

The Preprocessor

- **Objectives**
 - Understand the nature of the beast
- **Contents**
 - Macros and side-effects
 - Avoiding macro/function name conflicts
 - String support
 - The assert() macro
 - #if, #ifdef and #ifndef
 - #pragma
 - Advanced macro issues
- **Practical**
 - Use advanced preprocessor features
- **Summary**



The C preprocessor is the first stage of the code translation process. It performs various administrative tasks before the compiler starts in earnest. With Standard C has come a number of useful and powerful techniques that make the preprocessor a genuinely useful tool to the C programmer. The preprocessor is a powerful but somewhat blunt tool (it does not respect scope) that must be used carefully. Many experienced C programmers view it as a bit of wild beast (a dragon?).

In this chapter we shall investigate these new techniques and show how C programs can be made more reliable and robust through selective use of these facilities.

Overview

- The preprocessor is the first phase of compilation. Normally, its output is not seen
- Some products have standardised on the "E" flag to allow the output from the preprocessor to be examined

UNIX

```
cc -E prog.c | more
```

Microsoft

```
cl /E prog.c | more
```

GNU

```
gcc -E prog.c | more
```

- The preprocessor knows nothing about C



The C preprocessor normally runs unseen, with its result being passed directly to the compiler. Errors that may be attributed to the preprocessor are therefore difficult to spot.

Many compilers provide a command-line option to terminate compilation after the preprocessing stage. For example, Microsoft provide the /E flag to allow the output from the preprocessor to be examined. Not all manufacturers have standardised on this flag, and some compilers don't have any flag at all to allow the preprocessor output to be inspected.

One duty of the preprocessor is to remove comments. To be precise, a comment in the source code is replaced by a space in the preprocessor output file. Therefore, when viewing the preprocessor output file, the file will contain blank lines where comments have been removed. This can cause some disorientation when looking at the output. One possible solution when using the Microsoft compiler, for example, is to use the /C option (leave comments alone).

If any header files are included, these will also appear in the preprocessor output. This can result in a lot of text, so it is often advisable to pipe the preprocessor output through some form of paginating utility.

It is both an advantage and a disadvantage that the preprocessor knows none of the syntax of C at all; an advantage because we can "bend the rules", a disadvantage because compilation errors can so easily result.

Predefined macros

- The preprocessor expands several identifiers; their values are maintained as preprocessing takes place

<code>__LINE__</code>	const int	Number of line being compiled
<code>__FILE__</code>	const char*	Name of file being compiled
<code>__DATE__</code>	const char*	Compilation date "mm dd yyyy"
<code>__TIME__</code>	const char*	Compilation time "hh:mm:ss"
<code>__STDC__</code>	const int	Set to 1 by conforming compiler
<code>__func__</code>	const char*	Name of the current function C99 or later

- Examples:

```
printf("Reached line %d in file %s\n",
      __LINE__, __FILE__);
printf("You compiled this program at %s on %s\n",
      __TIME__, __DATE__);
```

A number of identifiers are maintained by the preprocessor. In fact, `__LINE__` (that's two underscores both before and after) and `__FILE__` have been unofficially available for some years. The Standards committees, realising their usefulness, made them a requirement rather than an optional extra. Strictly speaking `__LINE__` evaluates to a literal constant. The type of this literal constant depends on its value and the ranges of the various integers.

Introduced at C99 was `__func__`, which gives the name of the current function, again very useful in debugging macros.

The identifier `__STDC__` should be set by all compilers which conform entirely to C89. Unfortunately, this flag is not always set, even though compilers may conform to ANSI. Others (some new with C99) include:

```
__STDC_HOSTED__ 1
__STDC_VERSION__ 199901
__STDC_IEC_559__ 1
__STDC_IEC_559_COMPLEX__ 1
__STDC_ISO_10646__ 200009
```

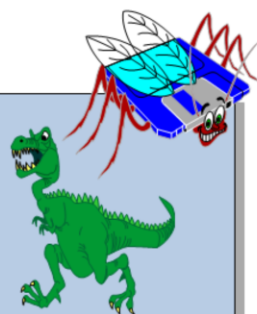
The values shown are from the GNU C compiler, version 3.2.2. Microsoft Visual C++, for example, does not conform to these standards, so none of these are set.

