

# Preprocessing

1

C Foundation

# Translation Phases

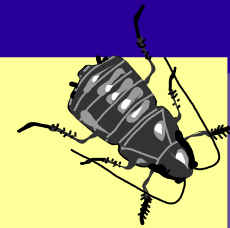
- 1. multibyte character mapped, trigraphs replaced
- 2. \ newline deleted to form logical lines
- 3. decomposed into preprocessing tokens
- 4. preprocessing directives executed (#includes phases 1-4 recursively)
- 5. source character set and escape sequences mapped
- 6. adjacent string literals are concatenated
- 7. preprocessing tokens converted to tokens, translation unit is semantically analysed and translated

- phase 6:
  - adjacent string literals are concatenated

```
const char * lines[] =  
{  
    "the boy stood on the burning deck",  
    "his hearts was all a quiver",  
    "he gave a cough, his leg fell off",  
    "and floated down the river"  
};  
assert(sizeof(lines) / sizeof(lines[0]) == 4);
```

commas

```
const char * lines[] =  
{  
    "the boy stood on the burning deck",  
    "his hearts was all a quiver",  
    "he gave a cough, his leg fell off"  
    "and floated down the river"  
};  
assert(sizeof(lines) / sizeof(lines[0]) == 3);
```



no  
comma

4

# function macros

## a function-like macro accepts arguments

```
# define identifier( identifier-listopt )  
    pp-tokensopt
```

space here is not allowed

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))  
func(MAX(precision, delta + epsilon));
```

```
func((precision) > (delta + epsilon)  
    ? (precision) : (delta + epsilon));
```

# 5 function macros

- can accept a variable number of arguments
- `__VA_ARGS__` expands to the elided arguments

space here is not allowed

```
# define identifier( identifier-listopt, ... )  
    pp-tokensopt
```

```
#define DEBUG(...) fprintf(stderr, __VA_ARGS__ )  
DEBUG("error: %s", message);
```

```
fprintf(stderr, "error: %s", message);
```

c99

# # operator

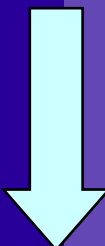
6 · the # operator converts its argument to a string literal

- if the argument is a string literal or character constant \ is inserted before " and \

```
#define REPORT(test, ...) \
    ((test) \
     ? puts(#test) \
     : printf(__VA_ARGS__))

REPORT(x > y, "x is %d but y is %d", x, y);
```

\ phase 2 logical lines



```
((x > y)
 ? puts("x > y")
 : printf("x is %d but y is %d", x, y);
```

# ## operator concatenates two arguments

```
#define DEBUG(s, t) printf("x" # s "= %d, " \
                          "x" # t "= %s", \
                          x ## s, x ## t)

DEBUG(1, 2);
```

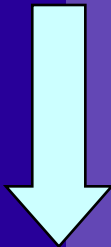
```
printf("x" "1" "= %d, " "x" "2" "= %s", x1, x2);
```

```
printf("x1= %d, x2= %s", x1, x2);
```

phase 6 string literal concatenation

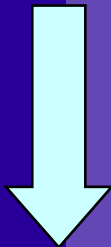
• you can define an identifier as a macro name with a replacement list

```
# define identifier pp-tokensopt
```



```
#define BUFFER_SIZE (100)  
char buffer[BUFFER_SIZE];
```

```
char buffer[(100)];
```



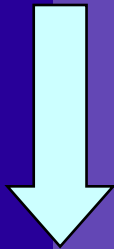
```
#define printf my_printf  
printf("error: %s", message);
```



```
my_printf("error: %s", message);
```

• you can define and undefine an identifier as a macro name

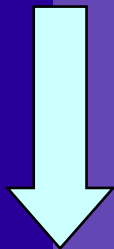
```
# define identifier  
# undef identifier
```



```
#define BUFFER_SIZE /*nothing*/
```

```
char buffer[BUFFER_SIZE];
```

```
char buffer[];
```



```
#define BUFFER_SIZE (100)
```

```
#undef BUFFER_SIZE
```

```
char buffer[BUFFER_SIZE];
```

```
char buffer[BUFFER_SIZE];
```

- **\_\_func\_\_**
  - the name of the current function (a string literal)
- **\_\_FILE\_\_**
  - the name of the current source file (a string literal)
- **\_\_LINE\_\_**
  - the line number of the current source line (an integer constant)
- **\_\_DATE\_\_**
  - the date of translation of the preprocessing translation unit (a string literal)
- **\_\_TIME\_\_**
  - the time of translation of the preprocessing translation unit (a string literal)

sections of code can be conditionally included/excluded from preprocessing (and hence from translation)

```
# if constant-expression  
# elif constant-expression  
# else  
# endif
```

← #elif == #else #if

```
#if VERSION == 1  
# define INCFILE "version1.h"  
#elif VERSION == 2  
# define INCFILE "version2.h"  
#else  
# define INCFILE "versionN.h"  
#endif
```

```
#if 0  
...  
#endif
```

← how to exclude code when the excluded code contains `/*comments*/`  
(remember `/* comments */` do not nest)

the `#if` expression can determine if a macro token has been `#defined` or not

```
# if defined(identifier)
# if !defined(identifier)
```

```
# ifdef identifier
# ifndef identifier
```

equivalent

This is the idiomatic way to make header files idempotent<sup>†</sup>

outer/table.h

```
#ifndef OUTER_TABLE_INCLUDED
#define OUTER_TABLE_INCLUDED
...
...
...
#endif
```

in a single translation make sure every source file has a unique token

<sup>†</sup>an idempotent operation produces the same results no matter how many times it is performed

## the commonest directive

- #include X is replaced by the entire contents of X
- >50% of compilation is typically for #inclusions

h-char == any character except > or newline

```
# include < h-chars >
```

q-char == any character except " or newline

```
# include " q-chars "
```

much rarer third form must expand to <> or ""

```
# include pp-tokens
```

is this a tab character?

"" for local headers

```
#include "outer\table.h"
```

this is *not* a string literal

<> for system headers

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

rarer

```
#include INCFILE
```

# #include local header before system headers

- this helps to ensures they don't accidentally compile because of a previous #include
- a source file should #include its own header before any other header
- consider checking each header compiles individually as part of the build

eg.h

```
FILE * eg(void);  
...
```

this should #include <stdio.h> itself

eg.c

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



eg.c

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



## · the #error directive

- issues the specific diagnostic message
- terminates the translation as a failure
- useful when conditional

```
# error pp-tokensopt
```

↑ diagnostic message

```
#if TARGET == 1
#  define INCFILE "version1.h"
#elif TARGET == 2
#  define INCFILE "version2.h"
#else
#  error "TARGET must be 1 or 2"
#endif
```

- causes implementation-defined behaviour

