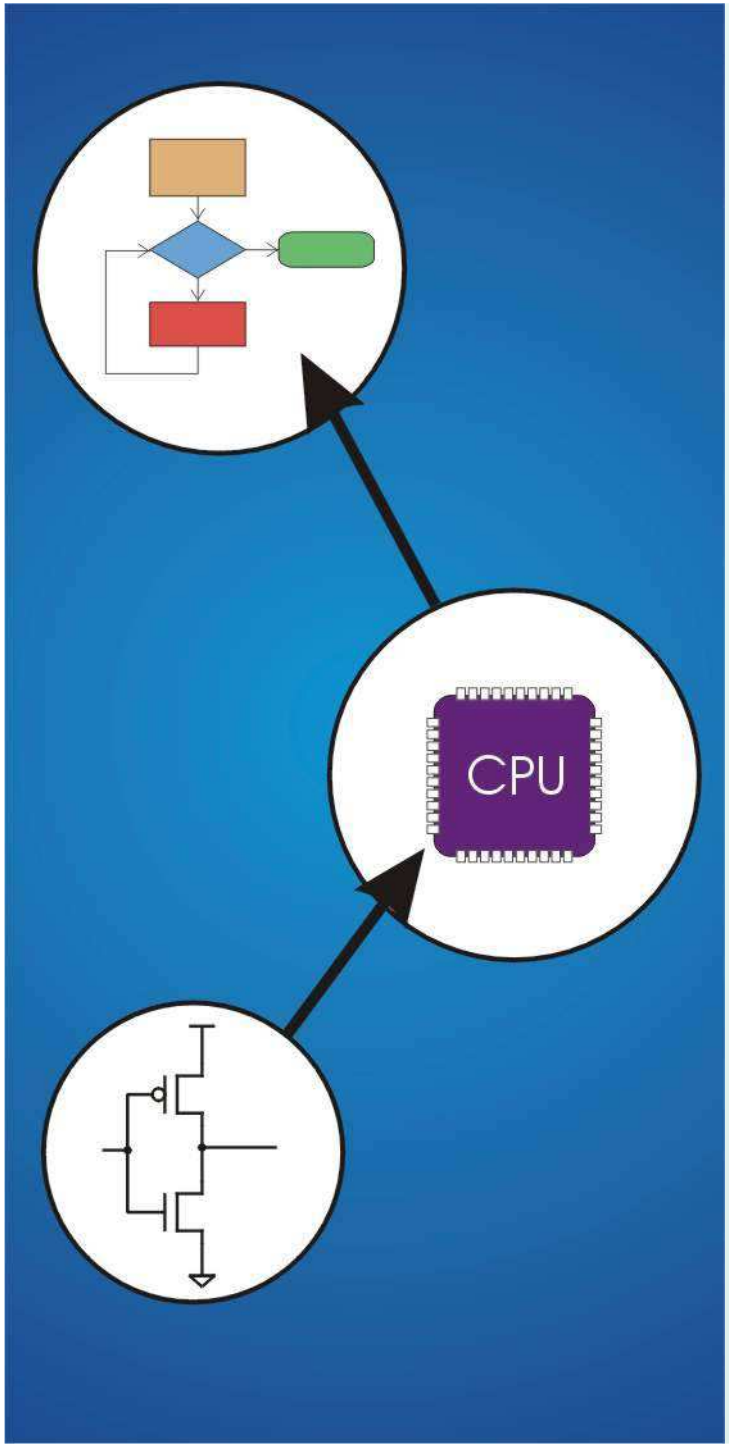

Chapter 18
I/O in C

Original slides from Gregory Byrd, North Carolina State University

Modified slides by C. Wilcox, Y. Malaiya
Colorado State University

Standard C Library

- I/O commands are not included as part of the C language.
- Instead, they are part of the Standard C Library.
  - A collection of functions and macros that must be implemented by any ANSI standard implementation.
  - Automatically linked with every executable.
  - Implementation depends on processor, operating system, etc., but interface is standard.
- Since they are not part of the language, compiler must be told about function interfaces.
- Standard header files are provided, which contain declarations of functions, variables, etc.

