Basic Input / Output in C

Systems Programming

Streams

- File input / output in C is stream-based
  - » Following the UNIX paradigm already discussed in the first part of the course

- The C data structure that represents a stream is called **`FILE`** rather than "stream“

- **`FILE`**
  - » Datatype used to represent stream objects
  - » Holds all of the internal state information about the connection to the associated file
    - File position indicator
    - Buffering information
    - Error and end-of-file status indicators
      - We can test for them using the **`ferror`** and **`feof`** functions – more on this later

Standard Streams

These streams are declared in the header file `<stdio.h>`

`FILE * stdin`

» The standard input stream, which is the normal source of input for the program

`FILE * stdout`

» The standard output stream, which is used for normal output from the program

`FILE * stderr`

» The standard error stream, which is used for error messages and diagnostics issued by the program

Opening Streams

- Opening a file with the `fopen` function creates a new stream and establishes a connection between the stream and a file
  - This may involve creating a new file

```
FILE * fopen (const char *filename, const char *opentype)
```

- The `opentype` argument controls how the file is opened and specifies attributes of the resulting stream. It must begin with one of the following sequences of (mode) characters:
  - **`r`** - Open an existing file for reading only
  - **`w`** - Open the file for writing only. If the file already exists, it is truncated to zero length. Otherwise a new file is created.
  - **`a`** - Open a file for append access. If the file already exists, its initial contents are unchanged and output to the stream is appended to the end of the file. Otherwise, a new, empty file is created.
  - **`r+`** - Open an existing file for both reading and writing. The initial contents of the file are unchanged and the initial file position is at the beginning of the file.
  - **`w+`** - Open a file for both reading and writing. If the file already exists, it is truncated to zero length. Otherwise, a new file is created.
  - **`a+`** - Open or create file for both reading and appending. If the file exists, its initial contents are unchanged. Otherwise, a new file is created. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

Closing Streams (1/2)

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

» This function causes *`stream`* to be closed and the connection to the corresponding file to be broken. Any buffered output is written, and any buffered input is discarded. The `fclose` function returns a value of `0` if the file was closed successfully, and `EOF` if an error was detected. It is often important to check for errors when you call `fclose` to close an output stream, because real, everyday errors can be detected at this time. For example, when `fclose` writes the remaining buffered output, it might get an error because the disk is full. Even if you know the buffer is empty, errors can still occur when closing a file if you are using NFS (or any other file/file system across a network).

`int` **`fcloseall`** *`(void)`*

» This function causes all open streams of the process to be closed and the connection to corresponding files to be broken. All buffered data is written, and any buffered input is discarded. The `fcloseall` function returns a value of `0` if all the files were closed successfully, and `EOF` if an error was detected. This function should be used only in special situations, e.g. when an error occurred and the program must be aborted. Normally each single stream should be closed separately so that problems with individual streams can be identified. It is also problematic since the standard streams will be closed too.

Closing Streams (2/2)

- If the `main` function to your program returns, or if you call the `exit` function, all open streams are automatically closed properly

- If your program terminates in any other manner, such as by calling the `abort` function or from a fatal signal, open streams might not be closed properly. Buffered output might not be flushed and files may be incomplete.

End-Of-File and Errors

`int EOF`

» This macro is an integer value that is returned by a number of stream functions to indicate an end-of-file condition, or some other error situation. With the GNU library, `EOF` is `-1`. In other libraries, its value may be some other negative number.

`int feof (FILE *stream)`

» The `feof` function returns nonzero if and only if the end-of-file **indicator** for the stream *`stream`* is set

`int ferror (FILE *stream)`

» The `ferror` function returns nonzero if and only if the error indicator for the stream *stream* is set, indicating that an error has occurred on a previous operation on the stream

- In addition to setting the error indicator associated with the stream, the functions that operate on streams also set `errno` in the same way as the corresponding low-level functions that operate on file descriptors. For example, all of the functions that perform output to a stream - such as `fputc`, `printf`, and `fflush` - are implemented in terms of `write`, and all of the `errno` error conditions defined for write are meaningful for these functions.