```
# pragma pp-tokensopt
```

- also available via the `_Pragma` operator

```
_Pragma ( string-literal )
```

```
#pragma ivdep /* vectorization hint */  
while (n-- > 0)  
    ...
```

```
#define VECTOR_HINT _Pragma("ivdep")  
  
VECTOR_HINT  
while (n-- > 0)  
    ...
```

- macro names should never use lowercase
  - uppercase and underscore only

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



this looks like a function call (with a sequence point) but it's not

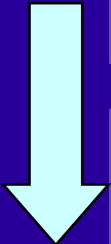
```
max(delta, precision);
```

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



this looks like a function macro

```
MAX(delta, precision);
```



# single expression function macros

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

don't include a trailing semi-colon

better

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

put each argument in parentheses

better

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

put the whole replacement text in parentheses

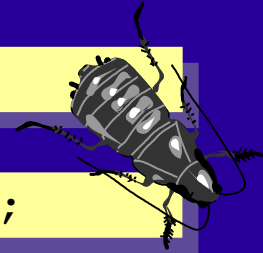
better

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

```
f(MAX(n++, limit));
```

```
f(((n++) > (limit) ? (n++) : (limit)));
```

beware of repeated side-effects



- macros bigger than a single expression...
  - can easily foul up their surrounding context

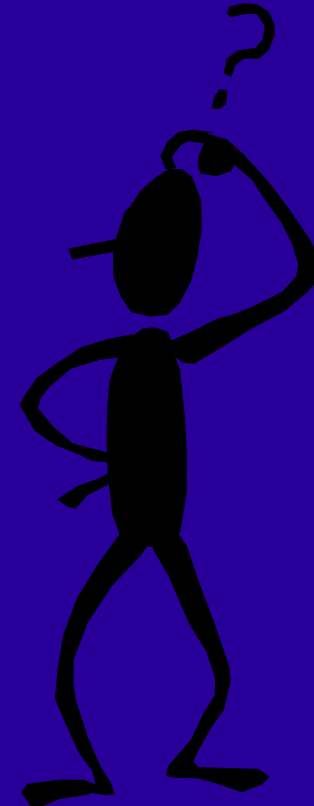
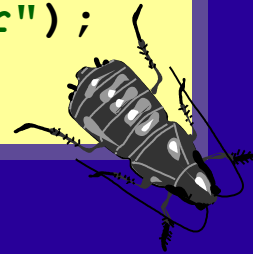
```
#define TRACE(msg)  if (dbg_mode) puts(msg)
```



```
if (whatever())  
↓ TRACE("whatever");  
else  
  ...
```



```
if (whatever())  
  if (dbg_mode)  
  ↓ puts("whatever");  
  else  
    ...
```



- rephrase the logic in a single expression

```
#define TRACE(msg) \
    ((void) (dbg_mode && puts(msg)))
```

```
if (whatever())
    TRACE("whatever");
else
    ...
```



```
if (whatever())
    ((void) (dbg_mode && puts("whatever")));
else
    ...
```


## use the do-while(0) trick

```
#define TRACE(msg) do { \
    if (dbg_mode) \
        puts(msg); \
} while (0)
```

don't include a trailing semi-colon



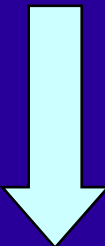
```
if (whatever())
    TRACE("whatever");
else
    ...
```



```
if (whatever())
    do {
        if (dbg_mode)
            puts("whatever");
    } while (0);
else
    ...
```

- be wary of the preprocessor
  - it knows practically nothing about C
  - it silently changes the source being compiled
  - header guards and includes are unavoidable
  - but for other #directives consider alternatives
    - function-like macro → inline function
    - object-like macro → const variable
    - object-like macro → enumerator

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



