

PreProcessing

```
#include <stdio.h>

#define PP_NARG(...) \
    PP_NARG_(__VA_ARGS__, PP_RSEQ_N())

#define PP_NARG_(...) \
    PP_ARG_N(__VA_ARGS__)

#define PP_ARG_N( \
    _1, _2, _3, _4, _5, _6, N, ...)    (N)

#define PP_RSEQ_N() \
    6,5,4,3,2,1,0

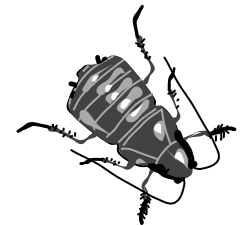
int main(void)
{
    printf("%d", PP_NARG(a,b,c,d));
    printf("%d", PP_NARG(a+b,c+d));
    return 0;
}
```

42

Deep C - a 3 day course
Jon Jagger & Olve Maudal

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



gotcha

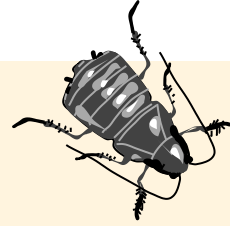
phase 6: adjacent string literals are concatenated

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

← 3 commas

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

← 2 commas



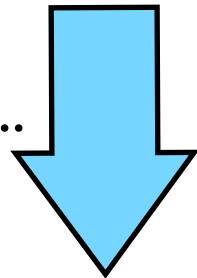
object macros

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

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

one or more spaces after
the identifier

preprocesses to...

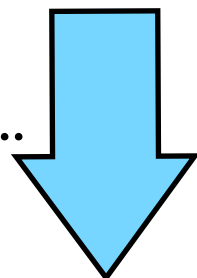


```
#define BUFFER_SIZE (100)
```

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

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

preprocesses to...



```
#define printf my_printf
```

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

?

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

predefined macros

`__func__`

- the name of the current function (a string literal) (c99)

`__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)

function macros

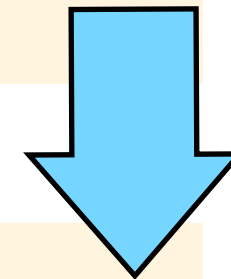
a function-like macro accepts arguments

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



no space between the identifier and
the left parentheses

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



preprocesses to...

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

macro guidelines

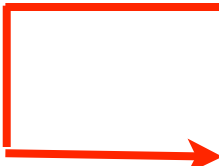
macro names should use UPPERCASE and _ underscore only

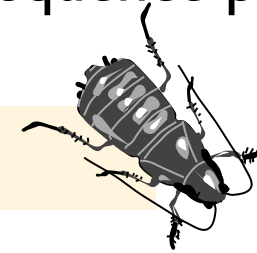
- never lowercase
- this is a very strong convention

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



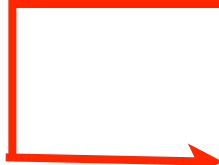
this looks like a function call (with a sequence point)
but it's not, it's a macro :-)

max(delta, precision);



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

this looks like a function MACRO (without a sequence point)
and it is indeed a MACRO

MAX(delta, precision);

macro guidelines

```
#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)
```

if the replacement tokens form an expression
put the whole replacement text in parentheses



better

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

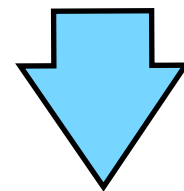

macro side effects

Macro arguments with side effect can happen multiple times

- Surprising enough even without considering sequence points!

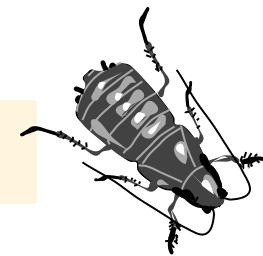
```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

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



preprocesses to...

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



macro guidelines

c99

Consider if you can replace the macro with an inline function

```
#define IS_ODD(n) ((n) % 2 == 1)
```



better

```
static inline bool is_odd(int n)
{
    return n % 2 == 1;
}
```

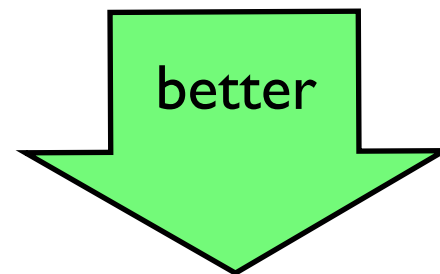


logical lines

A backslash followed immediately by a newline is deleted (phase 2)

- Allows multiple physical lines to form one logical line
- Used to increase readability of macro replacement

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



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

where `\n` is being used to
represent a newline character

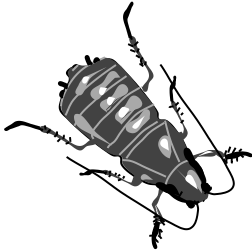
macro problem

macros bigger than a single expression...

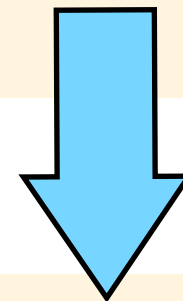
- can easily interfere with their surrounding context

```
#define LOG(msg)  if (in_log_mode) \n                  puts(msg)
```

??