The `__DATE__` and `__TIME__` identifiers refer to the date and time of *compilation*, not to runtime. These symbols might be used in diagnostic messages to distinguish between different versions of an application.

Macros and side effects

- Macros are less secure than functions

- Consider the following:



```
#define MAX(a, b)      (a > b) ? a : b
#define MIN(a, b)      (((a) < (b)) ? (a) : (b))

int max(int, int);
int min(int, int);

int i = 12, j = 10, k;

k = max(i, j) * 2;      /* k becomes 24 */
k = MAX(i, j) * 2;      /* k is not 24 */
k = min(i++, j++);      /* k is 10, i is 13, j is 11 */

i = 12;                 /* reset i and j */
j = 10;
k = MIN(i++, j++);      /* k is not 10, j is not 11 */
```

Macros are prone to problems in C. For example, compare the results of a *function call* and a *macro call* to determine the maximum of two integers:

```
int i=12, j=10, k;

k = max(i, j) * 2;      // 1 - Function call
k = MAX(i, j) * 2;      // 2 - Macro call
```

Using the `max()` function, the result of the first statement will be 24 (the maximum of 10 and 12, multiplied by 2). In the second statement, however, the macro is expanded as follows and an incorrect result of 12 is obtained:

```
k = (i>j) ? i : j*2;    //Result of MAX macro expansion
k = (i>j) ? i : (j*2);  //Effective parentheses implied
```

A correct result is obtained if an extra set of parentheses is added around the expansion text of `MAX` (below). Note the parentheses around arguments.

```
#define MAX(a, b) ( ((a) > (b)) ? (a) : (b) )

k = MAX(i, j) * 2;      // Call the MAX macro as before
k = (((i) > (j)) ? (i) : (j)) * 2;  // Macro expansion OK
```

The `MIN` macro has these extra sets of parentheses and is as robust as possible. There are still situations where `MIN` will give incorrect results, specifically when the arguments passed into the macro have side-effects:

```
k = MIN(i++, j++);      // Arguments to MIN have side-effects
k = ((i++) < (j++)) ? (i++) : (j++);  // After expansion
```

The `++` operator ends up being applied twice during the expansion!

Avoiding macro/function conflicts

- It is possible to invoke a function with the same name as a macro by placing parenthesis around the name
- This persuades the preprocessor to leave well alone:

```
#define max(a, b)    (a > b) ? a : b

int (max)(int a, int b)
{
    return (a > b) ? a : b;
}

int main(void)
{
    int i = 10, j = -10, k;

    k = (max)(i, j);    /* call function */
    k = max(i, j);      /* invoke macro */

    return 0;
}
```

Footnote



Bad form to create a macro function with a lower case name

If a macro happens to be defined with the same name as a function, the function can still be called. This is done by placing the macro/function name in parenthesis. The preprocessor will only substitute a macro when the macro is followed by an opening parenthesis (intervening whitespace is allowed).

Placing parentheses around the macro name effectively removes the name from the opening parenthesis meaning the preprocessor will leave it alone.

It is valid to place parentheses around any C statement without effect, thus:

```
i = 23;
i = (23);
(i = 23);
(i = (23));
```

are all equivalent statements.

Stringizing operator

- The # operator converts macro parameters into strings
- It encloses parameters in double quotes as follows:

```
#define I_VAL(expr) printf("%s has value %d\n", #expr, (expr))
#define S_VAL(expr) printf("%s\n", #expr)
#define F_VAL(expr) printf("%s has value %f\n", #expr, (expr))
```

```
I_VAL(i * j + 5);
S_VAL(c:\autoexec.bat);
F_VAL(f / 9.6 * sin(0.25));
```

```
printf("%s has value %d\n", "i * j + 5", (i*j+5));
printf("%s\n", "c:\\autoexec.bat");
printf("%s has value %f\n",
      "f / 9.6 * sin(0.25)", (f / 9.6 * sin(0.25)));
```



If the # stringizing operator is used before a macro parameter, that parameter is turned into a string (basically, it is surrounded by double quotes by the preprocessor).