```
inline int maxi(int lhs, int rhs)  
{  
    return lhs > rhs ? lhs : rhs;  
}
```

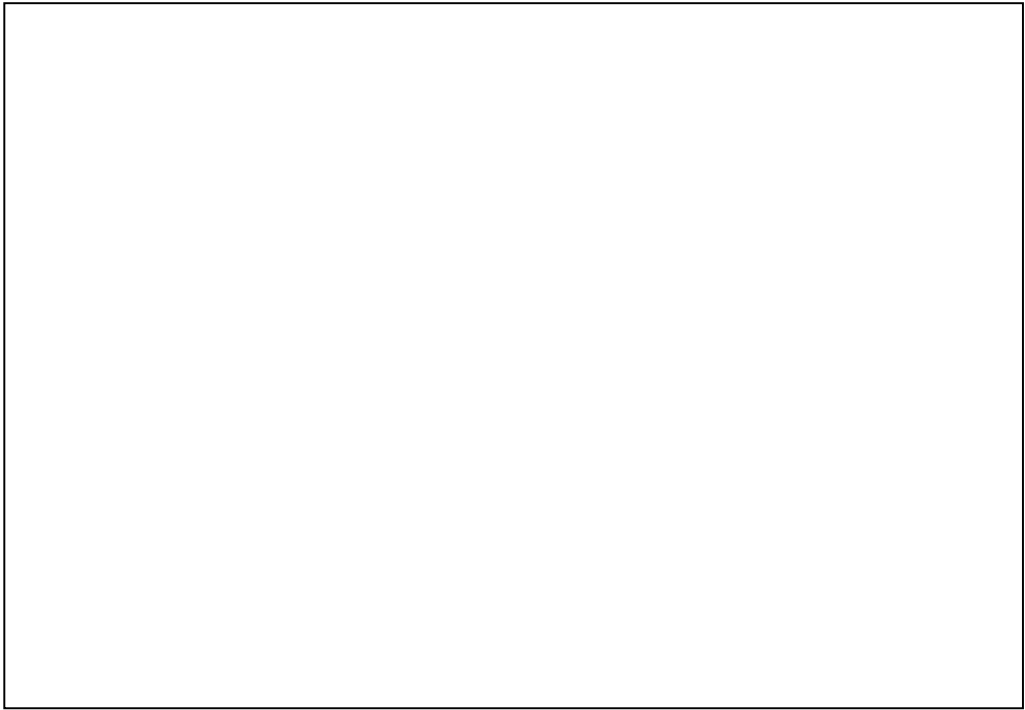


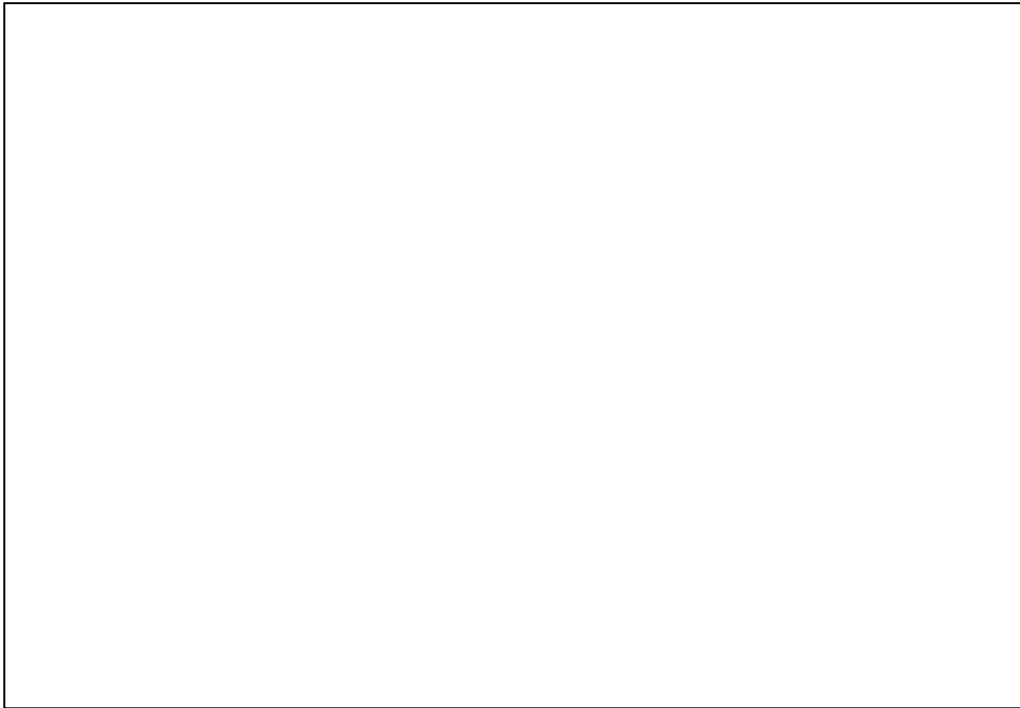
"I'd like to see Cpp [the C pre processor] abolished."  
Bjarne Stroustrup.

The Design and Evolution of C++. p426

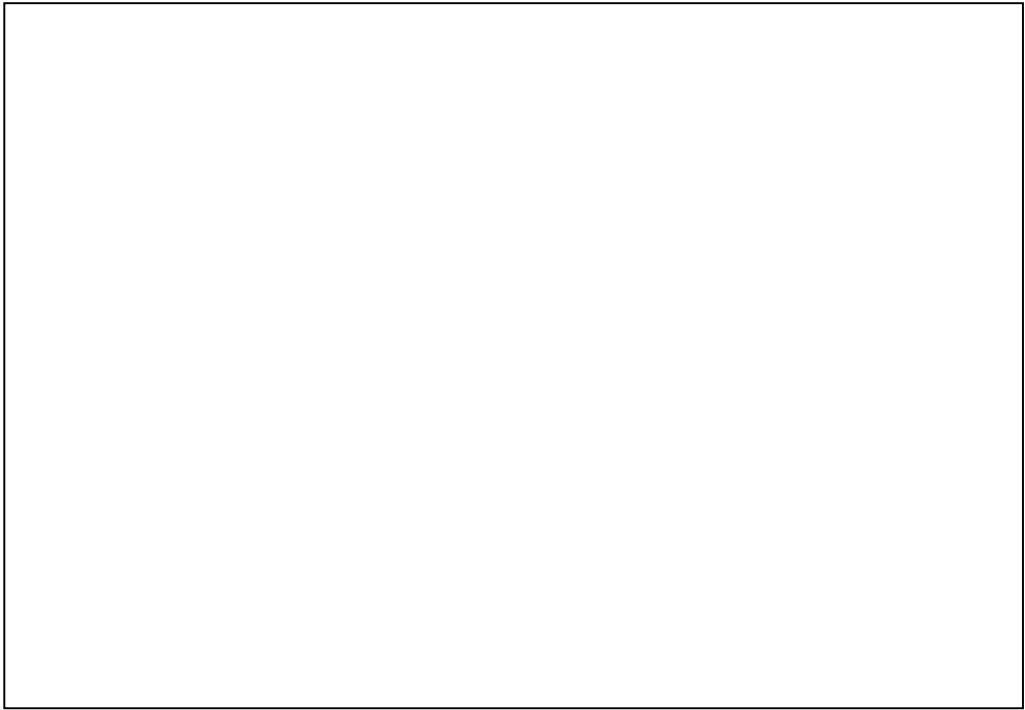
"In retrospect, maybe the worst aspect of Cpp is that it has stifled the development of programming environments for C."  
Bjarne Stroustrup.

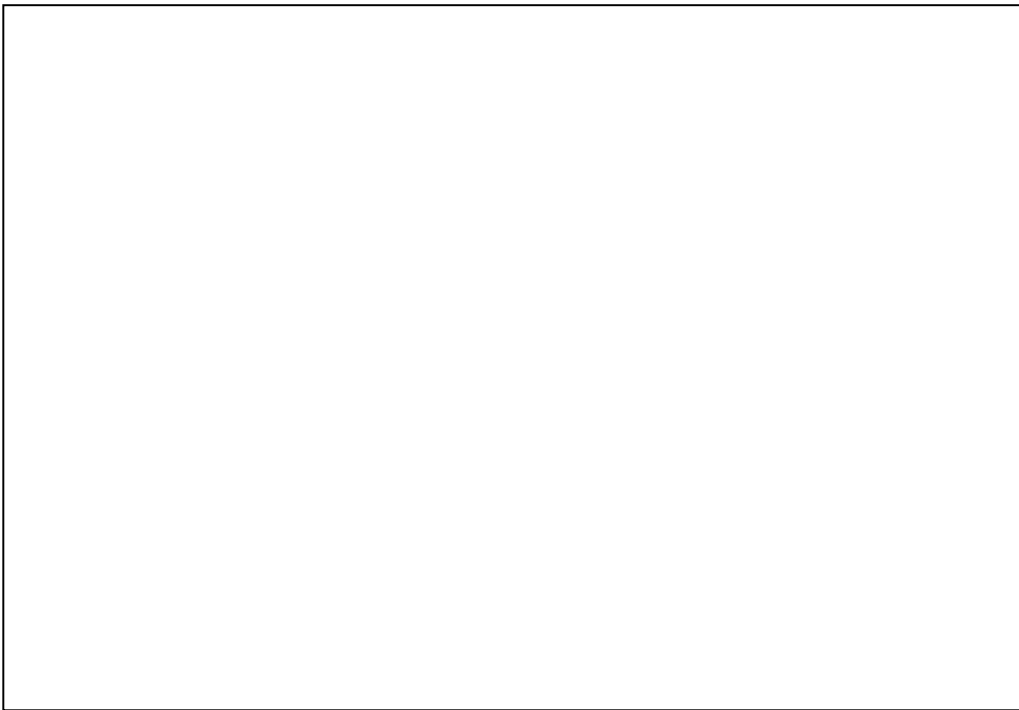
The Design and Evolution of C++. p424





