifiers

Token	Translation
<%	{
%>	}
<:	[
:>	]
%%	#
bitand	&
and	&&
bitor	|
or	||
xor	^
compl	~
and_eq	&=
or_eq	|=
xor_eq	^=
not	!
not_eq	!=

LISTING 12 - "Hello, World!" with the new C++ digraphs and tokens

```
/* dhello.c:    Greeting program using C++ digraphs */
#include <stdio.h>

main(int argc, char *argv<::>)
<%
    if (argc > 1 and argv<:1:> != NULL)
        printf("Hello, %s!??/n",argv<:1:>);
    else
        printf("Hello, world!??/n");
    return 0;
%>
```

LISTING 13 - Use macros and an inline function to test the standard C++ string class

```
// tstr.cpp:     Test the C++ string class

#include <iostream.h>
#include <stddef.h>
#include <cstring.h>

// Handy display macros
#define result(exp) \
    cout << #exp ":  \"" << (exp) << '\"' << endl
#define test(obj,exp) \
    exp, print(#obj ", after " #exp ":\n",obj)

// Print a string in quotes
inline void print(const char *p, const string& s)
{
    cout << p << '"' << s << '"' << endl;
}

main()
{
    string s1("Now is the time for all worthy carbon units"),
           s2 = "to come to the aid of their sector.",
           s3 = '\n',
           s4(s1);
    size_t len = s1.length();

    // Test some operators
    result(s1 == s4);
    result(s1 < s4);
    result(s1 + s3 + s2);
    test(s1,s1 += s3 + s2);
    result(s1 == s4);
    test(s1,s1.resize(len));
    result(s1 == s4);
    cout << endl;

    // Search and replace
    size_t pos = s1.find("all");
    if (pos != NPOS)
        test(s1,s1.replace(pos,3,"some"));
    pos = s1.find("worthy");
    if (pos != NPOS)
    {
        result(s1.substr(pos,5));
        test(s1,s1.insert(pos,"un"));
    }
    cout << endl;

    // More searching
    result(s1.find_first_of("aeiou"));
    result(s1.find_first_not_of("aeiou"));
    result(s1.find_last_of("aeiou"));
```

```
    result(s1.find_last_not_of("aeiou"));
    cout << endl;

    // Subscripting
    pos = s2.find_first_of('d');
    test(s2,s2[pos] = 'l');
    return 0 ;
}

/* Output:
s1 == s4:  "1"
s1 < s4:  "0"
s1 + s3 + s2:  "Now is the time for all worthy carbon units
to come to the aid of their sector."
s1, after s1 += s3 + s2:
"Now is the time for all worthy carbon units
to come to the aid of their sector."
s1 == s4:  "0"
s1, after s1.resize(len):
"Now is the time for all worthy carbon units"
s1 == s4:  "1"

s1, after s1.replace(pos,3,"some"):
"Now is the time for some worthy carbon units"
s1.substr(pos,5):  "worth"
s1, after s1.insert(pos,"un"):
"Now is the time for some unworthy carbon units"

s1.find_first_of("aeiou"):  "1"
s1.find_first_not_of("aeiou"):  "0"
s1.find_last_of("aeiou"):  "43"
s1.find_last_not_of("aeiou"):  "45"

s2, after s2[pos] = 'l':
"to come to the ail of their sector."
*/
```