```
if (toggled())  
    LOG("on");  
else  
    LOG("off");
```



preprocess to...

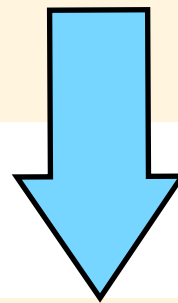
```
if (toggled())  
    if (in_log_mode)  
        puts("on");  
else if (in_log_mode)  
    puts("off");
```

macro solution 1

rephrase the logic in a single expression

```
#define LOG(msg) \
    ((void) (in_log_mode && puts(msg)))
```

```
if (toggled())
    LOG("on");
else
    LOG("off");
```



preprocess to...

```
if (toggled())
    ((void) (in_log_mode && puts("on")));
else
    ((void) (in_log_mode && puts("off")));
```

macro solution 2

embed the macro replacement inside a do-while(0) loop

```
#define LOG(msg)  do {                                \n                  if (in_log_mode)                  \n                      puts(msg);                    \n                  } while (0)
```

don't include a trailing semi-colon here 
instead put it here at each point of use

```
if (toggled())  
    LOG("on");  
else  
    LOG("off");
```

 preprocess to...

```
if (toggled())  
    do { if (in_log_mode) puts("on"); } while (0);  
else  
    do { if (in_log_mode) puts("off"); } while (0);
```

variadic function macros

c99

can accept a variable number of arguments

`__VA_ARGS__` expands to the elided arguments

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

no space between the identifier and the left parentheses

```
#define DEBUG(...) fprintf(stderr, __VA_ARGS__)
```

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

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

preprocesses to...

Here's an amazing c99 macro to count how many arguments a function macro is called with!

c99



```
#include <stdio.h>

#define PP_NARG(...) \
    PP_NARG_(__VA_ARGS__, PP_REV_SEQ_N())

#define PP_NARG_(...) \
    PP_ARG_N(__VA_ARGS__)

#define PP_ARG_N( \
    _1, _2, _3, _4, _5, _6, N, ...)    (N)

#define PP_REV_SEQ_N() \
    6,5,4,3,2,1,0

int main(void)
{
    printf("%d", PP_NARG(a,b,c,d));
    printf("%d", PP_NARG(a+b,c+d));
    return 0;
}
```

4

2

Don't use this.

#include

the most common directive

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

h-char == any character except > or newline

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

q-char == any character except " or newline

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

must expand to < > or " " form

```
# include pp-tokens
```

" " for local headers → #include "widget\table.h" ← is this a tab character?
< > for system headers → #include <stdio.h> ← this is not a string literal
rare → #include INCFILE

include order

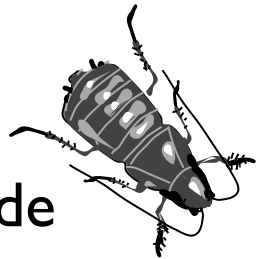
a source file should `#include` its own header before any other header

- this helps to ensure they don't accidentally compile because of a previous `#include`
- consider ensuring 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>  
...
```



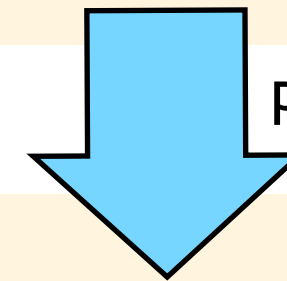
operator

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);
```



preprocesses to...

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



The point of the `#` operator is to create a string representation of its argument
as the developer sees it