The macro:

```
PR_EXPR(expr)      printf("%s\n", #expr)
```

when invoked as:

```
PR_EXPR(i + 5);
```

produces:

```
printf("%s\n", "i + 5");
```

Anything likely to be misinterpreted, such as the backslash character, is handled correctly. For example:

```
PR_EXPR(c:\net\autoexec.bat);
```

becomes

```
printf("%s\n", "c:\\net\\autoexec.bat");
```

Token pasting operator

- The **##** operator takes the macro parameters to its right and left and pastes them together
- It cannot be the first or last item in a macro

```
#define DISPLAY(x) printf("file%sname = %s\n", #x, file##x##name)
const char * filename = "autoexec.bat";
const char * file2name = "config.sys";
const char * file3name = "command.com";
const char * file4name = "ibmbio.com";
```

```
DISPLAY(1);
DISPLAY(2);
DISPLAY(3);
DISPLAY(4);
```

```
for (i = 1; i <= 4; i++)
    DISPLAY(i);
```

The **##** operator provides a portable way of merging separate tokens into one single token. Before the invention of this operator, programmers had to use devious tricks such as the following:

```
#define glue(a token, b token) a token/**/b token
```

This depends on the preprocessor removing comments and therefore seeing the two tokens `a token` and `b token` as one. Unfortunately, instead of removing comments completely, Standard C preprocessors replace them with a single space. Thus, a Standard C compiler will never do what the programmer requires here.

Note the macro arguments are pasted at compile time, not at run time. The `for` loop above fails to print the four variables. Instead, it attempts to print the (presumably) undefined variable `fileiname` four times.

String concatenation

- **Standard C preprocessors concatenate adjacent strings in the input program**
- **Example:**

```
printf( "%s: usage\n"
        "-e    use exit status\n"
        "-l    enable logging\n"
        "-s    suppress output\n"
        "-v    verbose output\n"
        "-h    help information\n", progname);
```



- **More efficient than multiple calls to `printf`**

Adjacent strings are glued together by the compiler without spaces. Long strings may now be represented neatly without a large part of the string disappearing off the right-hand side of the screen.

String concatenation is particularly appropriate for extended `printf` strings, as shown in the above example, where a single call to `printf` is significantly more efficient than repeated calls to `printf`.

In K&R days, programmers used to use the line-splicing operator `"\` at the end of each line in the string in order to fold the next line onto the end of the current line:

```
/* How to print long strings with pre-ANSI compilers */
printf ("%s: usage\n\
-e    use exit status\n\
-l    enable logging\n\
-s    suppress output\n\
-v    verbose output\n\
-h    help information\n", progname);
```

There are no quotes around each line of text; effectively the whole text is enclosed in the outermost quotes at the beginning and end of the whole string. One of the annoying things about this old-fashioned approach is that the natural indentation of the program gets obscured by having to place subsequent lines of `printf` text at the left-hand margin in order to ensure that there is no spurious white space in the final string.

#if

- **#if can be used to test the value of preprocessor constants**

```
#if SYMBOL
    ... C code in here ...
#endif      /* SYMBOL */
```

- **Code will be included for compilation if SYMBOL is #defined as a non-zero value**
- **If SYMBOL is not defined, it defaults to zero and the code is not included for compilation**
- **Other conditions are possible:**

```
#if SYMBOL == 4
#if SYMBOL > 2
#if THIS_SYMBOL != THAT_SYMBOL
```

if you can keep your head...

#if is the most straightforward of the preprocessor's conditional constructs. A constant integer expression is evaluated. This may not include sizeof, any casts or any enumerated types. The absence of sizeof from this list is unfortunate, constructs such as these are prohibited:

```
// The following #if construct is illegal
    #if sizeof(int) == 2
...may need to use long ints here
    #elif sizeof(int) == 4
...int is fine after all...
    #endif
```

Early preprocessors would accept tokens following #endif:

```
#if      SYMBOL
    ...C code depending on SYMBOL being non zero...
#endif   SYMBOL
```

This meant if and endif could be made symmetrical. Unfortunately, the ANSI committee (and hence the ISO committee also) decided to outlaw this practice. Standard C preprocessors now require the following syntax:

```
...as above...
#endif      /* SYMBOL */
```