Basic I/O Functions

- The standard I/O functions are declared in the <stdio.h> header file.

Function	*Description*
`putchar`	Displays an ASCII character to the screen.
`getchar`	Reads an ASCII character from the keyboard.
`printf`	Displays a formatted string,
`scanf`	Reads a formatted string.
`fopen`	Open/create a file for I/O.
`fprintf`	Writes a formatted string to a file.
`fscanf`	Reads a formatted string from a file.

Text Streams

- All character-based I/O in C is performed on text streams.
- A stream is a sequence of ASCII characters, such as:
  - the sequence of ASCII characters printed to the monitor by a single program
  - the sequence of ASCII characters entered by the user during a single program
  - the sequence of ASCII characters in a single file
- Characters are processed in the order in which they were added to the stream.
  - e.g., a program sees input characters in the same order as the user typed them.
  - Standard input stream (keyboard) is called stdin.
  - Standard output stream (monitor) is called stdout.

Character I/O

`putchar(c)` Adds one ASCII character (c) to stdout.

`getchar()` Reads one ASCII character from stdin.

- These functions deal with "raw" ASCII characters; no type conversion is performed.

```
char c = 'h';
...
putchar(c);
putchar('h');
putchar(104);
```

Each of these calls prints 'h' to the screen.

Buffered I/O

- In many systems, characters are buffered in memory during an I/O operation.
  - Conceptually, each I/O stream has its own buffer.
- Keyboard input stream
  - Characters are added to the buffer only when the newline character (i.e., the "Enter" key) is pressed.
  - This allows user to correct input before confirming with Enter.
- Output stream
  - Characters are not flushed to the output device until the newline character is added.

Input Buffering

```
printf("Input character 1:\n");
inChar1 = getchar();

printf("Input character 2:\n");
inChar2 = getchar();
```

- After seeing the first prompt and typing a single character, nothing happens.
- Expect to see the second prompt, but character not added to stdin until Enter is pressed.
- When Enter is pressed, newline is added to stream and is consumed by second getchar(), so **`inChar2`** is set to **`'\n'`**.

Output Buffering

```
putchar('a');
/* generate some delay */
for (i=0; i<DELAY; i++) sum += i;

putchar('b');
putchar('\n');
```

- User doesn't see any character output until after the delay.
- `'a'` is added to the stream before the delay, but the stream is not flushed (displayed) until `'\n'` is added.

Formatted I/O

- Printf and scanf allow conversion between ASCII representations and internal data types.
- Format string contains text to be read/written, and formatting characters that describe how data is to be read/written.

%d	signed decimal integer
%f	signed decimal floating-point number
%x	unsigned hexadecimal number
%b	unsigned binary number
%c	ASCII character
%s	ASCII string

Special Character Literals

- Certain characters cannot be easily represented by a single keystroke, because they
  - correspond to whitespace (newline, tab, backspace, ...)
  - are delimiters for other literals (quote, double quote, ...)
- These are represented by the following sequences:

`\n`	newline
`\t`	tab
`\b`	backspace
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\0`*nnn*	ASCII code *nnn* (in octal)
`\x`*nnn*	ASCII code *nnn* (in hex)

printf

- Prints its first argument (format string) to stdout with all formatting characters replaced by the ASCII representation of the corresponding data argument.

```
int a = 100;
int b = 65;
char c = 'z';
char banner[10] = "Hola!";
double pi = 3.14159;

printf("The variable 'a' decimal: %d\n", a);
printf("The variable 'a' hex: %x\n", a);
printf("The variable 'a' binary: %b\n", a);
printf("'a' plus 'b' as character: %c\n", a+b);
printf("A char %c.\t A string %s\n A float %f\n",
       c, banner, pi);
```

Missing Data Arguments

- What happens when you don't provide a data argument for every formatting character?

```
printf("The value of nothing is %d\n");
```

- **%d** will convert and print whatever is on the stack in the position where it expects the first argument.

  In other words, something will be printed, but it will be a garbage value as far as our program is concerned.