Simple Output by Characters or Lines

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

» The `fputc` function converts the character *`c`* to type unsigned char, and writes it to the stream *`stream`*. `EOF` is returned if a write error occurs; otherwise the character *`c`* is returned.

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

» This is just like `fputc`, except that most systems implement it as a macro, making it faster. One consequence is that it may evaluate the *`stream`* argument more than once, which is an exception to the general rule for macros. `putc` is usually the best function to use for writing a single character.

`int` **`putchar`** *`(int c)`*

» The `putchar` function is equivalent to `putc` with `stdout` as the value of the *`stream`* argument.

`int` **`fputs`** *`(const char *s, FILE *stream)`*

» The function `fputs` writes the string *s* to the stream *`stream`*. The terminating nul character is not written. This function does **not** add a newline character, either. It outputs only the characters in the string. This function returns `EOF` if a write error occurs, and otherwise a non-negative value.

`int` **`puts`** *`(const char *s)`*

» The puts function writes the `string` *s* to the stream `stdout` followed by a newline. The terminating nul character of the string is not written. (Note that `fputs` does *not* write a newline, whereas this function does!) `puts` is the most convenient function for printing simple console messages.

Character Input

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

- This function reads the next character as an unsigned char from the stream *`stream`* and returns its value, converted to an `int`. If an end-of-file condition or read error occurs, `EOF` is returned instead.
  - 0…255 → Actual data byte; -1 (or other) → EOF

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

- This is just like `fgetc`, except that it is permissible (and typical) for it to be implemented as a macro that evaluates the *`stream`* argument more than once. `getc` is often highly optimized, so it is usually the best function to use to read a single character.

`int` **`getchar`** *`(void)`*

- The `getchar` function is equivalent to `getc` with `stdin` as the value of the *stream* argument

Line-oriented Input (1/3)

`char * fgets (char *s, int count, FILE *stream)`

» The `fgets` function reads characters from the stream *`stream`* up to and including a newline character and stores them in the string *`s`*, adding a nul character to mark the end of the string. You must supply *`count`* characters worth of space in *`s`*, but the number of characters read is at most *`count`*`−1`. The extra character space is used to hold the nul character at the end of the string. If the system is already at end of file when you call `fgets`, then the contents of the array *`s`* are unchanged and a null pointer is returned. A null pointer is also returned if a read error occurs. Otherwise, the return value is the pointer *`s`*. **Warning:** If the input data has a nul character, you can't tell. So don't use `fgets` unless you know the data cannot contain a nul.

`char * gets (char *s)`

» **Deprecated function.** The function `gets` reads characters from the stream `stdin` up to the next newline character, and stores them in the string *`s`*. The newline character is discarded (note that this differs from the behavior of `fgets`, which copies the newline character into the string). If `gets` encounters a read error or end-of-file, it returns a null pointer; otherwise it returns *`s`*.

» **Warning:** The `gets` function is **very dangerous** because it provides no protection against overflowing the string *`s`*. You should **always** use `fgets` or `getline` instead. To remind you of this, the linker (if using GNU ld) will issue a warning whenever you use `gets`.

Line-oriented Input (2/3)

`ssize_t getline` *`(char **lineptr, size_t *n,`*
`FILE *stream)`

- » This function reads an entire line from *`stream`*, storing the text (including the newline and a terminating nul character) in a buffer and storing the buffer address in `*`*`lineptr`*. Before calling `getline`, you should place in `*`*`lineptr`* the address of a buffer `*`*`n`* bytes long, allocated with `malloc`. If this buffer is long enough to hold the line, `getline` stores the line in this buffer. Otherwise, `getline` makes the buffer bigger using `realloc`, storing the new buffer address back in `*`*`lineptr`* and the increased size back in `*`*`n`*.
- » If you set `*`*`lineptr`* to a null pointer, and `*`*`n`* to zero, before the call, then `getline` allocates the initial buffer for you by calling `malloc`. In either case, when `getline` returns, `*`*`lineptr`* is a `char *` which points to the text of the line.
- » When `getline` is successful, it returns the number of characters read (including the newline, but not including the terminating nul). This value enables you to distinguish nul characters that are part of the line from the nul character inserted as a terminator.
- » This function is a **GNU extension**, but it is the recommended way to read lines from a stream. The alternative standard functions are unreliable.
- » If an error occurs or end of file is reached without any bytes read, `getline` returns `-1`.

