ECS 36A, April 23, 2024

Announcements

- Midterm has been moved to Tuesday, May 7
    - It was scheduled for Thursday, May 2
- Midterm study guide, sample midterm are on Canvas
    - Sample midterm is shorter than the real one will be
    - I will post answers to it on Monday, April 29
- Tutoring is available from the CS Tutoring Club

++, --

- How they work depends on if they come before or. after the value
- If before, use the value of the variable and *then* increment or decrement the variable
- If after, increment or decrement the value of the variable and *then* use the value
- Examples (assume a =7 initially)
  - x = ++a; x is 8, a is 8
  - x = a++; x is 7, a is 8
  - x = --a; x is 6, a is 6
  - x = a--; x is 7, a is 6

Evaluation of Arguments to Functions

- The order is not defined; it is usually left to right or right to left
  - But it could be much weirder . . .
- Example: suppose x is 5

```
printf("%d %d\n", ++x, ++x)
```

could print 6 7 or 7 6

Formatted Read

These read input or a string, and attempt to match the input with the format. If any match the desired input. the appropriate variable is set. It stops when it reads the first character that does not match the format.

- scanf (*format*, *variables*)
    - Same as fscanf(stdin, *format*, *variables*)
- sscanf(*string*, *format*, *variables*)
    - Reads from the given *string*

The Formats

A string with the following format indicators:

- One or more whitespace characters: matches 0 or more whitespaces in the input
- Ordinary character (except whitespace or %): matches the next character
- Conversion specification: begins with % (to match a % in the input, say %%)

Conversion Specifiers

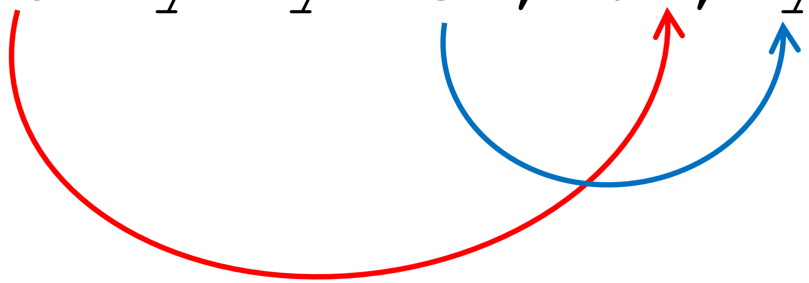
- Integer specification
  - ”%d”: next comes a decimal integer
  - “%o”: next comes an octal integer
  - “%x”: next comes a hexadecimal integer
  - ”%f”, “%e”, “%g”: next comes a floating point number
  - “%c”: next comes a single character
  - “%s”: next comes a string (a sequence of characters without whitespace)
- Examples:
  - “(%d,%d)” matches “(5,6)” but not “(5 6)”
  - “%s %s” matches “hello<tab>there” but not “hellothere”

Variables

- These are listed in the order of the conversion specifiers
- All must be pointers (addresses)

Examples:

```
int x; char y[1000];
scanf(“%d xyzzy %s”, &x, y)
```



In Program, Reading Integer

```
/* read an integer, looping until you do */
while(scanf("%d", &n) != 1){
      /* you didn't; report an error */
      fprintf(stderr, "Enter a positive integer\n");
}
printf("I read %d\n", n);
```

- Here's what happens when I type an integer:

```
% a.out
8
I read 8
```

Problem With This

- Works well if you type an integer
- But what happens if you don't?
- Here's the output when I typed "hello":

```
% a.out
hello
Enter a positive integer
Enter a positive integer
Enter a positive integer
. . .
```

What Happened?