A compiler (or any translator) is of course free to merge these logical phases into a single physical phase to improve performance as long as it maintains their semantics; that is, as long as it appears logically to operate as distinct phases.





Putting a space between the identifier and the left parenthesis creates an object-like macro rather than a function-like macro. In other words, the parentheses and the identifier list become part of the replacement list. For example:

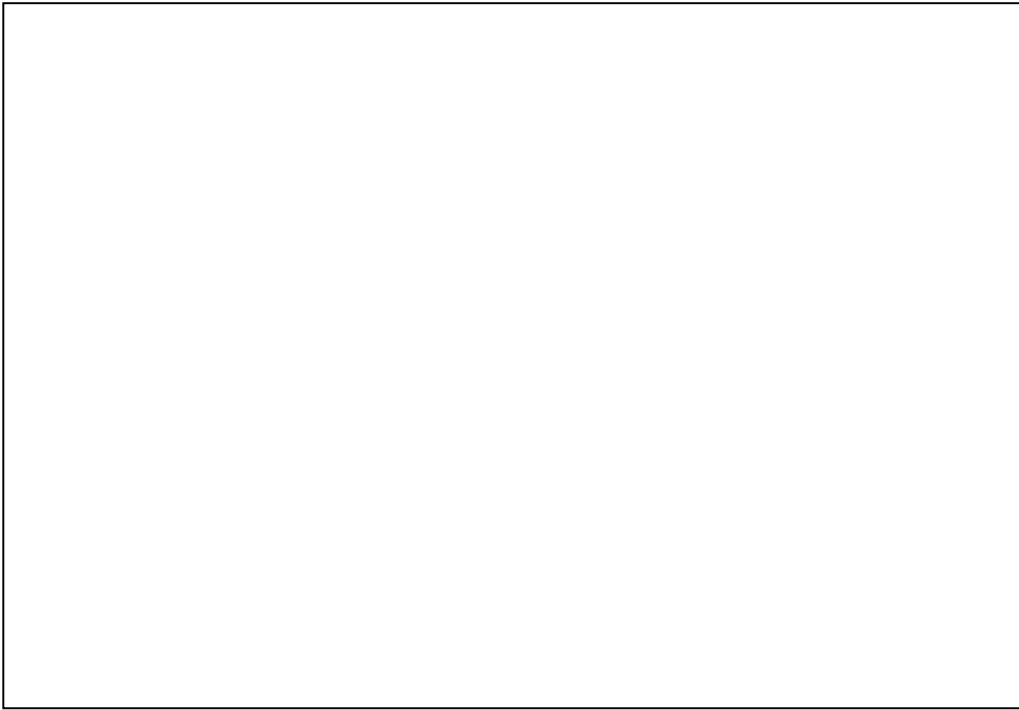
```
#define MAX (a,b) ((a) > (b) ? (a) : (b))
```

would cause the following

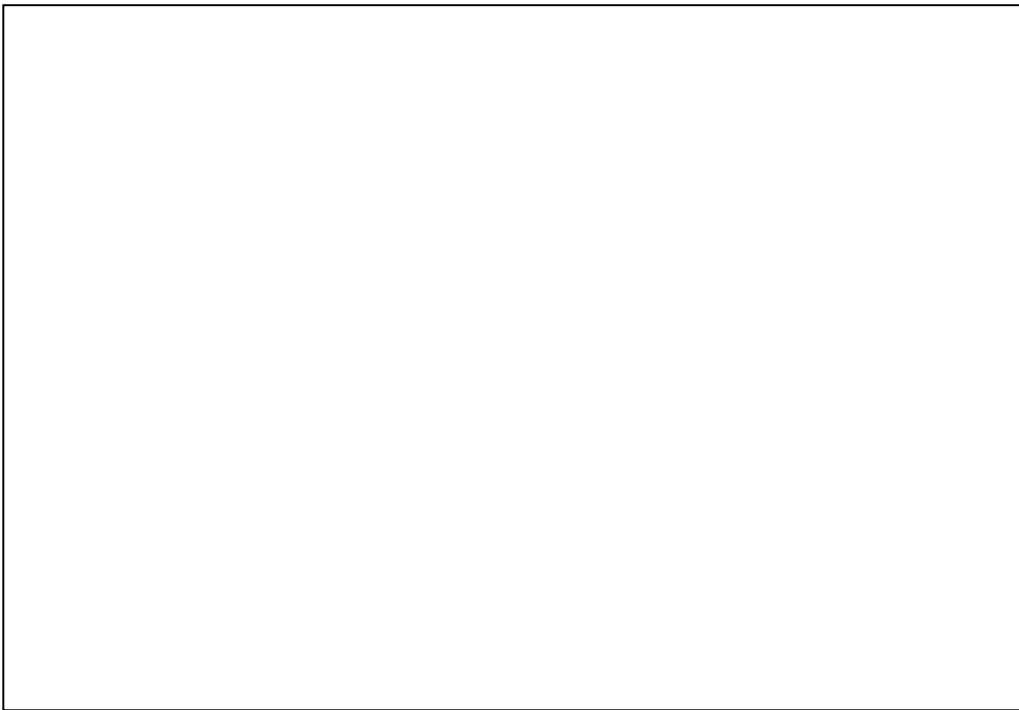
```
MAX(2,3)
```

to be expanded to