Line-oriented Input (3/3)

```
ssize_t getdelim (char **lineptr, size_t *n,
                  int delimiter, FILE *stream)
```

- This function is like `getline` except that the character which tells it to stop reading is not necessarily newline. The argument *`delimiter`* specifies the delimiter character; `getdelim` keeps reading until it sees that character (or end of file). The text is stored in *`lineptr`*, including the delimiter character and a terminating nul. Like `getline`, `getdelim` makes *`lineptr`* bigger if it isn't big enough.
- `getline` is in fact implemented in terms of `getdelim`, just like this:

```
ssize_t getline (char **lineptr, size_t *n, FILE *stream)  {
 return getdelim (lineptr, n, '\n', stream);
}
```

Block Input/Output

```
size_t fread (void *data, size_t size, size_t count,
              FILE *stream)
```

- This function reads up to `count` objects of size `size` into the array *data*, from the stream `stream`. It returns the number of objects actually read, which might be less than `count` if a read error occurs or the end of the file is reached. This function returns a value of zero (and doesn't read anything) if either `size` or `count` is zero. If `fread` encounters end of file in the middle of an object, it returns the number of complete objects read, and discards the partial object. Therefore, the stream remains at the actual end of the file.

```
size_t fwrite (const void *data, size_t size,
               size_t count, FILE *stream)
```

- This function writes up to `count` objects of size `size` from the array `data`, to the stream `stream`. The return value is normally `count`, if the call succeeds. Any other value indicates some sort of error, such as running out of space.

Formatted Output (1/9)

- The **`printf`** function can be used to print any number of arguments. The template string argument you supply in a call provides information not only about the number of additional arguments, but also about their types and what style should be used for printing them.

- Ordinary characters in the template string are simply written to the output stream as-is, while *conversion specifications* introduced by a `%` character in the template cause subsequent arguments to be formatted and written to the output stream. For example,

```
int pct = 37;
char filename[] = "foo.txt";
printf ("Processing of '%s' is %d%% finished.\nPlease be
patient.\n", filename, pct);
```

  produces output like

```
Processing of 'foo.txt' is 37% finished.
Please be patient.
```

- This example shows the use of the **`%d`** conversion to specify that an **`int`** argument should be printed in decimal notation, the **`%s`** conversion to specify printing of a string argument, and the `%%` conversion to print a literal `%` character.

Formatted Output (2/9)

- Output conversion syntax
  - » The conversion specifications in a **`printf`** template string have the general form:

```
% [ param-no $] flags width [ . precision ] type conversion
```

  - » In more detail, output conversion specifications consist of an initial % character followed in sequence by:
    - An optional specification of the parameter used for this format
    - Zero or more *flag characters* that modify the normal behaviour of the conversion specification
    - An optional decimal integer specifying the *minimum field width*
    - An optional *precision* to specify the number of digits to be written for the numeric conversions. If the precision is specified, it consists of a period (.) followed optionally by a decimal integer (which defaults to zero if omitted).
    - An optional *type modifier character*, which is used to specify the data type of the corresponding argument if it differs from the default type
    - A character that specifies the conversion to be applied

Formatted Output (3/9)

