

# Introduction to C, C++, and Unix/ Linux

CS 60

Today

- More preprocessor/macros
- Header files, printf/scanf, ...
  - Reading for today: K&R ch. 4, 7
  - Reading for Wednesday: K&R ch. 5

# C preprocessing

Lines starting with # are Pre-Processor directives. They are not statements, hence do not end in ‘;’

```
#include <stdio.h>
main()
{
    if (1)
        printf("Its summer!\n");
}
```

The C preprocessor has many features – and pitfalls

## C preprocessing (cont.)

- Compilation begins by moving the C source files through a preprocessor – a simple text substitution device
- One can designate strings for substitution with the expression:

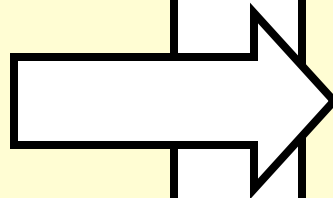
```
#define <expression> <substitute>
```

- This can be used to define constants and to define macros

## Example – constant

```
#define GREETING "Hello World"
```

```
int main ( void )  
{  
    printf (GREETING);  
}
```



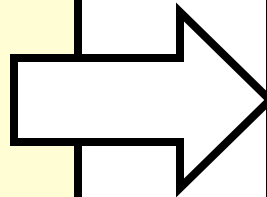
```
int main ( void )  
{  
    printf ("Hello World");  
}
```

Good – we can put all the text strings used by the computer at the top of the file, for easy future editing

## Example – macro

```
#define SayHello(str) \  
printf("Hi there, %s\n", str)
```

```
int main ( void )  
{  
    SayHello("Jack");  
    SayHello("Jill");  
}
```



```
int main ( void )  
{  
    printf("Hi there, %s\n", "Jack");  
    printf("Hi there, %s\n", "Jill");  
}
```

Good – we can make our code more readable and easier to modify

## C preprocessor – constants and macros

```
#define GREETING "Hello World"
#define VERSION 1.2
#define HOMEDIR "/cs/faculty/mturk"

#define SayHello(str) printf("Hi \
    there, %s\n", str)
#define DOIT(expr, num) \
    for (i=0; i<num; i++) expr;
```

## Aside – “white space”

- In a source file, blanks, tabs, newlines, etc. are collectively called “white space”
  - You can’t see ‘em
- In C, white space serves to separate tokens
  - Identifiers, keywords, constants, strings, operators...
- The C compiler otherwise ignores white space

So these are the same:

```
int x,y=100;
```

```
int x, y = 100;
```

```
printf("YES");
```

```
printf ( "YES" ) ;
```

```
main(void)
```

```
main ( void )
```

But not:

```
#define OK(x) -(x)
```

```
#define OK (x) \  
- (x)
```

# C preprocessing – macros

- So...

```
#define g(y) get_index(y)
```

```
g(fp) → get_index(fp)
```

```
#define g (y) get_index(y)
```

```
g(fp) → (y) get_index(y) (fp)
```

Oops, extra white space

(which is generally not a problem in C)

