The C Preprocessor

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Preprocessing

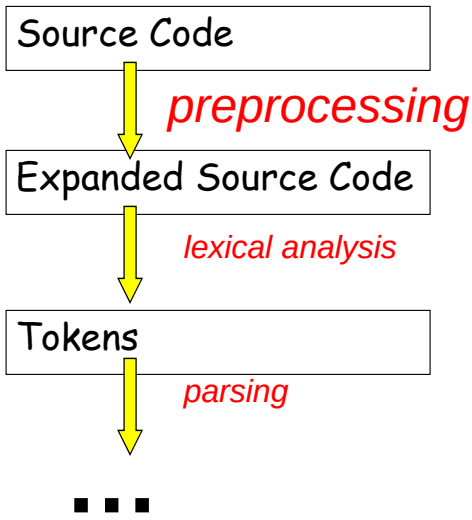
- Modifies the contents of the source code file before compiling begins
- The proprocessor is run automatically when you compile your program
  - use **`gcc –E`** option if you want to see just the results of the preprocessing step
- It is (mostly) simple string substitution

```
#define PI 3.1415926
double x = PI * d;
```

preprocess to get…

```
double x = 3.1415926 * d;
```

Steps in Compiling (Review)




```
#define N 3
a=c+b*N;
```

```
a=c+b*3;
```

```
a  =  c + b * 3  ;
```


Uses of Preprocessing

1. (header) file inclusion
   (e.g., `#include <stdio.h>`)
2. macro substitution for common (short) fragments of code
   (e.g., `#define PI 3.1415926`)
3. conditional compilation
   (e.g., `#ifdef DEBUG` … `#endif`)

- No preprocessing provided in Java

Preprocessor Commands

- Any line starting with the # character
- A preprocessing command is terminated by the end of the line, unless continued with a \
- Ex.:

```
#define PISHORT 3.1416

#define PILONG \
        3.14159265358979323846264
```


#define

- **#define identifier token-sequence**
- Proprocessor: anywhere it finds **identifier** in the program, it replaces it with **token-sequence**
- One use: giving names to "magic" constants, ex.:

```
#define E 2.718282
#define BIGRAISE 50000
#define FALSE 0
#define TRUE 1
#define ERROR -1
#define EQ ==
#define TABSIZE 100
```

#define (cont'd)

```
if (really_good_year EQ TRUE)
   salary += BIGRAISE;
```

preprocess to get…

```
if (really_good_year == 1)
   salary += 50000;
```

- This is not the same as declaring a variable; no storage is allocated
- You've already used such constants: **EOF**, **RAND_MAX**

#define (cont'd)

```
int table[TABSIZE];
…
for (i = 0; i < TABSIZE; i++)
   if (table[i] EQ 15)
   …
```

- is translated by the preprocessor (before compiling) into…

```
int table[100];
…
for (i = 0; i < 100; i++)
   if (table[i] == 15)
   …
```

More About #define

- **#defines** can also contain **#define**'d values

```
#define PI 3.1415926
#define TWOPI 2*PI
```

By convention, **#define** identifiers are written in ALL CAPS

Do not terminate **#define** by '**;**' or it becomes part of **token_sequence**!

common source of bugs
terminating macro definition with ';'

```
#define PI 3.1416 ;
…
area = PI * r * r;
```

→

```
area = 3.1416 ; * r * r;
```


#define … (cont'd)

If a **#define** contains multiple statements, put the whole thing in braces (i.e., use block structure)

```
#define MYMAC \
    i = j; \
    j = 16
…
if (m > 27)
   MYMAC ;
```

→

```
if (m > 27)
    i = j;
    j = 16;
```

Without using braces (wrong)

common source of bugs
failure to delimit a block macro with {}

#define … (cont'd)

```
#define MYMAC \
{ \
    i = j; \
    j = 16; \
}
…
if (m > 27)
   MYMAC;
```

```
if (m > 27)
{
    i = j;
    j = 16;
}
```

With braces (right)

More… (cont'd)

- The **token_sequence** does not need to be a valid expression or statement, e.g.,

```
#define TIMES2 * 2
…
a = b TIMES2 ;
```

```
a = b * 2 ;
```

Macro Expansion

- **#define** can take parameters or arguments (like functions), e.g.,

Note: no white space!

```
#define DIAM(radius) 2*PI*(radius)
…
diameter = DIAM(r);
```

```
diameter = 2*PI*(r);
```

looks like a function call, but it's not!

