Input / Output

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

`<stdio.h>`: I/O Functions

- *Buffer*: area of memory used to reduce number of expensive system calls
  - i.e., get input and write output in blocks or chunks
- *Stream*: source of data being read, or destination of data being written
  - (actually, a file descriptor/handle + a buffer)
- Two types of streams
  1. text, ASCII characters, structured as lines terminated by `'\n'`
  2. binary, sequence of bytes with no particular structure

`<stdio.h>` … (cont'd)

- Every C program begins execution with 3 streams
  - **`stdin`**, **`stdout`**, and **`stderr`**
- The program does not need to open or close these streams; happens automatically

Input Redirection

- We can change the location that a program's **`stdin`**, **`stdout`**, and/or **`stderr`** streams read from or write to.
- One way to do this is by re-directing one or all of these to read or write to a file.

`<` redirects a **`stdin`** to **read from** a file.
Example: `./a.out < infile.txt`

`>` redirects a **`stdout`** to **write to** a file.
Example: `./a.out > outfile.txt`

Input Redirection (cont'd)

- You can **redirect** the standard **input from a file**, e.g.,

```
pgm99 < infile.txt
```

- You can **redirect** the standard **output to a file**, e.g.,

```
pgm99 > outfile.txt
```

- **Note**: the **EOF** (end of file) character on your keyboard is either **`Ctrl-d`** (Unix, Linux, Mac OS X) or **`Ctrl-z`** (Windows)

Formatted Output

The format string passed to the **`printf`** function can include formatting placeholders and special characters (e.g., **`\t`**)that create special formatting. For example, the following code:

```
int y = 10;
float pi = 3.14;
printf("%g \t %s \t %d\n", pi, "hello", y);
```

Produces the following output:

```
3.14    hello    10
```

Formatting Placeholders for Common C Types

`%f, %g`	**placeholders for a float or double value**
`%d`	placeholder for a decimal value (char, short, int)
`%u`	placeholder for an unsigned decimal
`%c`	placeholder for a single character
`%s`	placeholder for a string value
`%p`	placeholder to print an address value
`%ld`	placeholder for a long value
`%lu`	placeholder for an unsigned long value
`%lld`	placeholder for a long long value
`%llu`	placeholder for an unsigned long long value

Specifying Field Width - Examples

`%5.3f` prints float value in space 5 chars wide, with 3 places beyond decimal

`%20s` prints the string value in a field of 20 chars wide, right justified

`%-20s` prints the string value in a field of 20 chars wide, left justified

`%8d` prints the int value in a field of 8 chars wide, right justified

`%-8d` prints the int value in a field of 8 chars wide, left justified

Placeholders to Specify Different Representations

`%x`	prints value in hexadecimal (base 16)
`%o`	prints value in octal (base 8)
`%d`	prints value in signed decimal (base 10)
`%u`	prints value in unsigned decimal (unsigned base 10)
`%e`	prints float or double in scientific notation

***Note**: There is no formatting option to display a value in binary.*

File Input/Output

<stdio.h> fopen()

FILE * fopen(const char *filename, const char *mode)

Establishes a connection between a file or device and a stream

Returns pointer to object of type **FILE**, records information for controlling stream

- returns **NULL** on failure

```
FILE * infile;
infile = fopen("/tmp/testfile.txt", "r");
if (infile == NULL)
    { (void) printf("Error.\n"); return -1;}
```

`<stdio.h>` `fopen()` (cont'd)

- Mode
  - "**`r`**" - open for reading
  - "**`w`**" - create file for writing (discard previous contents)
  - "**`a`**" - append to existing file or create for writing
  - (+ some others, less important)
- If '**`b`**' appended to above modes, file is opened as binary file

`<stdio.h>` Binary Files

- Needed if
  - non-ASCII data, or
  - need to handle differences between outputs produced by different platforms (e.g., Windows ↔ Linux)
- Examples of binary files
  - images: .bmp, .gif, .jpg, .tif
  - audio: .wav, .ac3
  - video: .avi
  - word processing: .rtf
  - encrypted files
  - etc.

<stdio.h> fgetc()

```
int fgetc(FILE *stream)
int getc(FILE *stream)
```

Read next character of stream as **unsigned char** (converted to **int**)

returns **EOF** if end of file or error

getchar() is equivalent to **getc(stdin)**

```
int res;
unsigned char c;
if ((res = getc(stdin)) == EOF)
    …take action here…
c = (unsigned char) res;
```

<stdio.h> fputc()

```
int fputc(int c, FILE *stream)
int putc(int c, FILE * stream)
```

Write the character **c** (converted to **unsigned char**) to **stream**

Returns character written, or **EOF** on error

putchar(c) equivalent to **putc(c, stdout)**

```
(void) putc('H', stdout);
(void) putc('I', stdout);
(void) putc('!', stdout);
```

<stdio.h> ungetc()

```
int ungetc(int c, FILE * stream)
```

Pushes **c** (converted to **unsigned char**) back onto **stream** !