**Yipes!**

## C preprocessing (cont.)

What's wrong with this?

```
#define square(x) x * x
```

It could be called thus:

```
square (z + 4)
```

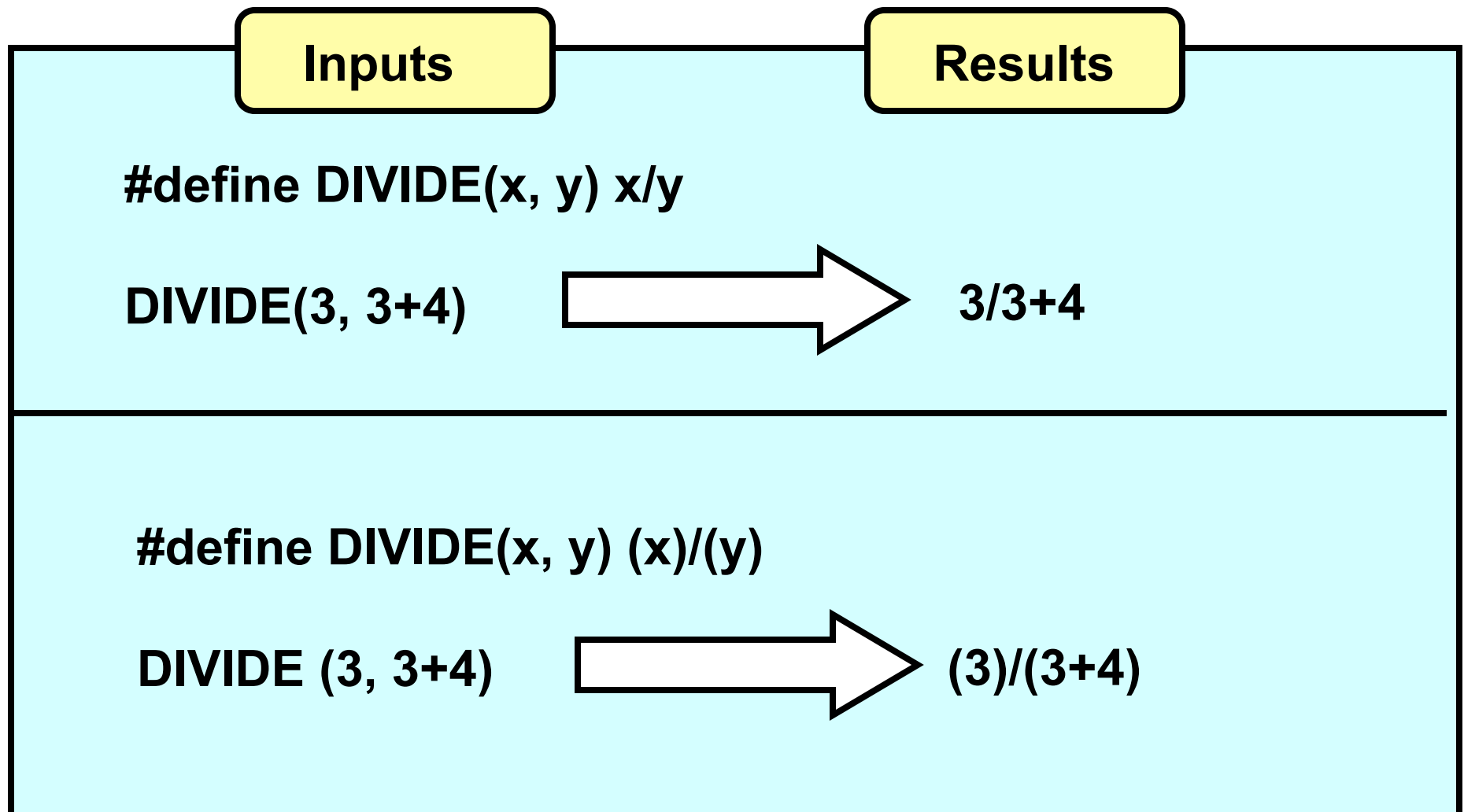
Resulting in:

```
z + 4 * z + 4
```

And not the intended:

```
(z+4) * (z+4)
```

3/7 = 5? Liberally use parentheses!



```
#define CM_TO_IN(x) ((x)/2.54)
```

← Macro

```
float cm_to_in(long val)
```

← Function

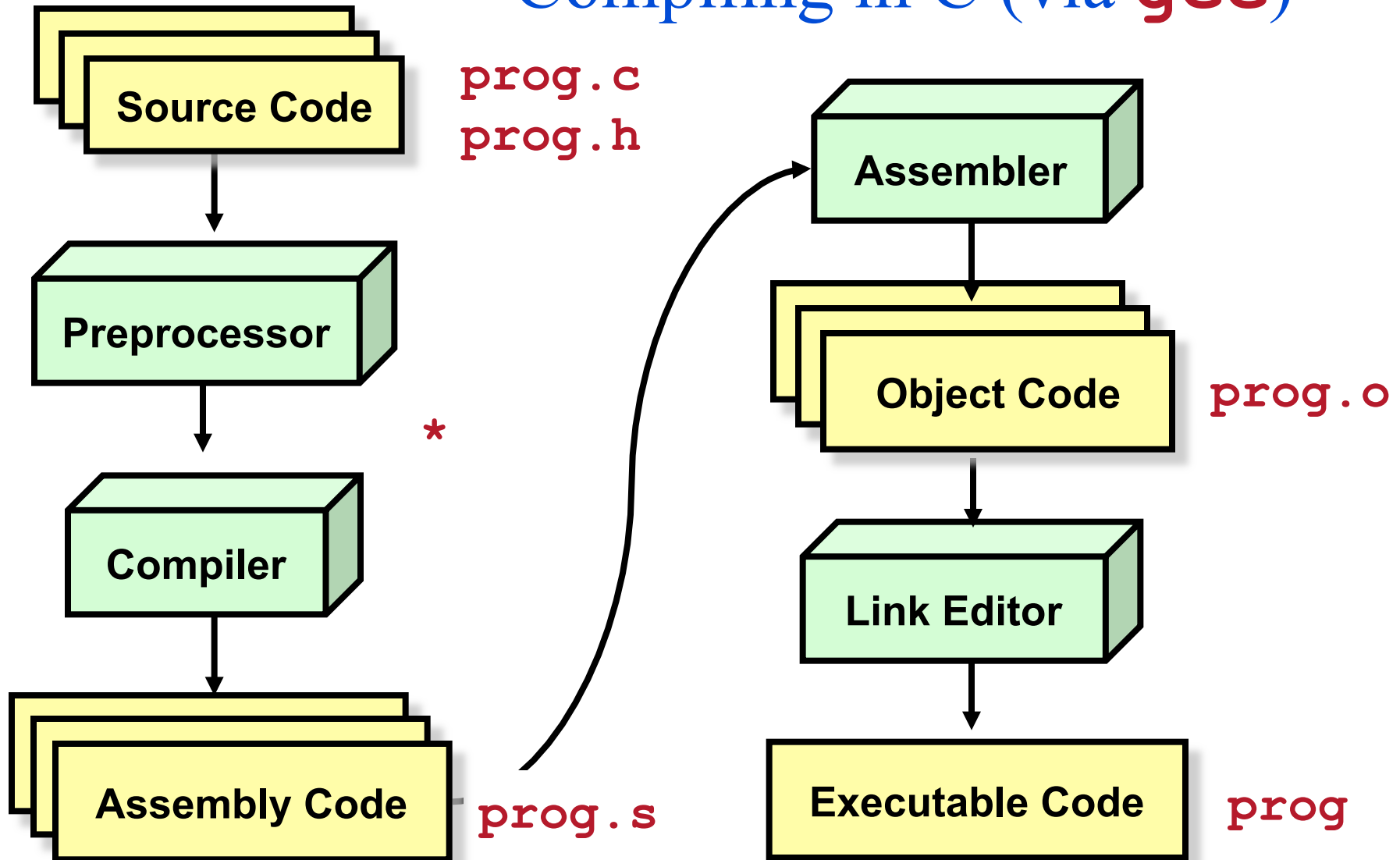
```
{  
    return val/2.54;  
}
```

```
int main(void)
```

```
{  
    long i;  
    for (i=0; i<1000000000; i++)  
        CM_TO_IN(i);  
}
```

Which to use: `cm_to_in()` or `CM_TO_IN()` ?

# Compiling in C (via gcc)



# Intermediary files

- **gcc -E** produces source code after preprocessing
  - Prints to standard output
- **gcc -S** produces assembly code (.s)
- **gcc -c** produces object (machine) code (.o)
- **gcc** produces an executable file
  - **-o prog** produces a program called “prog”
  - **./prog** runs the program

## Source file – main.c

```
int main(void)
{
}
```

Now run `gcc -E main.c`

gcc -E main.c (after preprocessor)

```