- *scanf* tries to read an integer
- It encounters a non-integer and so does not read it
    - The non-integer ('h' in this case) is pushed back into the input stream
    - So the next time something is read, it will be the 'h'
- *scanf* returns 0
- **while** condition true, so it enters the loop and prints the error message
- It goes back to the **while** statement
- Repeat
    - And re-read the 'h' …

How to Fix It . . . Almost

- Problem: you need to throw away the rest of the line
- Fix: replace while loop:

```
/* read an integer, looping until you do */
while(scanf("%d", &n) != 1){
      /* you didn't; report an error */
      fprintf(stderr, "Enter a positive integer\n");
      while((n = getchar()) != EOF && n != '\n')
            ;
}
printf("I read %d\n", n);
```

note addition

Let's Test It

- 3 conditions: there is integer input, there is non-integer input, and there isn't any input
  - We know integer input works
- Non-integer input:

```
% a.out
hello
Enter a positive integer
8
I read 8
```

- It worked!

Let's Test It

- No input; type EOF to indicate the immediate end of input
  - On Linux and Macs, it's control-D; on Windows, it is usually one of control-D, control-C, or control-Z (consult your manual)

```
% a.out
^D
Enter a positive integer
Enter a positive integer
Enter a positive integer
```

- It didn't work ☹

What Happened

- On EOF, *scanf* returns **EOF** (–1)
- It enters the while loop and prints the error message
- It tries to read the rest of the line but gets **EOF**, so it falls out of the inner loop and goes back to the outer **while** statement
- Repeat

```
/* read an integer, looping until you do */
while(scanf("%d", &n) != 1){
        /* you didn't; report an error */
        fprintf(stderr, "Enter a positive integer\n");
        while((n = getchar()) != EOF && n != '\n')
                ;
}
```

The Real Fix

```
/* read an integer, looping until you do */
while((r = scanf("%d", &n)) != 1 && r != EOF){
        /* you didn't; report an error */
        fprintf(stderr, "Enter a positive integer\n");
        while((n = getchar()) != EOF && n != '\n')
               ;
}
if (r != EOF)
        printf("I read %d\n", n);
```

note change 2

note addition 1

Another Approach

- Read the line into a buffer, then apply *scanf* to the contents of the buffer
- New functions:
- `sscanf(char *buf, char *format, pointers)`
  - Just like *scanf*, but reads from buf and not the input
- `fgets(char *buf, int maxch, stdin)`
  - Read up to maxch–1 characters, or up to the next newline, and store them (*including* the newline) in buf; then return buf
  - On EOF, return NULL

In Program, Reading Integer

```
/* read a line, looping until you get an integer */
while (fgets(buf, 10, stdin) != NULL){
	/* see if there is an integer there */
	if (sscanf(buf, "%d", &n) != 1){
		/* nope; report an error */
		fprintf(stderr, "Enter a positive integer\n");
	}
	else{
		/* yep; print it and leave */
		printf("I read %d\n", n);
		break;
	}
}
```

Formatted Write

These write to a file or the standard output as indicated by the format string.

- *printf* (*format*, *variables*)
  - Same as *fprintf*(**stdout**, *format*, *variables*)
- *sprintf*(*string*, *format*, *variables*)
  - Writes its output into *string*
- *fprintf*(*fp*, *format*, *variables*)
  - Writes to the file with file pointer *fp*
  - You saw this with the file pointer **stderr** for the standard error output

The Formats

A string with the following format indicators:

- Anything except the conversion specifiers is copied
- Conversion specification: begins with %; “%%” outputs a single “%”
  - ”%d”: print next argument as a decimal integer
  - “%o”: print next argument as an octal integer
  - “%x”: print next argument as a hexadecimal integer
  - “%c”: print next argument as a single character
  - “%s”: print next argument as a string (sequence of characters)
  - “%f”: print next argument as a floating point number, like 12.345678
  - “%e”: print next argument as a floating point number, line 1.2345678e1
  - "%p": print next argument as a pointer value, like 0x7fff2435

Format Modifiers

- %#x, %#o: put “0x” in front of hexadecimal output, 0 in front of octal output
- %+d: always print a sign, either “+” or “-”
- %*n*d: print integer in a field of *n* characters
- %0*n*d: as above, but pad with leading 0s
- %-*n*d: left align the number in the field

If the number is bigger than the field, the field size is ignored

More Format Modifiers

For floating-point numbers:

- %*n*.*m*f: print float in a field of size *n*, with *m* decimal digits
  - If *m* is 0, then no decimal point or decimal fraction is printed
  - Printed value is in normal floating point form, like 123.4567
- %*n*.*m*e: print float in a field of size *n*, with *m* decimal digits
  - If *m* is 0, then no decimal point or decimal fraction is printed
  - Printed value is in scientific notation, like 1.234567e2

Still More Format Modifiers

For characters:

- %12.4c: this ignores the .4 and prints the character in a field of 12 wide
- %012c: as above, but ignores the 0

For strings:

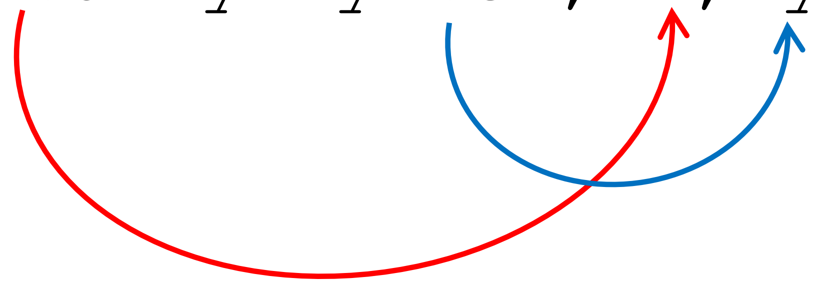
- %12.4s: prints first 4 characters of string in a field 12 wide
- %-12.4s: left justifies the above
- %012s: this ignores the 0

Variables

- These are listed in the order of the conversion specifiers
- They need not be pointers

Examples:

```
int x; char y[1000];
printf("%d xyzzy %s", x, y)
```



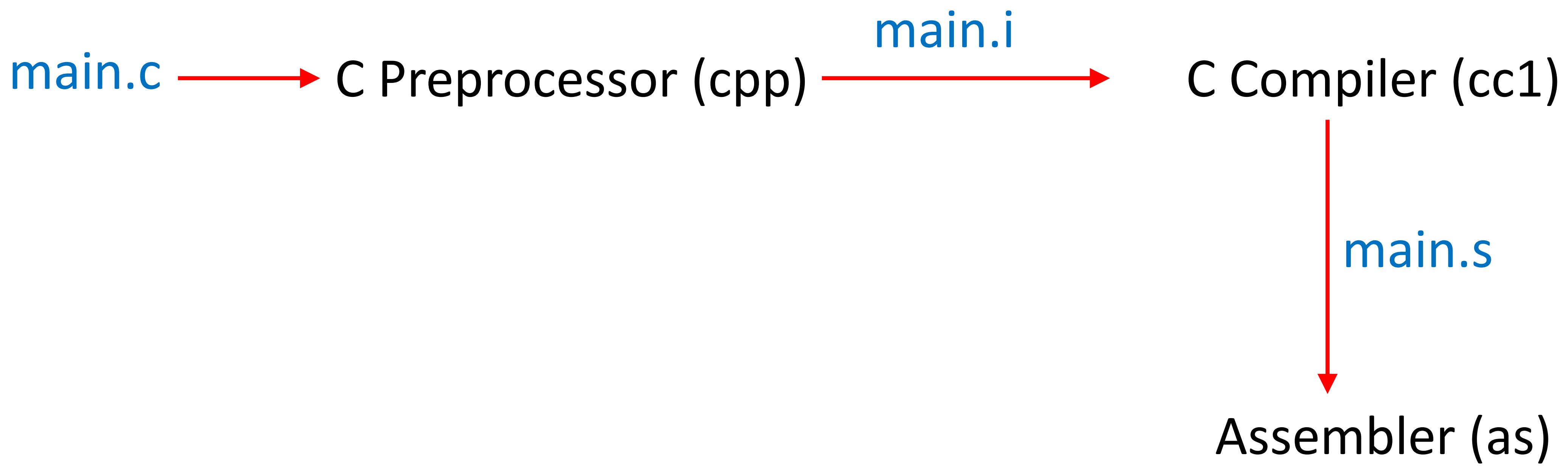
C Compiler Structure

$$\begin{tikzcd}\text{main.c} \arrow[r] & \text{C Preprocessor (cpp)}\end{tikzcd}$$

