

Preprocessing

Translation Phases

2

- 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



- 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));
```

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

space here is not allowed

```
# define identifier( identifier-list, opt ... )  ↵  
    pp-tokensopt
```

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

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

- the # operator converts its argument to a string literal
 - ◆ if the argument is a string literal or character constant \ is inserted before " and \

\ phase 2 logical lines

```

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

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

```

```

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

```

operator

- **## operator concatenates two arguments**

operator

```
#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

- **##** operates on single preprocessor tokens
 - ◆ one to the left of **##** and one to the right of **##**

```
#define GLUE(a, b) a ## b
```

```
double d = GLUE(6.4e, -2); /* 6.4e-2 */
```

```
double d = 6.4e ## -2
```

```
double d = 6.4e ## - 2
```

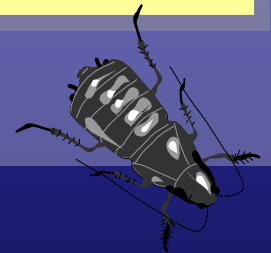
```
double d = (6.4e ## -) 2
```

```
double d = 6.4e- 2
```



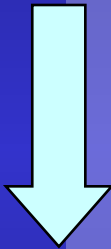
compile time error

Beware



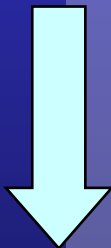
- you can define an identifier as a macro name with a replacement list

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



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

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



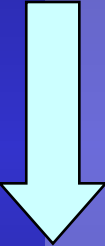
```
#define printf my_printf  
printf("error: %s", message);
```



```
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];
```

#define #undef

- **__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[†]

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

conditionals

[†]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 "
```

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

- Declares an identifier as the name of a type
 - ◆ And no more; doesn't reveal its representation

```
struct wibble;
```

- ◆ Advantages over full type definition via include
 - reduced physical dependencies
 - reduced build times
 - greater abstraction

```
#include "wibble.h" →
```

```
#include "..."  
#include <...>  
  
struct wibble  
{  
    ...  
};
```

- When is an include necessary?
- When is a forward declaration sufficient?

```
#include "wibble.h"
```

```
typedef struct wibble wibble;
```

```
struct s
```

```
{
```

```
    wibble value;
```

```
    wibble * ptr;
```

```
};
```

```
void arg_by_value(wibble);
```

```
void arg_by_ptr(wibble *);
```

```
wibble return_by_value(void);
```

```
wibble * return_by_ptr(void);
```

1

2

3

4

5

6



- Only **1** requires include
 - ◆ The function declarations are not definitions
 - ◆ inline definitions *might* need a #include

```
typedef struct wibble wibble;

struct s
{
    int value;
    wibble * ptr;
};

void arg_by_value(wibble);
void arg_by_ptr(wibble *);

wibble    return_by_value(void);
wibble *  return_by_ptr(void);
```

- **#include local header before system headers**
 - ◆ this helps to ensure 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
```

#error

- 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)  
    ...
```

#pragma

- macro names in UPPER_CASE only
 - ◆ never use lowercase

```
#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);
```

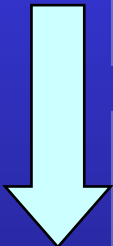
```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```



this looks like a function macro

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

guidelines



- 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 corrupt their surrounding context

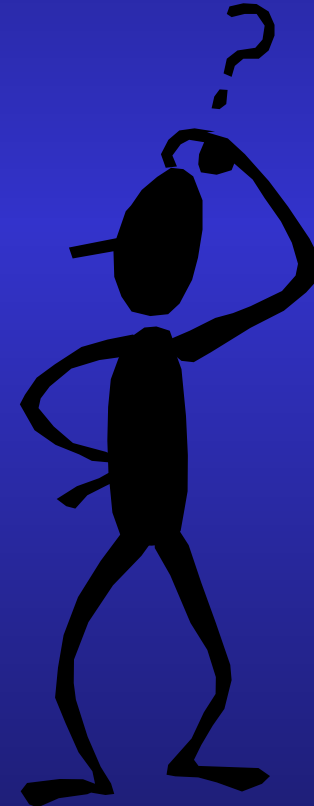
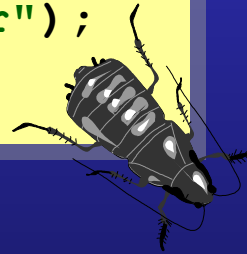
```
#define TRACE(msg)  if (dbg_mode) puts(msg)
```



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



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



problem

- see if it is possible to rephrase 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
    ...
```

solution

- alternatively, if there is no way to use a single expression, use the do-while(0) trick

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

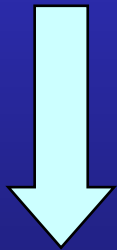
don't include a trailing semi-colon

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

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

- **in general – 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

summary



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



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



- This course was written by

Expertise: Agility, Process, OO, Patterns
Training+Designing+Consulting+Mentoring

{ JSL }

Jon Jagger

jon@jaggersoft.com

www.jaggersoft.com