#ifdef and #if defined

- **#ifdef can be used to determine if a preprocessor symbol has been defined:**

```
#ifdef SYMBOL
... code depending on SYMBOL being defined ...
#endif /* SYMBOL */
```

- **This may be replaced by the defined operator**

```
#if defined(SYMBOL)
... code as above ...
```

- **This operator is more flexible than #ifdef, consider:**

```
#if defined(SYM_1) && defined(SYM_2)
... more code ...
```

`#if` is used to test the value of a preprocessor symbol, whereas `#ifdef` is used to test whether the symbol is defined at all.

Note that most compilers allow preprocessor symbols to be defined on the command line of the compiler. Most manufacturers have chosen the `/D` flag to do this. For example, the following command-line instruction and preprocessor directive have the same effect:

```
cl /DSYMBOL prog.c // Command-line switch
#define SYMBOL // Equivalent preprocessor directive
```

Symbols may be given alternative values as follows. Note that on Unix systems the quotation marks may themselves need to be in quotes.

```
cl /DSTRING_SYMBOL="string with spaces" prog.c
```

`#if` with the `defined` operator allows constructs to be written much more concisely than with `#ifdef`. Using only `#ifdef`, the construct in the slide above would have been written as:

```
#ifdef SYM_1
#ifdef SYM_2
... C code depending on SYM_1 and SYM_2 being defined
#endif /* SYM_2 */
#endif /* SYM_1 */
```

The assert macro

```
#if defined(NDEBUG)
    #define assert(test) ((void)0)
#else
    void _Assert(const char *,size_t,const char *);
    #define assert(test) \
        ((test)?(void)0 :_Assert(__FILE__, (size_t)__LINE__, #test))
#endif /* NDEBUG */
```

Possible implementation
of assert in <assert.h>
C99 versions include __func__

```
void _Assert(const char *file, size_t line, const char *test)
{
    fprintf(stderr, "%s : %lu %s --assertion failed\n",
        file, (unsigned long)line, test);
    abort();
}
```

```
assert(ptr != NULL);
assert(sizeof(int) == 4);
assert(answer == 42);
```

Example usage

The standard header file `assert.h` defines the `assert()` macro. The example shown above is one possible implementation.

The expression passed to the macro is tested. If the expression fails, an error message is printed on the standard error device. The error message consists of:

- The test expression which failed, printed as a string.

- The name of the source file which failed the assertion.

- The line number in the source file.

A C99 version would include the function name (from `__func__`)

Finally, the macro calls `abort()` raising the `SIG_ABRT` signal (which will be covered later in the course), thereby causing the death of the invoking program.

Writing a conforming implementation of `assert()` is surprisingly tricky. The macro must not directly call any library function (no standard header file is allowed to `#include` any other standard header file). Also, the macro must expand to a void expression.

#ifndef

- **#ifndef is true if the symbol is not defined**

```
#ifndef SYMBOL
    ...SYMBOL is undefined...
#endif /*SYMBOL*/
```

- **May be used to give symbols default values**

```
#ifndef SYMBOL
#define SYMBOL (42)
#endif
```

- **Deprecated to the Standard C defined operator**

```
#if !defined(SYMBOL)
#define SYMBOL (42)
#endif
```

`#ifndef` is a preprocessor operator which has largely been made obsolete by the Standard C `defined` operator. It is retained for backwards compatibility.

Defining preprocessor symbols on the compiler command-line will cause errors if defined symbols are given new values. These error messages may be avoided by nesting the definition of a variable within a `#ifndef` or `#if !defined` construct:

```
#if !defined(SYMBOL)
#define SYMBOL SOME_DEFAULT_VALUE
#endif /* not defined SYMBOL */
```

Protection from multiple inclusion

- The preprocessor can be used to guard against multiple file inclusion
- Also provides protection from re-defining struct templates and typedefs

```

#ifndef DATE_INCLUDED
#define DATE_INCLUDED
...
struct date_tag
{
    int day, month, year;
};
typedef struct date_tag date;
...
#endif /* DATE_INCLUDED */

```

date.h

Standard (C89) header files are "idempotent" – can be included any number of times in any order.