C Compiler Structure

$$\begin{tikzcd}
\text{main.c} \arrow[r] & \text{C Preprocessor (cpp)} \arrow[r, "\text{main.i}"] & \text{C Compiler (cc1)}
\end{tikzcd}$$

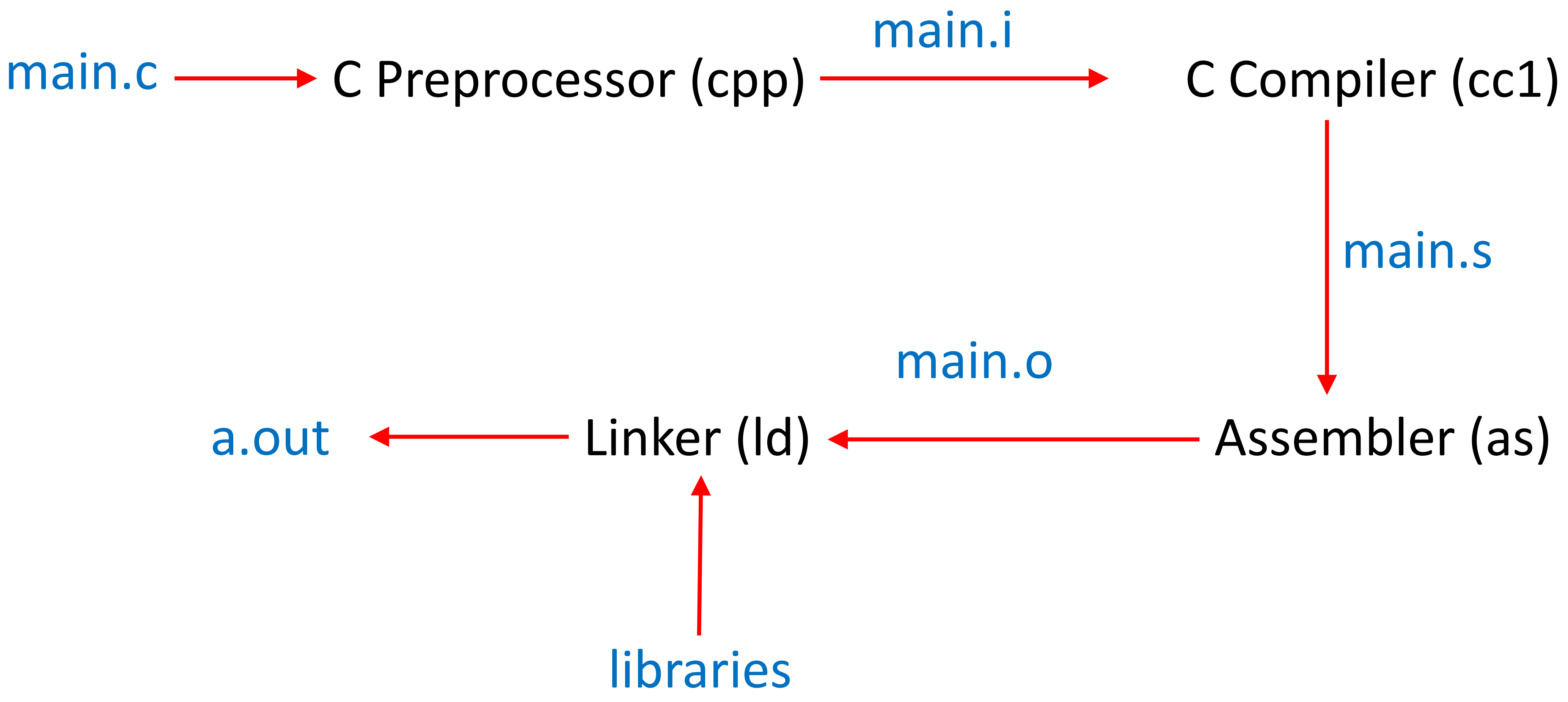
C Compiler Structure



C Compiler Structure

$$\begin{tikzcd}
\text{main.c} \arrow[r] & \text{C Preprocessor (cpp)} \arrow[r, "\text{main.i}"] & \text{C Compiler (cc1)} \arrow[d, "\text{main.s}"] \\
 & \text{Linker (ld)} & \text{Assembler (as)} \arrow[l, "\text{main.o}"'] \\
 & \text{libraries} \arrow[u] &
\end{tikzcd}$$