# 1 "main.c"  
# 1 "<built-in>"  
# 1 "<command line>"  
# 1 "main.c"  
int main(void)  
{  
}
```

gcc -S main.c

```
.file "main.c"
      .text
.globl main
      .type main, @function
main:
      pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      andl $-16, %esp
      movl $0, %eax
      subl %eax, %esp
      leave
      ret
      .size main, .-main
      .section .note.GNU-stack,"",@progbits
      .ident "GCC: (GNU) (Red Hat Linux)"
```

main2.c (with **stdio.h** header file)

```
#include <stdio.h>
int main(void)
{
}
```

gcc -E main2.c (after preprocessor)

```
# 1 "main2.c"  
# 1 "<built-in>"  
# 1 "<command line>"  
# 1 "main2.c"
```

... then 908 more lines of code, then...

```
int main(void)  
{  
}
```

```
.file "main2.c"
```

gcc -S main2.c

```
.text
```

```
.globl main
```

```
.type main, @function
```

```
main:
```

```
    pushl %ebp
```

```
    movl %esp, %ebp
```

```
    subl $8, %esp
```

```
    andl $-16, %esp
```

```
    movl $0, %eax
```

```
    subl %eax, %esp
```

```
    leave
```

```
    ret
```

```
.size main, .-main
```

```
.section .note.GNU-stack,"",@progbits
```

```
.ident "GCC: (GNU) (Red Hat Linux)"
```