```
(a,b) ((a) > (b) ? (a) : (b))(2,3)
```



The variable argument list feature for macros is new in C99.



The intent of the `#operator` is to provide a string that contains the expression it was created from *as-the-developer-sees-it*. For example:

```
#define STR(exp) #exp
const char * s = STR( "x\n" );
```

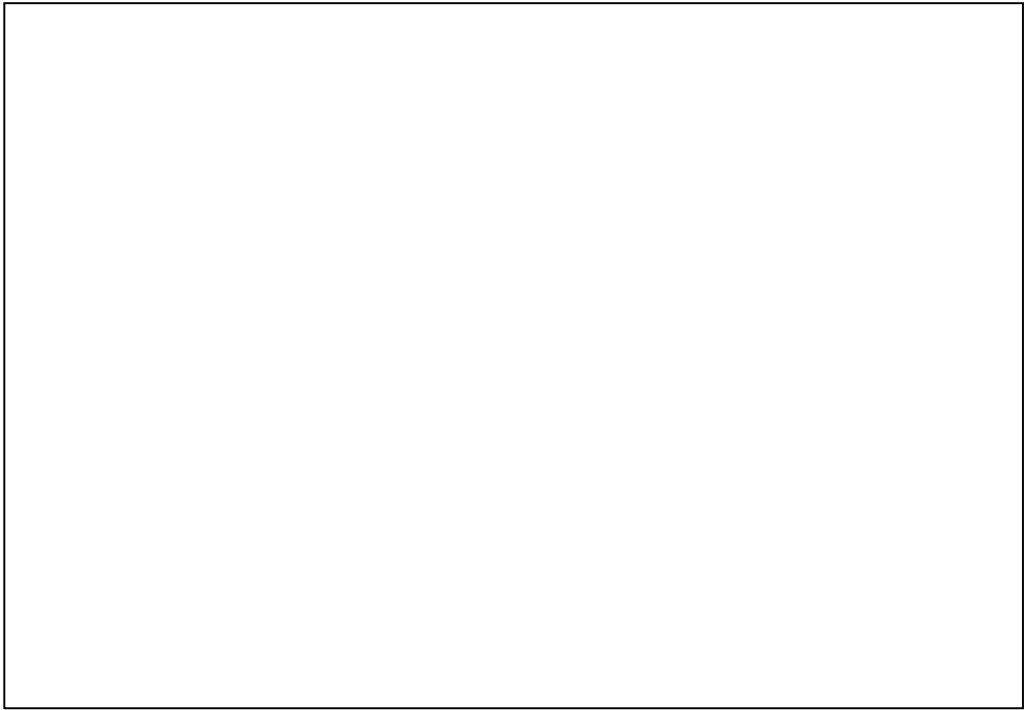
In this example, the string `"x\n"` looks visually as though it contains three characters: `x`, `\`, and `n`. Of course in a true string-literal the `\` would be interpreted as an escape character and the string would actually only contain two characters: `x` and `\n`. However, in the example `"x\n"` is stringized via the `STR` macro. The stringizing operator inserts a `\` character before any existing `\` character. The preprocessed version becomes:

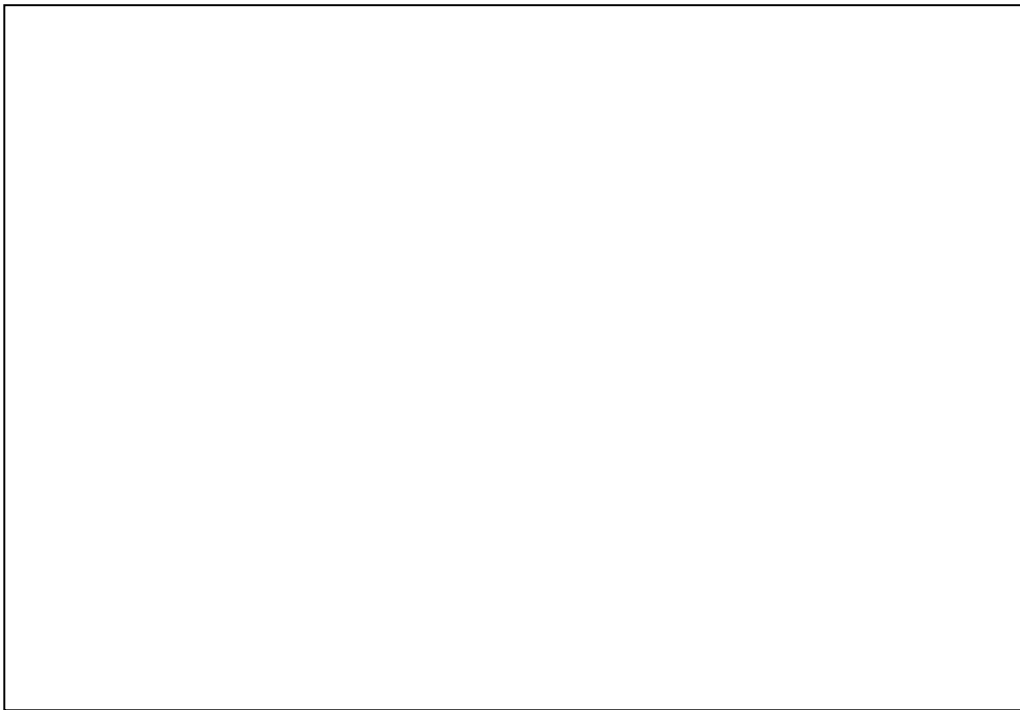
```
const char * s = "x\\n";
```

which is equivalent to:

```
const char s[] = { 'x', '\\', 'n', '\0' };
```

If we now print `s` (for example using the `puts` function) then the output will reflect the original expression *exactly* as the develop sees it.





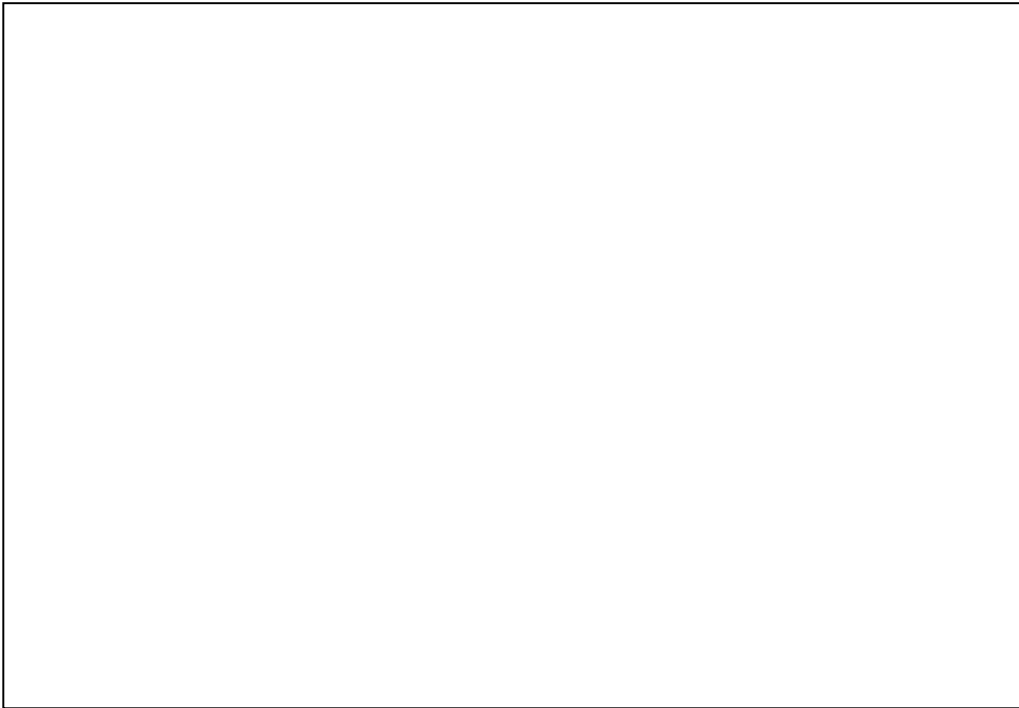
The `-D` option is universally used as a command line option to define an object-like macro. For example