```
#include <stdio.h>

#define STR(arg)  #arg

int main(void)
{
    puts("x\ty");
    puts(STR("x\ty"));
    return 0;
}
```

```
x      y
x\t y
```

For example

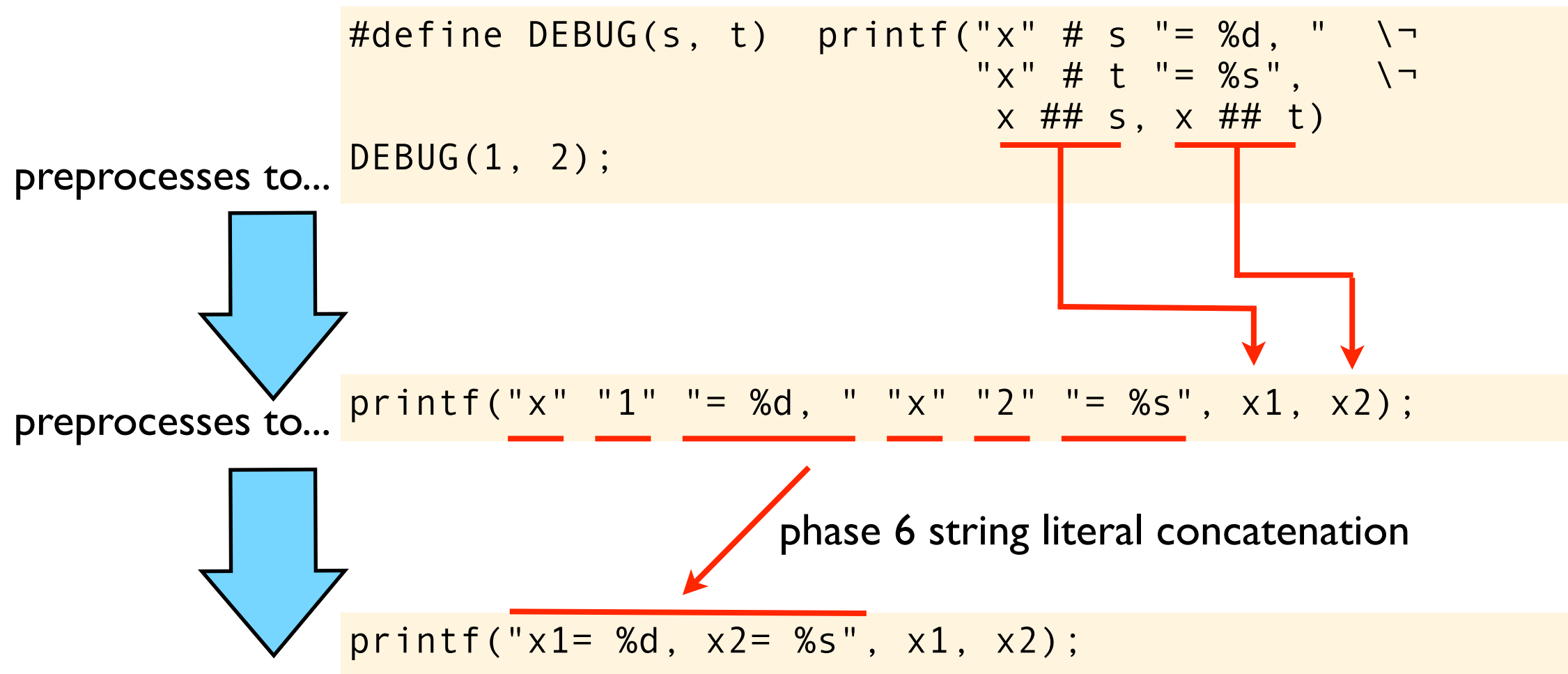
`"x\ty"` is { `'x'`, `'\t'`, `'y'`, `'\0'` }

but

`STR("x\ty")` is { `'x'`, `'\\'`, `'t'`, `'y'`, `'\0'` }

operator

operator concatenates two arguments

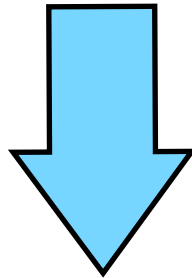


#define #undef

you can define and undefine an identifier as a macro name

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

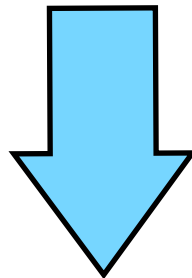
preprocesses to...



```
#define BUFFER_SIZE /*nothing*/  
char buffer[BUFFER_SIZE];
```

```
char buffer[];
```

preprocesses to...



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

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

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

anonymous enums

- consider anonymous enums as an alternative to #defines

```
#define MAX_SIZE (1024)  
char buffer[MAX_SIZE];
```



```
enum { MAX_SIZE = 1024 };  
char buffer[MAX_SIZE];
```



```
enum { max_size = 1024 };  
char buffer[max_size];
```



conditionals

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

`#elif == #else #if` 

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

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

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

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

conditionals

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 †

widget_table.h

```
#ifndef WIDGET_TABLE_INCLUDED  
#define WIDGET_TABLE_INCLUDED  
...  
...  
...  
#endif
```

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

† idempotent basically means "once-only"

#error

the #error directive

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

error *pp-tokens_{opt}*

diagnostic message

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

#pragma

c99

causes implementation-defined behaviour

```
# pragma pp-tokensopt
```

also available via the `_Pragma` operator

```
_Pragma pp-tokensopt
```

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

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

summary

- 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 int macro → enumerator

"I'd like to see Cpp [the C pre processor] abolished."
p426

"In retrospect, maybe the worst aspect of
Cpp is that it has stifled the development
of programming environments for C."
p424