- Common output conversions

`%d, %i, %ld, %lld`	Print an integer as a signed decimal number; %d and %i are synonymous for output, but are different when used with scanf for input; %ld is for long, %lld for long long
`%o`	Print an integer as an unsigned octal number
`%u, %lu, %llu`	Print a integer (long/long long) as an unsigned decimal number
`%x, %X`	Print an integer as an unsigned hexadecimal number
`%f`	Print a floating-point number in normal (fixed-point) notation
`%e, %E`	Print a floating-point number in exponential notation
`%g, %G`	Print a floating-point number in either normal or exponential notation, whichever is more appropriate for its magnitude
`%a, %A`	Print a floating-point number in a hexadecimal fractional notation which the exponent to base 2 represented in decimal digits
`%c, %C`	Print a single character. %C is an alias for %lc (supported for Unix std. compatibility)
`%s, %S`	Print a string. %S is an alias for %ls which is supported for Unix std. compatibility
`%p`	Print the value of a pointer (=address)
`%n`	Get the number of characters printed so far (never produces any output)
`%%`	Print a literal % character

Formatted Output (4/9)

- Integer conversions
  - Flags:

`-`	Left-justify the result in the field (instead of the normal right-justification)
`+`	For the signed **`%d`** and **`%i`** conversions, print a plus sign if the value is positive
`<Space>`	For the signed **`%d`** and **`%i`** conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the **`+`** flag ensures that the result includes a sign, this flag is ignored if you supply both of them.
`#`	For the **`%o`** conversion, this forces the leading digit to be **`0`**, as if by increasing the precision. For **`%x`** or **`%X`**, this prefixes a leading **`0x`** or **`0X`** (respectively) to the result. This doesn't do anything useful for the **`%d`**, **`%i`**, or **`%u`** conversions.
`0`	Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base. This flag is ignored if the **`-`** flag is also specified, or if a precision is specified.

  - If a precision is supplied, it specifies the minimum number of digits to appear; leading zeros are produced if necessary
  - If you don't specify a precision, the number is printed with as many digits as it needs
  - If you convert a value of zero with an explicit precision of zero, then no characters at all are produced

Formatted Output (5/9)

- Floating point conversions
  - » Flags:

–	Left-justify the result in the field (instead of the normal right-justification).
+	Always include a plus or minus sign in the result.
`<Space>`	If the result doesn't start with a plus or minus sign, prefix it with a space instead. Since the **+** flag ensures that the result includes a sign, this flag is ignored if you supply both of them.
#	Specifies that the result should always include a decimal point, even if no digits follow it. For the **`%g`** and **`%G`** conversions, this also forces trailing zeros after the decimal point to be left in place where they would otherwise be removed.
0	Pad the field with zeros instead of spaces; the zeros are placed after any sign. This flag is ignored if the **–** flag is also specified.

  - » The precision specifies how many digits follow the decimal-point character for the **`%f`**, **`%e`**, and **`%E`** conversions. For these conversions, the default precision is 6. If the precision is explicitly **`0`**, this suppresses the decimal point character entirely. For the **`%g`** and **`%G`** conversions, the precision specifies how many significant digits to print. Significant digits are the first digit before the decimal point, and all the digits after it. If the precision is **`0`** or not specified for **`%g`** or **`%G`**, it is treated like a value of **`1`**. If the value being printed cannot be expressed accurately in the specified number of digits, the value is rounded to the nearest number that fits.

Formatted Output (6/9)

- Examples

```
"|%5d|%-5d|%+5d|%+-5d|% 5d|%05d|%5.0d|%5.2d|%d|\n" →
|    0|0    |   +0|+0   |    0|00000|     |   00|0|
|    1|1    |   +1|+1   |    1|00001|    1|   01|1|
|   -1|-1   |   -1|-1   |   -1|-0001|   -1|  -01|-1|
|100000|100000|+100000|+100000| 100000|100000|100000|100000|100000|
```

```
"|%13.4a|%13.4f|%13.4e|%13.4g|\n" →
|  0x0.0000p+0|       0.0000|   0.0000e+00|            0|
|  0x1.0000p-1|       0.5000|   5.0000e-01|          0.5|
|  0x1.0000p+0|       1.0000|   1.0000e+00|            1|
| -0x1.0000p+0|      -1.0000|  -1.0000e+00|           -1|
|  0x1.9000p+6|     100.0000|   1.0000e+02|          100|
|  0x1.f400p+9|    1000.0000|   1.0000e+03|         1000|
| 0x1.3880p+13|   10000.0000|   1.0000e+04|        1e+04|
| 0x1.81c8p+13|   12345.0000|   1.2345e+04|    1.234e+04|
| 0x1.86a0p+16|  100000.0000|   1.0000e+05|        1e+05|
| 0x1.e240p+16|  123456.0000|   1.2346e+05|    1.235e+05|
```