This technique is used to achieve "idempotency".

The preprocessor can be used to protect against including header files repeatedly. Any file, an include file especially, may `#include` another. Imagine `#include "first.h"` followed later by a `#include "second.h"`. If `first.h` includes `second.h`, unending error messages could result. These errors may be avoided by using the following protection mechanism:

```

#if !defined(SOME_SYMBOL)
#define SOME_SYMBOL
...
#endif /* SOME_SYMBOL */

```

If this surrounded the entire contents of `first.h`, the header file could be included any number of times without problems. On the first pass, the symbol `SOME_SYMBOL` would not yet be defined and the `#define` would be processed to define the symbol. If the header file is included thereafter, the entire contents of the header file would be skipped because the symbol is already defined.

The same thing may be done with `typedef` to avoid defining the new type twice (actually `typedef` merely creates new names for existing types):

```

#if !defined(MY_COMPLEX_TYPE)
#define MY_COMPLEX_TYPE
typedef double *(*My_Complex_Type[34])[80];
#endif /* !defined(MY_COMPLEX_TYPE) */

```

The same technique may be used where it is possible for a preprocessor symbol to be defined more than once, since re-defining a preprocessor symbol is an error.

#pragma

- **#pragma is a Standard C directive which may affect certain parts of the compiler or optimiser**
- **Pragmas are compiler specific - there is no standard that specifies what pragmas should be supported by ANSI C compilers**
 - a pragma may have different meanings on different compilers!
- **Unrecognised pragmas are ignored**

```
#pragma check_pointer(off)
for (p = a, end = &a[size]; p != end; ++p)
    *p = 0;
#pragma check_pointer(on)
#pragma tic
```

#pragma is a C89 (ANSI) introduction. Although the directive is picked up by the preprocessor, it may affect either the compiler or the optimiser and the way it produces code. Pragmas provide a way of guaranteeing that code is compiled with certain options in force, rather than leaving things up to the user (who may not know what flags to provide to the compiler).

Unfortunately, there is no standard list of pragmas (but see next slide). Thus, pragmas which are placed in C code to control compilation under a Microsoft compiler, for instance, may have a disastrous effect with another vendor's compiler.

As an example, the following list shows some of the more commonly-used pragmas supported by the Microsoft compiler:

#pragma check_stack

Enables/disables stack checking

#pragma function

Specifies that calls to a function will be conventional

#pragma intrinsic

Specifies that calls to a function will be expanded inline.

Note: C99 supports real inline functions, but Microsoft do not.

#pragma pack

Controls structure packing (also supported by GNU)

Pragma

- C99 introduced some new pragma features:
- `_Pragma ("string literal")` as an alternative to `#pragma`

```

#pragma pack (1)
typedef struct
{
    char a;
    int b;
    char c;
} MESSY;
#pragma pack (0)

```

C89

```

_Pragma ("pack (1) ")
typedef struct
{
    char a;
    int b;
    char c;
} MESSY;
_Pragma ("pack (0) ")

```

C99

- Pragmas prefixed **STDC** are now reserved
 - New standard pragmas for complex mathematics:
 - **STDC** FP_CONTRACT, FENV_ACCESS, CX_LIMITED_RANGE

The `#pragma` directive has an alternative form in C99, `_Pragma (string)`. The pragma value must be enclosed in parentheses and double quotes.

For years we have been able to give a golden rule that there are no standard pragmas, well now there are! The three standard pragmas are rather obscure:

Floating point contracted expressions are evaluated as if they were atomic, that is no exceptions will be raised if rounding errors or overflow occurs. The standard pragma `FP_CONTRACT` allows this to be switched off (or on).

Normally floating-point access flags are tested on each FP operation, but the `FENV_ACCESS` pragma allows the programmer control over this and allow optimisation.

`CX_LIMITED_RANGE` allows the compile to optimise its use of complex formula, on the assumption that they are safe (!).

Expect more standard pragmas in future versions.