# Beyond “Hello, world”

- Let’s look at a program that does more than “Hello, world”

```
#include <stdio.h>
#define STEP 5
```

C Comments are (for now) only /\* ... \*/, and not //. More recent versions add // comments.

```
/* Calculate temp conversion table */
```

```
int main () {
    int fahr, celc;
    int min, max, step=STEP;
}
```

Declare all local variables at the top of the procedure.

```
printf("Enter lower limit (Fahrenheit): ");
scanf("%d", &min);
printf("Enter upper limit (Fahrenheit): ");
scanf("%d", &max);
```

```
fahr = min;
while (fahr <= max) {
    celc = 5 * (fahr-32)/9;
    printf ("%d\t%d\n", fahr, celc);
    fahr = fahr + step;
}
```

```
return 0;
```

Non-zero means error or unusual termination

```
}
```

```
#include <stdio.h>
#define STEP 5
```

The #include statement allows us access to existing libraries.

```
/* Calculate temp conversion table */
int main () {
    int fahr, celc;
    int min, max, step=STEP;

    printf("Enter lower limit (Fahrenheit): ");
    scanf("%d", &min);
    printf("Enter upper limit (Fahrenheit): ");
    scanf("%d", &max);

    fahr = min;
    while (fahr <= max) {
        celc = 5 * (fahr-32) / 9;
        printf ("%d\t%d\n", fahr, celc);
        fahr = fahr + step;
    }
    return 0;
}
```

C's looping constructs are just like Java's. The "boolean" termination is different, since C has no boolean type.

```
#include <stdio.h>
#define STEP 5

/* Calculate temp conversion
int main () {
    int fahr, celc;
    int min, max, step=STEP;

    printf("Enter lower limit (Fahrenheit): ");
    scanf("%d", &min);
    printf("Enter upper limit (Fahrenheit): ");
    scanf("%d", &max);

    fahr = min;
    while (fahr <= max) {
        celc = 5 * (fahr-32) / 9;
        printf ("%d\t%d\n", fahr, celc);
        fahr = fahr + step;
    }
    return 0;
}
```

By default, a function without a declared return value returns an int. Here, main doesn't have parameters. There's a way to get command line arguments, just like "String[ ] args" in Java, but it uses *pointers to pointers*. (We cover this later.)

Integer truncation occurs here, just like in Java.

This is the rough equivalent of a System.out.println() method in Java. The formatting for this "printf" call is a little complicated at first, but is very powerful.

# Header files

- Header files are specified two ways:
  - `#include <header.h>`
  - `#include "header.h"`
- System headers (`<header.h>`)
  - System-supplied definitions, macros, library functions
  - E.g., `stdio.h`, `math.h`, `malloc.h`
- Local headers (`"header.h"`)
  - Serves as the “public section” of your code, for use by others – specifies the API – and by related files
  - Constant definitions, macros, function declarations, ...

# Typical header file

```
#ifndef HEADER_NAME_H  
#define HEADER_NAME_H
```

```
//=====
```

```
// File, author, modification documentation  
// Module description  
// Copyright, etc.
```

```
//=====
```

```
// Macro Definitions
```

```
//=====
```

```
// Variable Declarations
```

```
//=====
```

```
// Function Declarations
```

```
#endif /* HEADER_NAME_H */
```

Why do this?

```
#include <stdio.h>
#include "prog1.h"
int main(void)
{
    ...
}
```

prog1.c



prog1.h

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

#define MAXNUM 100
float comb(int n);
...
```

# Header file etiquette

- Use `#ifndef`, `#define`, `#endif` to make sure the header file is only loaded once per file compilation
- For local header files, document well
  - Adequate documentation of author, purpose, modification history, etc.
  - Clearly label areas for constants, macros, variable and function declarations, etc.
- Header file should have no variable or function *definitions* – only *declarations*

# Basic console I/O – printf( ) and scanf( )

- Include the standard I/O header
  - `#include <stdio.h>`
- Example:

```
int num = 5;  
int nsq = num*num;  
printf ("%d squared is %d\n", num, nsq);
```

Substitute integers here

Newline character

From here

## Typical printf error

```
int num = 5;  
int nsq = num*num;  
printf ("%d squared is %d\n", nsq);
```

- C will “make up something” for the second substitution, using random garbage off the stack
  - You probably won’t get an error message