Formatted Output (7/9)

- Formatted output functions (in **`<stdio.h>`**)

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

  - The **`printf`** function prints the optional arguments under the control of the template string *template* to the stream **`stdout`**. It returns the number of characters printed, or a negative value if there was an output error.

```
int fprintf (FILE *stream, const char *template, ...)
```

  - This function is just like **`printf`**, except that the output is written to the stream ***`stream`*** instead of **`stdout`**

```
int sprintf (char *s, const char *template, ...)
```

  - This is like **`printf`**, except that the output is stored in the character array ***`s`*** instead of written to a stream. A nul character is written to mark the end of the string. The **`sprintf`** function returns the number of characters stored in the array **`s`**, not including the terminating nul character.
  - The behaviour of this function is undefined if copying takes place between objects that overlap--for example, if **`s`** is also given as an argument to be printed under control of the **`%s`** conversion.
  - **Warning:** The **`sprintf`** function can be **dangerous** because it can potentially output more characters than can fit in the allocation size of the string **`s`**. Remember that the field width given in a conversion specification is only a *minimum* value. To avoid this problem, you can use **`asprintf`**, described below.

Formatted Output (8/9)

- Dynamically allocating formatted output

```
int asprintf (char **ptr, const char *template, ...)
```

  - This function is similar to **`sprintf`**, except that it dynamically allocates a string to hold the output, instead of putting the output in a buffer you allocate in advance. The **`ptr`** argument should be the *address* of a **`char *`** object, and **`asprintf`** stores a pointer to the newly allocated string at that location. The return value is the number of characters allocated for the buffer, or less than zero if an error occurred. Usually this means that the buffer could not be allocated.
    - Do not forget to release this memory (→ free) allocated for you by the function!
  - This function is a **GNU extension**

  - An example of using asprintf :

```
char * make_message (char *name, char *value) {
   char *result = NULL;
   if (asprintf (&result, "value of %s is %s", name, value) < 0)
       return NULL;
   return result;
}
```

Formatted Output (9/9)

- Saving yourselves a few hours of sleep…
  - When compiling with the **`-Wformat`** option, the GNU C compiler checks calls to **`printf`** and related functions. It examines the format string and verifies that the **correct number** and **types of arguments** are supplied.
    - This will NOT catch buffer overflows (compile-time check, not runtime!)
      → You still have to use asprintf (or similar functions)!
  - This is also true for the **`scanf`** family of functions, which we will see next

Formatted input (1/7)

- Calls to `scanf` are superficially similar to calls to printf in that arbitrary arguments are read under the control of a template string

- While the syntax of the conversion specifications in the template is very similar to that for `printf`, the interpretation of the template is oriented more towards free-format input and simple pattern matching, rather than fixed-field formatting:
  - Most `scanf` conversions skip over any amount of "white space" in the input file, and there is no concept of precision for the numeric input conversions as there is for the corresponding output conversions
  - Ordinarily, non-whitespace characters in the template are expected to match characters in the input stream exactly, but a matching failure is distinct from an input error on the stream

- Another area of difference between `scanf` and `printf` is, that you must remember to supply pointers rather than immediate values as the optional arguments to scanf; the values that are read are stored in the objects that the pointers point to

Formatted input (2/7)

- When a matching failure occurs, `scanf` returns immediately, leaving the first non-matching character as the next character to be read from the stream
  - The normal return value from `scanf` is the number of values that were assigned, so you can use this to determine if a matching error happened before all the expected values were read
    - Note: “Values assigned” ≠ “Bytes/characters/… read”!