Trigraphs and digraphs

- **C89 (ANSI) preprocessors support the substitution of trigraphs as follows:**

??<	{	??([??=	#
??>	}	??)]	??/	\
??~	~	??!		??'	^

- Although not elegant, they are vital for programmers with non-English keyboards
- **C95 attempted to make things simpler (?) by adding digraphs to C punctuation:**

<%	{	<:	[?:	#
%>	}	:>]	%%:	##

- Trigraphs are preprocessor symbols, digraphs are not

Many keyboards do not have the keys necessary to enter C programs. For instance, curly braces are missing from a surprising number of keyboards. It is almost impossible to imagine the difficulty of programming on terminals such as these.

An additional problem is that even though a program may have been entered on a US or UK keyboard, for example, some printers do not have the character set to print it properly.

The Standard uses a table constructed out of the ISO 646-1983 Invariant Code Set. This is a set of characters that all terminals and printers are guaranteed to contain.

Trigraphs have been under-used since they were introduced. One reason was because they are preprocessor symbols, which can lead to peculiar effects. C95 (and C99) include so-called **digraphs** as part of the C punctuation, so they have the same 'status' as the characters they replace. Unfortunately, it is possible to mix trigraphs and digraphs in the same code!

Trigraph exercise


```
??=include <stdio.h>

void validate_exams(int a ??( ??), size_t n)
??<
    size_t i;

    for (i = 0; i < n; ++i)
    ??<
        int mark = a ??( i ??);

        if (mark < 0 ??!?! 100 < mark)
            printf( "Invalid ??/n" );

    ??>
??>
```



The example shown above is a piece of code which uses trigraphs. Note that all trigraphs are substituted as expected, including the one present in the `printf` string. The string printed is actually

```
Invalid\n
```

In fact, it is rather difficult to prevent the preprocessor from substituting trigraphs within strings. What if you really do want to print out `??/` as three literal characters? This is possible by concatenating string literals, as follows:

```
printf ("??"  "/" );
```

Substitution of trigraphs occurs before the concatenation of string literals during the preprocessor phase.

Passing types into a macro

- Since the preprocessor does not understand C, the following is possible:

```
#define SWAP(type, var1, var2) \  
{                               \  
    type  _temp_ = var1;       \  
    var1 = var2;               \  
    var2 = _temp_;             \  
}  
  
SWAP(int, i, j)  
SWAP(long double, ld1, ld2)  
SWAP(date, s1, s2)
```

Since the C preprocessor is unaware of C syntax, some of the "normal" rules of C can be broken in a macro. For instance, it is not valid C to pass a data type such as `int` into a function; only *values* can be passed into functions.

However, it is quite valid to pass a type into a macro and this can be used to create a macro that works with any data type, as shown above.

Variadic macros

- In C89, the preprocessor required fixed numbers of parameters to macros.
 - A truly variadic macro had to be faked
- In C99 Variadic Macros are supported
 - Declare macro parameters as ellipsis (...)
 - Use `__VA_ARGS__` to substitute the arguments

```
#define err_print(...) fprintf(stderr, "ERR: " __VA_ARGS__ );  
  
...  
  
err_print ("Value of ssl is: %s\n", ssl);
```

```
ERR: Value of ssl is: xx
```

In C89 (ANSI) C, emulating a variadic (variable argument list) macro was torturous, and usually meant multiple definitions. True variadic macros were introduced with C99, and use a method similar in some ways to variadic functions (see later).

In the macro definition the macro parameter list is defined as (...), and where the variable arguments are to be substituted we use `__VA_ARGS__` (that's two underscore characters on each side of the `VA_ARGS`). Other arguments can be included, but they must precede the ellipsis, for example:

```
#define pr(handle, ...) fprintf(handle, ": " __VA_ARGS__ );
```

Preprocessor miscellany

- Removing commented code

```
...
    if (argv[n][1] == 'a')    /* Give all matches */
        all++;
    #if 0
        else if (argv[n][1] == 's')    /* Silent mode */
            silent++;
    #endif
        else if (argv[n][1] == 'v')    /* Verbose mode */
            ...
```

- The **#error** directive terminates the preprocessor

```
#define OPTS    WIN32 + VMS + UNIX + OS2
#if OPTS > 1
    #error Choose one of Win32, VMS, UNIX or OS2
#endif
```

The preprocessor provides an effective way of removing C code containing comments. Such code may not simply be "commented out" since C comments cannot in general be nested.