```
>gcc -DABC wibble.c
```

effectively places the line

```
#define ABC
```

at the start of the file `wibble.c`



It is not an error to `#undef` an identifier that has not been defined.

The `-D` option is universally used as a command line option to define an object-like macro. For example

```
>gcc -DABC wibble.c
```

effectively places the line

```
#define ABC
```

at the start of the file `wibble.c`

The C standard reserves several macro names that may not be used as the subject of a `#define` or `#undef`:

```
defined, __DATE__, __FILE__, __LINE__, __STDC__, __STDC_HOSTED__,  
__STDC_VERSION__, __TIME__.
```

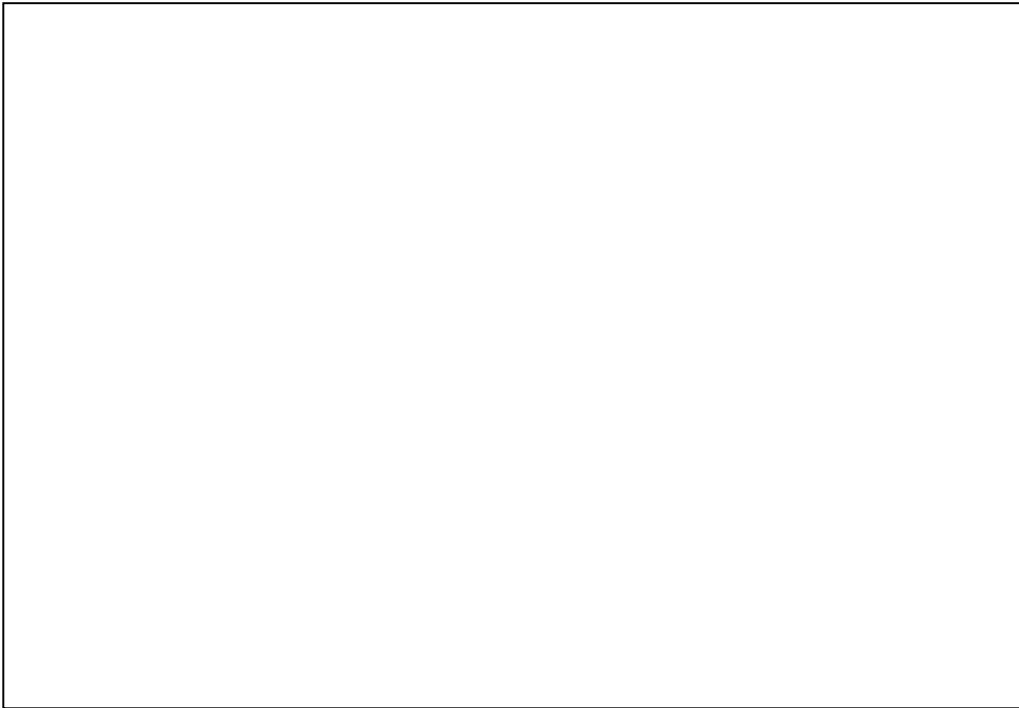
For example:

```
#define __FILE__ "wibble.h" /* undefined behavior */
```

The C standard also reserves macro names starting with an underscore followed by an uppercase letter or a second underscore.

For example:

```
#define _ABC "wibble.h" /* undefined behaviour */
```

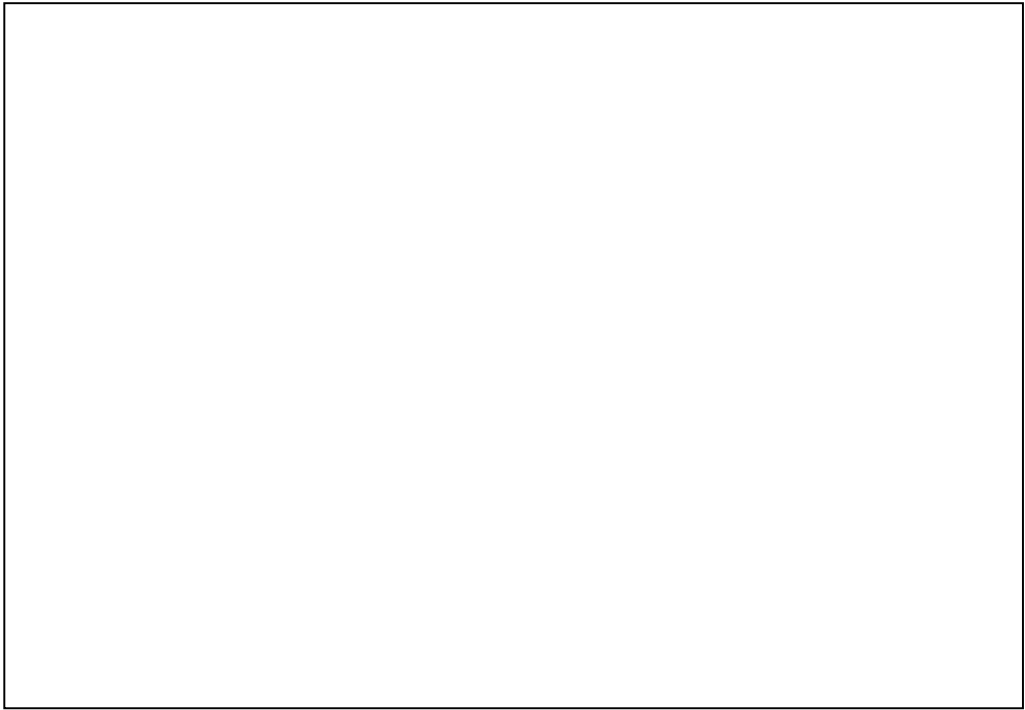


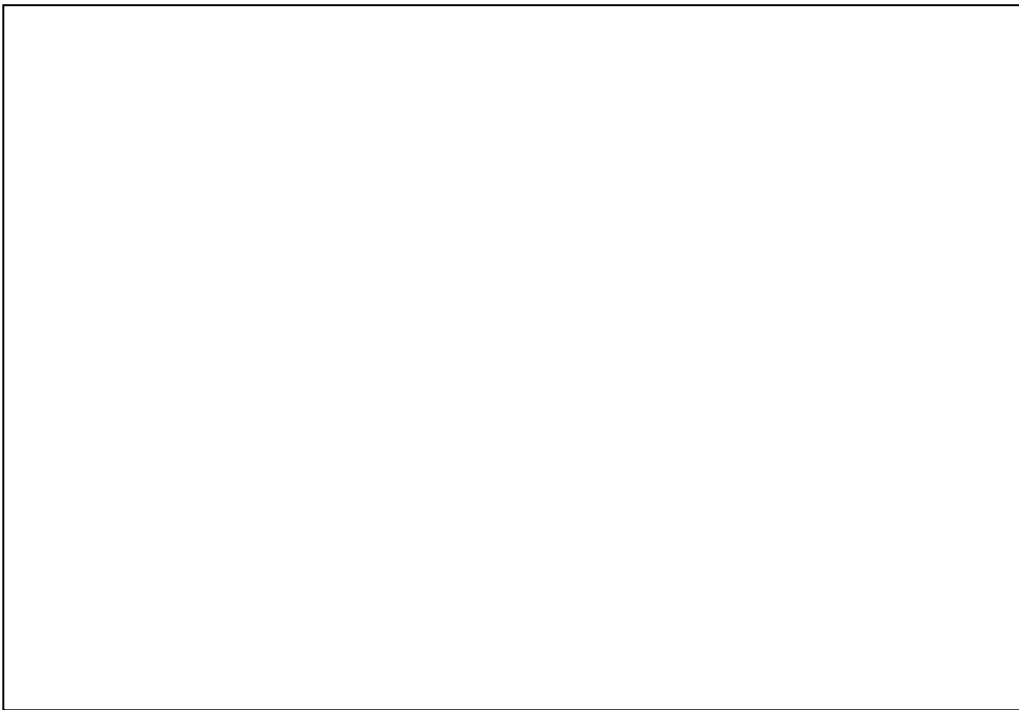
The values of the predefined macros `__DATE__` and `__TIME__` remain the same throughout the translation unit.

There are a few other predefined macros:

`__STDC__` defined to be 1 on a conforming implementation

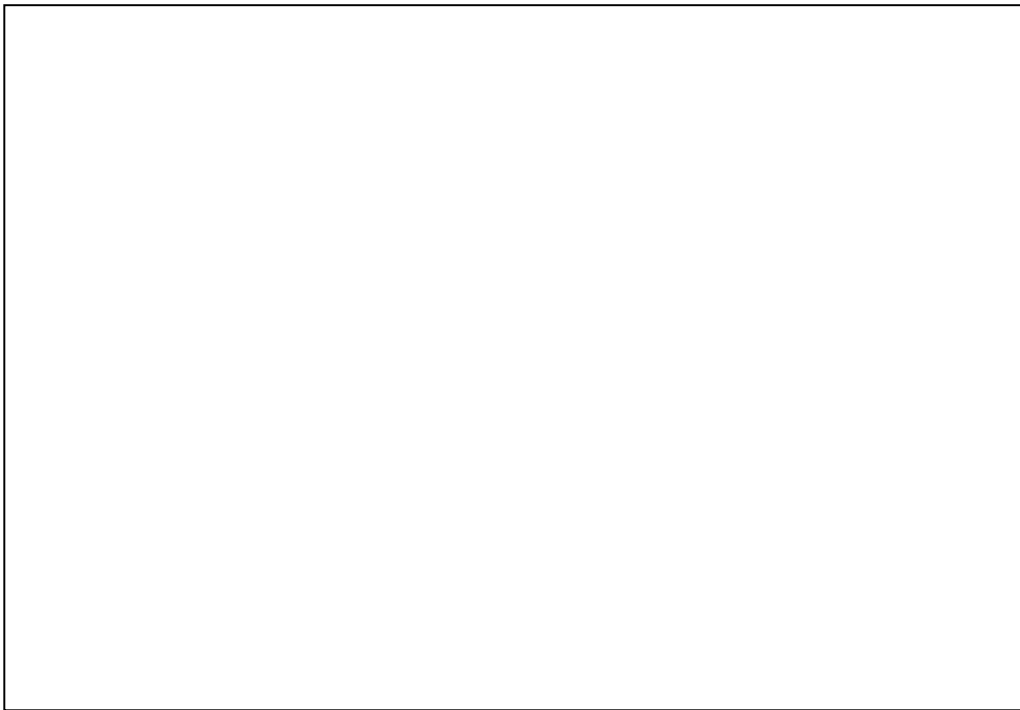
`__STDC_HOSTED__` defined to be 1 if the implementation is a hosted implementation or 0 if it is not.





Another technique developers sometimes use to ensure their header files are idempotent is to include the current date/time as part of the token. For example:

```
#ifndef TABLE_2008_April_10th_10_23AM  
#define TABLE_2008_April_10th_10_23AM  
...  
#endif
```



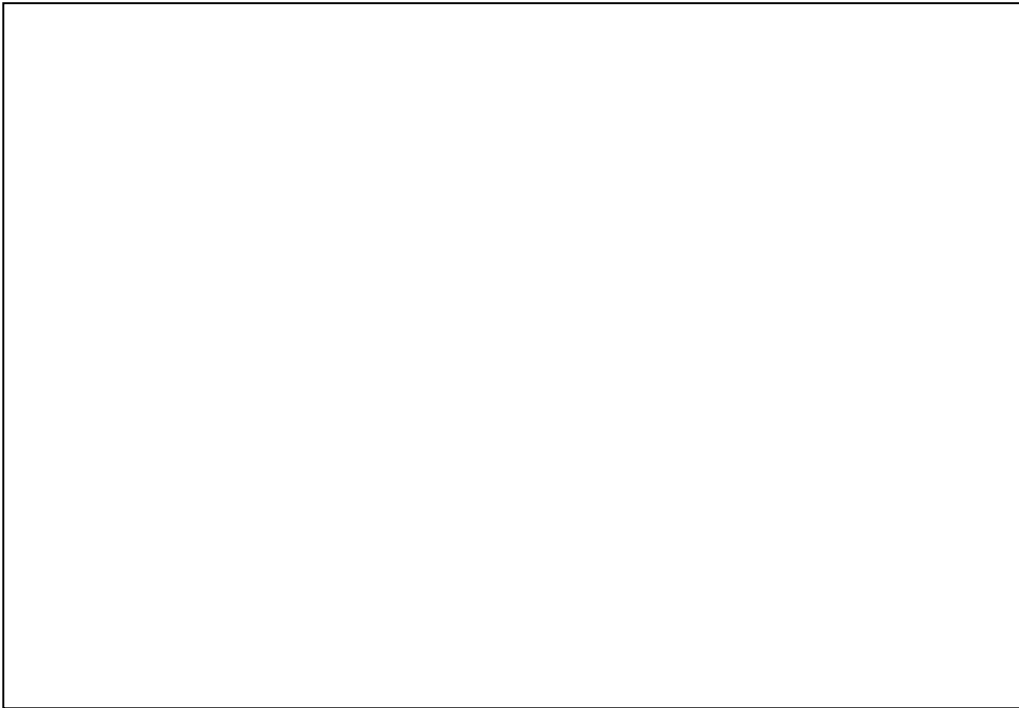
The locations where the translator searches for the files named between the `<>` or `""` characters are implementation defined.

If the location search fails for the "q-chars" version the translator will try again as if it read `<q-chars>`.

The sequence `\t` (for example) in a `#include filename` may or may not be a tab character. It depends on the implementation-defined mapping of the physical source file characters in phase 1. The filename can be made more portable by using a forward slash instead of a backslash.