- Clears the stream's end-of-file indicator.
- **c** will be read by next **getc** on **stream**

Only one character of pushback per stream is *guaranteed*

EOF may not be pushed back

Returns character pushed back, **EOF** on error

(see **ungetc_example.c** in *Code samples and Demonstrations* in *Canvas*)

<stdio.h> ungetc() … (cont'd)

- This program reads input words, prints one word per line
- No spaces between words, but each new word starts with a capital letter (e.g. "**DogCatBirdFishBee**")

```
char s[100], *p = s;
while (((*p=getc(stdin)) != EOF) && (*p != '\n'))
  if ((p > s) && (isupper(*p))) {
    ungetc(*p, stdin);  /* read one too many */
    *p = '\0';
    (void) printf("Word: %s\n", s);
    p = s;
  }
  else
    p++;
(void) printf("Word: %s\n", s);
```

(see **ungetc_example.c** in *Code samples and Demonstrations* in *Canvas*)

<stdio.h> fread()

```
size_t fread (void * ptr, size_t size,
              size_t nobj, FILE * stream)
```

Reads up to **nobj** objects of size **size** from **stream** into array pointed to by **ptr**

Returns number of objects read, less if error

(see **fread.c** in *Code samples and Demonstrations* in *Canvas*)

```
char items[NUMITEMS];
size_t nr = fread((void *) items, sizeof(char),
                  (size_t) NUMITEMS, stdin);
if (nr != NUMITEMS)
    … do something here …
```

<stdio.h> fwrite()

```
size_t fwrite (const void * ptr, size_t size,
               size_t nobj, FILE * stream)
```

Writes up to **nobj** objects of size **size** starting at address **ptr** to **stream**

Returns number of objects written, less than requested if error

<stdio.h> fseek()

```
int fseek (FILE *stream, long offset,
           int origin)
```

Sets file position (for subsequent reading or writing) to **offset** from **origin**

origin may be **SEEK_SET** (beginning of file), **SEEK_CUR** (current position), or **SEEK_END** (end of file)

Mainly for binary streams

Returns non-zero on error

<stdio.h> fseek() … (cont'd)

```
int res = fseek(infile, (long) 1000, SEEK_SET);
c = getc(infile);  /* now read 1001st byte */

int res = fseek(infile, (long) -5, SEEK_END);
c = getc(infile);  /* read 5th byte from end */
```

<stdio.h> fflush()

```
int fflush(FILE *stream)
```

Causes any buffered data to be immediately written to output file

Helpful if you don't want to wait for `'\n'` to see output

```
fflush(stdout);
```

Or if you want to discard all the input typed by the user so far

```
fflush(stdin);
```

<stdio.h> fclose()

```
int fclose(FILE * stream)
```

Actions

- flush any unwritten data to output file or device
- close the stream (cannot be read or written after)

```
(void) fclose(outfile);
```

<stdio.h> remove()

```
int remove(const char *filename)
```

Delete the named file, return **0** if successful

```
if (remove("/tmp/testfile.txt"))
    …error, take action here…
```

<stdio.h> fscanf()

```
int fscanf(FILE *stream, const char *fmt, …)
```

Like **scanf**, but specify stream to be read from

- **scanf(fmt, args…)** is same as
  fscanf(stdin, fmt, args…)

```
int sscanf(char * s, const char *fmt, …)
```

Like **scanf**, but … scans from a **string** instead of a file!

<stdio.h> fprintf()

```
int fprintf(FILE *stream,
            const char *fmt, …)
```

Like **printf**, but specify stream to be written to

printf(fmt, args…) is same as
frintf(stdin, fmt, args…)

```
int sprintf(char * s, FILE *stream,
            const char *fmt, …)
```

Like **printf**, but … prints to a **string** instead of a file!

`<stdio.h>` I/O Error Functions

```
int feof(FILE *stream)
```

Returns non-zero if **`EOF`** for **`stream`** has been reached

```
int ferror(FILE *stream)
```

Returns non-zero if error indicator for **`stream`** is set

```
void clearerr(FILE *stream)
```

Clears previously set error indicator for **`stream`**

- errors are not cleared unless programmer **explicitly** uses **`clearerr`**

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.