- Reads ASCII characters from stdin,
  matching characters to its first argument (format string),
  converting character sequences according to any
  formatting characters, and storing the converted values
  to the addresses specified by its data pointer arguments.

```
char name[100];
int bMonth, bDay, bYear;
double gpa;

scanf("%s %d/%d/%d %lf",
        name, &bMonth, &bDay, &bYear, &gpa);
```

scanf Conversion

- For each data conversion, scanf will skip whitespace characters and then read ASCII characters until it encounters the first character that should NOT be included in the converted value.

  %d Reads until first non-digit.

  %x Reads until first non-digit (in hex).

  %s Reads until first whitespace character.

- Literals in format string must match literals in the input stream.
- Data arguments must be pointers, because scanf stores the converted value to that memory address.

scanf Return Value

- The scanf function returns an integer, which indicates the number of successful conversions performed.
  - This lets the program check whether the input stream was in the proper format.
- Example:

```
scanf("%s %d/%d/%d %lf",
      name, &bMonth, &bDay, &bYear, &gpa);
```

Input Stream	*Return Value*
`Mudd 02/16/69 3.02`	`5`
`Muss 02 16 69 3.02`	`2`

Doesn't match literal '/', so scanf quits after second conversion.

Bad scanf Arguments

- Two problems with scanf data arguments

1. Not a pointer

```
int n = 0;
scanf("%d", n);
```

Will use the value of the argument as an address.

2. Missing data argument

```
scanf("%d");
```

Will get address from stack.

If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0). Otherwise, your program will just modify an arbitrary memory location, which can cause very unpredictable behavior.

Variable Argument Lists

- The number of arguments in a printf or scanf call depends on the number of data items being read or written.

Declaration of printf (from stdio.h):

```
int printf(const char*, ...);
```

- Recall calling sequence from Chapter 14
  - Parameters pushed onto stack from right to left.
  - This stack-based calling convention allows for a variable number of arguments, and fixed arguments (which are named first) are always the same offset from the frame ptr.

File I/O

- For our purposes, a file is a sequence of ASCII characters stored on some device.
- Allows us to process large amounts of data without having to type it in each time or read it all on the screen as it scrolls by.
- Each file is associated with a stream.
  - May be input stream or output stream (or both!).
- The type of a stream is a "file pointer", declared as:

```
FILE *infile;
```

- The **FILE** type is defined in <stdio.h>.

fopen

- The fopen **(pronounced "eff-open")** function associates a physical file with a stream.

```
FILE *fopen(char* name, char* mode);
```

- First argument: **`name`**
  - The name of the physical file, or how to locate it on the storage device. This may be dependent on the underlying operating system.
- Second argument: **`mode`**
  - How the file will be used:
    `"r"` -- read from the file
    `"w"` -- write, starting at the beginning of the file
    `"a"` -- write, starting at the end of the file (append)

fprintf and fscanf

- Once a file is opened, it can be read or written using **`fscanf()`** and **`fprintf()`**, respectively.

- These are just like **`scanf()`** and **`printf()`**, except an additional argument specifies a file pointer:

```
fprintf(outfile, "The answer is %d\n", x);

fscanf(infile, "%s %d/%d/%d %lf",
    name, &bMonth, &bDay, &bYear, &gpa);
```

fprintf and fscanf

```
float f1, f2;
  int i1, i2;
  FILE *my_stream;
  char my_filename[] = "snazzyjazz.txt";

  my_stream = fopen (my_filename, "w");
  fprintf (my_stream, "%f %f %#d %#d", 23.5, -12e6, 100, 5);

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  my_stream = fopen (my_filename, "r");
  fscanf (my_stream, "%f %f %i %i", &f1, &f2, &i1, &i2);

  /* Close stream; skip error-checking for brevity of example */
  fclose (my_stream);

  printf ("Float 1 = %f\n", f1);
  printf ("Float 2 = %f\n", f2);
  printf ("Integer 1 = %d\n", i1);
  printf ("Integer 2 = %d\n", i2);
```

This code example prints the following output on the screen:

Float 1 = 23.500000
Float 2 = -12000000.000000
Integer 1 = 100
Integer 2 = 5

If you examine the text file snazzyjazz.txt, you will see it contains the following text:

23.500000 -12000000.000000 100 5