```
#include "outer/table.h"
```

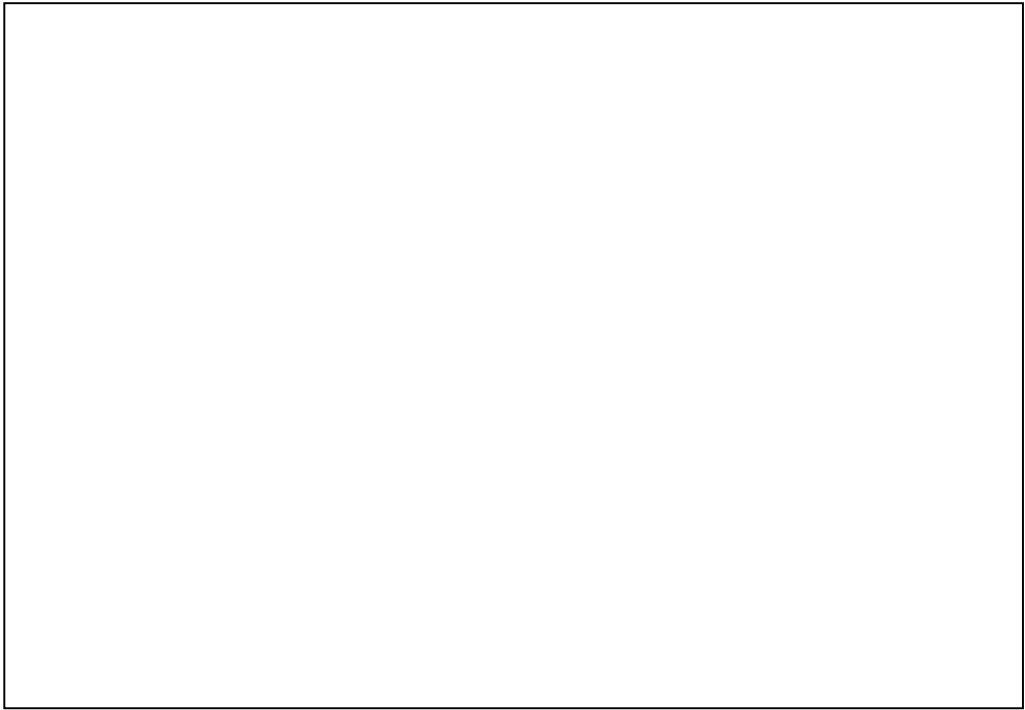
The use of the third kind of `#include` is less common but is quite often used by frameworks.

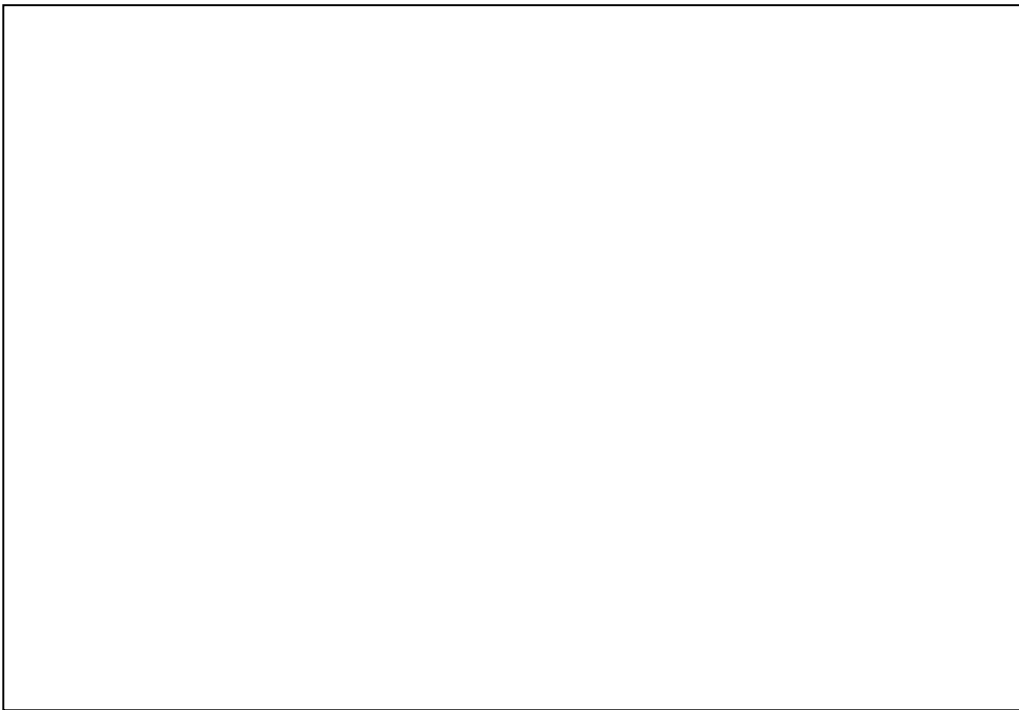


Many compilers do not (by default) allow you to compile a .h file.

However, compilers often have a switch which can be thrown to force a compilation. For example, on the Microsoft compiler uses the /TC command line switch for force a file to be compiled as C (regardless of its filename extension).

Another option is to copy a .h file to a file with a .c extension so it can be compiled.





The standard defines the `#pragma` directive in clause 6.10.6 and the `_Pragma` operator in clause 6.10.9

The standard reserves the following three specific `#pragmas`

`#pragma STDC FP_CONTRACT on-off`

`#pragma STDC FENV_ACCESS on-off`

`#pragma STDC_CX_LIMITED_RANGE on-off`

where *on-off* is either ON or OFF or DEFAULT