Parenthesize Macro Parameters

- Without parentheses:

common source of bugs
failure to parenthesize a macro parameter

```
#define SQUARE(x) x * x
…
y = SQUARE(z+1) ;
```

```
y = z+1 * z+1 ;
```

With parentheses:

```
#define SQUARE(x) (x) * (x)
…
y = SQUARE(z+1) ;
```

difference???

```
y = (z+1) * (z+1) ;
```

Macros vs. Functions

Which is better:

- a macro **F**?
- a function **f()**?

```
#define F(j,k) \
{ \
    int i; \
    i = j + k; \
    j = i * 2; \
}
…
if (m > 27)
   F(x,y);
```

```
int f(int j, int k)
{
    int i;
    i = j + k;
    return (i * 2);
}
…
if (m > 27)
   x = f(x,y);
```


Macros vs. Functions… (cont'd)

One difference: do not have to declare the type of the arguments of a macro – but it may still matter

```
#define DIAM(radius) 2*PI*(radius)
```

VS.

```
double diamf (float radius)
{…}
```

```
double diamd (double radius)
{…}
```

```
double diami (int radius)
{…}
```

Macro Expansion of Macro Expansion of …

- Ex:

```
#define ABSDIFF(a,b) \
            ((a)>(b) ? (a)-(b) : (b)-(a))
```

What is result of…?

```
x = ABSDIFF(5,35) ;
```

```
x = ((5)>(35) ? (5)-(35) : (35)-(5)) ;
```


Macro Expansion of… (cont'd)

What is result of…?

```
x = ABSDIFF(70, ABSDIFF(5,35)) ;
```

```
x = (((70) > ((5)>(35)?(5)-(35):(35)-(5)) ?
(70-((5)>(35)?(5)-(35):(35)-(5)) :
((5)>(35)?(5)-(35):(35)-(5)))-(5))-(70))) ;
```

Side Effects and Macro Arguments

- Watch out for input parameters to macros that have side effects (e.g., **x++**)

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

What happens with…

```
x = MAX(y++,z++);
```

```
x = ((y++)>(z++) ? (y++) : (z++));
```

common source of bugs

invoking macros with parameters that have side effects

Before execution

y	z
2	3

After execution

x	y	z
?	?	?

Advice: avoid side effects in macro arguments!

Two More Uses

Quoting with the # character

```
#define TMPFILE(dir,fname) #dir #fname
…
char *s = TMPFILE(/usr/tmp,test1) ;
```

```
char *s = "/usr/tmp" "test1" ;
```

- Concatenation with the ## characters

```
#define CAT(x,y) x ## y
…
a = CAT(b,123) ;
```

```
a = b123;
```

`#include`

- Inserts into the source code the contents of another file
  - often called a *header* file (filetype: **`.h`**)

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

standard library header file

user defined header file

Where does gcc look for these files?

- installation dependent (but often **`/usr/include`**)
- same directory as source code file
- other locations controlled by gcc **`-I`** option

`#include` (cont'd)

- Frequently part of header files:
  - constant definitions
  - function prototype declarations (for libraries)
  - **`extern`** declarations (we'll discuss later)
- When the header file changes, all source files that **`#include`** it have to be recompiled
  - i.e., there is a dependency of this source code on the contents of the header file

Some Useful (Standard) Header Files

- **`stdio.h`**
- **`stddef.h`**
- **`math.h`**
- **`string.h`**
- **`float.h`** and **`limits.h`**
- Take a look in **`/usr/include`** on your system

Conditional Compilation

- To control what source code gets compiled
- Common uses
  - to resolve, at compile time, platform (machine- or OS-) dependencies
  - to compile (or not) debugging code
- Requires the following preprocessor directives
  - **`#if / #ifdef / #ifndef`**
  - **`#elif / #else`**
  - **`#endif`**

Conditional Compilation: Example

```
#if defined(LINUX)
  #define HDR "linux.h"
#elif defined(WIN32)
  #define HDR "windows.h"
#else
  #define HDR "default.h"
#endif

#include HDR
```

- And when compiling this program, can define what **SYSTEM** is by using the **-D** option to **gcc**

```
gcc –DWIN32 myprog.c …
```

```
#include "windows.h"
```


Another Use

- To avoid repeatedly including the same header file
  - what problem would this cause if not prevented?

```
#ifndef MYHDRFILE   /* MYHDRFILE */
#define MYHDRFILE 1
…header file contents…
#endif              /* MYHDRFILE */
```

```
#include "myheader.h"
…
#include "myheader.h"
```

Included only once

```
…header file contents…
```

Recommendations

- GNOME: *"If you have random 'magic values' in your program or library, use macros to define them instead of hardcoding them where they are used"*

Debugging…

- Use macros to execute your program in debug mode
  - Assume program compiled with the following command

```
gcc –DDEBUG=1 –Wall –std=c99 myprog.c
```

  - How would your print a debug message in your code?