C Compiler Structure



C Preprocessor

All lines begin with #

- `#define`
- `#undef`
- `#include`

Defines built-in constants too; these normally begin with 2 "_"s or one "_" and a capital letter

- `__LINE__`
- `__FILE__`

`#define`

```
#define BOARD  8*8
```

- Replace every occurrence of the word “BOARD” with “8*8”

- Usually used to parameterize something; examples from stdio.h:
  - NULL is a macro (0)
  - EOF is a macro (–1)
- *Warning: this is textual substitution, so do not treat them as variables!*

Watch Out For This Sort of Macro

- Goal: create a chessboard, each side being 8 squares, and 2 extra squares for computation, for a total of 100 squares

```
#define SIDE 8+2
```

- Now every occurrence of SIDE is replaced by 8+2

```
char chess[SIDE*SIDE];
```

becomes

```
char chess[8+2*8+2];
```

So the board has 26 squares

Do This

- Goal: create a chessboard, each side being 8 squares, and 2 extra squares for computation, for a total of 100 squares

```
#define SIDE (8+2)
```

- Now every occurrence of SIDE is replaced by (8+2)

```
char chess[SIDE*SIDE];
```

becomes

```
char chess[(8+2)*(8+2)];
```

So the board has 100 squares

Parameterized Macro

```
#define isbetween0and9(x)    ((0<=(x))&&((x)<=9))
```

- **isbetween0and9**(4) returns 1 and **isbetween0and9**(–100) returns 0
- Beware — whatever is put for *x* is evaluated every time x occurs in the macro definition

```
x = 9; . . . isbetween0and9(x++)
```

becomes

```
x = 9; . . . ((0<=(x++))&&((x++)<=9))
```

or

```
x = 9; . . . ((0<=(9))&&((10)<=9))
```

which returns false (as 10 > 9)

#undef

- Delete a macro definition

```
#define XYZZY    "dizzy"
. . .
#undef XYZZY
int XYZZY = -20;
```

- Without the #undef, the declaration becomes:

```
int "dizzy" = -20;
```

which gives an error

#include

- Interpolate file into current source code
- When it does this, it preserves the line numbers of the original files by using these:

```
    # 9 "macros.c"
```

  Next line is treated as line 9 by the compiler and debuggers

#include

`#include <`*`file`*`>`

- Look for *file* in predetermined, system locations
  - Usually /usr/include, /usr/lib/include, and others
  - The "<" ">" are what tells the C preprocessor to do this

`#include "`*`file`*`"`

- Look for *file* in the current working directory first
  - The quotation marks are what tells the C preprocessor to do this

`–I `*`dir`*

- Add *dir* to the list of directories to be searched
  - Look in system directories *first*, then named directories

Predefined Constants

- `__LINE__`: current line number of the source file being preprocessed
  - Does *not* take into account any interpolations from included files
- `__FILE__`: file name as given to the preprocessor
  - Usually the file name you gave to cc, gcc, or lcc
- `__STDC__`: set if you are using standard C
  - Usually set to the latest version of C, so if you use an earlier version it will still be set
  - GNU C, and other versions of C, have non-standard extensions

System-Specific Predefined Constants

- Constants identifying the operating system; examples
  - `__linux__`, `__linux`, `linux`: Linux operating system
  - `__gnu_linux__`: GNU Linux system,
  - `__unix__`, `__unix`, or `unix`: Unix operating system
  - `__MACH__`, `__APPLE__`: MacOS operating system
- Constants identifying the architecture; examples
  - `__arm64__`, `__arm64`: 64-bit ARM chips
  - `__amd64__`, `__amd64`: 64-bit AMD chips
  - `__x86_64__`, `__x86_64`: 64-bit Intel chips