- The formatted input functions are not used as frequently as the formatted output functions. Partly, this is because it takes some care to use them properly. Another reason is that it is difficult to recover from a matching error.

Formatted input (3/7)

- Input conversion syntax

  - The conversion specifications in a scanf template string have the general form:

    ```
    % flags width type conversion
    ```

  - In more detail, an input conversion specification consists of an initial % character followed in sequence by:
    - An optional flag character *, which says to ignore the text read for this specification. When used, scanf reads input as directed by the rest of the conversion specification, but it discards this input, does not use a pointer argument, and does not increment the count of successful assignments.
    - An optional decimal integer that specifies the maximum field width. Reading of characters from the input stream stops either when this maximum is reached or when a non-matching character is found, whichever happens first. Most conversions discard initial whitespace characters (those that aren't explicitly documented), and these discarded characters don't count towards the maximum field width. String input conversions store a nul character to mark the end of the input; the maximum field width does not include this terminator.
    - An optional type modifier character. You can, e.g., specify a type modifier of l with integer conversions such as %d to specify that the argument is a pointer to a long int rather than a pointer to an int.
    - A character that specifies the conversion to be applied

Formatted input (4/7)

- Input conversions
  - » In general, the conversions are similar between **`printf`** and **`scanf`**. What you see here are just some of the important differences.

`%i`	Matches an optionally signed integer in **any** of the formats that the C language defines for specifying an integer constant
`%s`	Matches a string **containing only non-whitespace** characters
`%[`	Matches a string of characters that belong to a specified set
`%c`	Matches a string of one or more characters; the number of characters read is controlled by the maximum field width given for the conversion
`%p`	Matches a pointer value in the same implementation-defined format used by the %p output conversion for **`printf`**

Formatted input (5/7)

- String input conversions – additional information

  - To read in characters that belong to an arbitrary set of your choice, use the `%[` conversion. You specify the set between the `[` character and a following `]` character, using the same syntax used in regular expressions. As special cases:
    - A literal `]` character can be specified as the first character of the set
    - An embedded - character (that is, one that is not the first or last character of the set) is used to specify a range of characters
    - If a caret character `^` immediately follows the initial `[`, then the set of allowed input characters is everything except the characters listed

  - The `%[` conversion does not skip over initial whitespace characters

Formatted input (6/7)

- String input conversions – additional information (continued)

  - Here are some examples of `%[` conversions and what they mean:
    - `%25[1234567890]`

      Matches a string of up to 25 digits
    - `%25[][]`

      Matches a string of up to 25 square brackets
    - `%25[^ \f\n\r\t\v]`

      Matches a string up to 25 characters long that doesn't contain any of the standard whitespace characters
    - `%25[a-z]`

      Matches up to 25 lowercase characters

  - One more reminder: the `%s` and `%[` conversions are dangerous if you don't specify a maximum width, because input too long would overflow whatever buffer you have provided for it. No matter **how long** your buffer is, a user could always supply input that is **longer!**

Formatted input (7/7)

- Formatted input functions (in **`<stdio.h>`**)

```
int scanf (const char *template, ...)
```

» The **`scanf`** function reads formatted input from the stream **`stdin`** under the control of the template string *template*. The optional arguments are pointers to the places which receive the resulting values. The return value is normally the number of successful assignments. If an end-of-file condition is detected before any matches are performed, including matches against whitespace and literal characters in the template, then **`EOF`** is returned.

```
int fscanf (FILE *stream, const char *template, ...)
```

» This function is just like **`scanf`**, except that the input is read from the stream ***`stream`*** instead of **`stdin`**

```
int sscanf (const char *s, const char *template, ...)
```

» This is like **`scanf`**, except that the characters are taken from the nul-terminated string ***`s`*** instead of from a stream. Reaching the end of the string is treated as an end-of-file condition. The behaviour of this function is undefined if copying takes place between objects that overlap.