`#if 0` provides an effective way of doing this. The preprocessor evaluates 0 as false and subsequent code is not passed on to the compiler.

This method is preferable to the `#if NOTDEF` strategy used by some programmers. Although `NOTDEF` is a reasonably self-explanatory symbol, it is possible to define it with:

```
#define NOTDEF 1
```

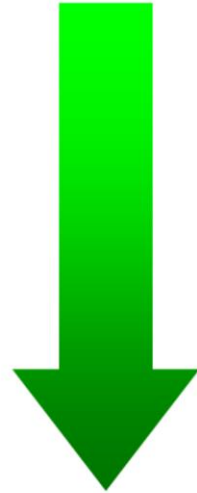
This is much safer, since an attempt to define the text string "0" as "1" will certainly fail.

A problem with this method is that colour sensitive editors cannot detect that some lines will not be compiled, so they appear as ordinary code. C99 supports the C++ style comment prefix, `//`, so a happier solution is to use the C++ style, then the 'old' C style comment delimiters for large blocks of code.

C89 (ANSI) preprocessors support the `#error` directive for terminating the preprocessor. The error message is written to the standard error device, and the compilation terminates.

Sequence of events

- **The preprocessor carries out its various tasks in the following order:**
 - Trigraphs are substituted
 - Lines whose last character is '\ ' are spliced together
 - Tokens are built
 - Comments are removed
 - Preprocessor directives (#include, etc.) are executed
 - Macros are expanded
 - Escape sequences ('\n', '\t', etc.) are substituted both in character constants and in string literals
 - Adjacent strings are glued together



The order in which the preprocessor completes various stages of preprocessing is important. The fact that trigraphs are substituted *before* strings are glued together implies statements such as the following work correctly:

```
printf ("What ??" " !");    // Original statement
```

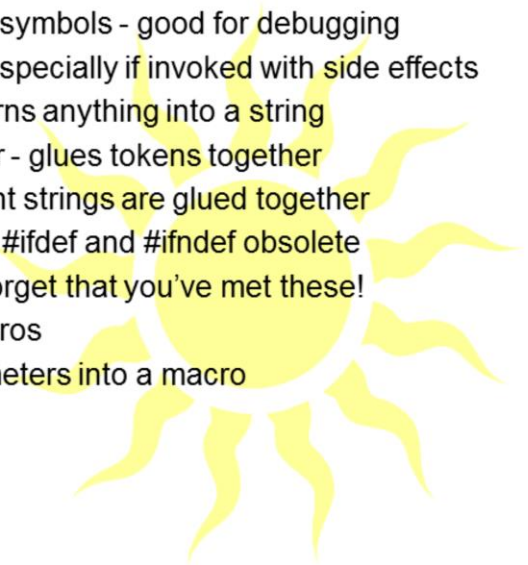
```
What ??!                    // Result of printf
```

If the `printf` text had been enclosed in a single string, the results would have been somewhat different:

```
printf ("What ??!");    // Alternative: single string
```

```
What |                      // Trigraph substitution now occurs
```

Summary

- **In this chapter we covered:**
 - The `__LINE__` and `__FILE__` symbols - good for debugging
 - Macros - can be dangerous, especially if invoked with side effects
 - The `#` stringizing operator - turns anything into a string
 - The `##` token-pasting operator - glues tokens together
 - String concatenation - adjacent strings are glued together
 - The `defined` operator - makes `#ifdef` and `#ifndef` obsolete
 - Trigraphs - you may want to forget that you've met these!
 - The passing of types into macros
 - The passing of multiple parameters into a macro
- 

In this chapter, we have studied many of the new preprocessor facilities introduced to C by the standards committees. A number of new preprocessor symbols have been added, including `__LINE__` and `__FILE__`, and a number of new directives have been incorporated, such as `#error`, to enhance the range of techniques possible.

We have also seen a variety of subtle and useful constructs that can be used to produce more efficient and robust code. Portability between K&R C and Standard C has been discussed, and we have presented a number of preprocessor mechanisms that can be used to make a difficult task a little easier.

