C Basics 7: Library I/O

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
   - **C standard library I/O (stdio)**
   - **default I/O streams**
   - **buffering**
   - **EOF**
   - **formatted I/O**
   - **formatting strings**
   - **reading lines**

C Basics 7: Library I/O

8. Errors and Error Handling
9. Functions and Parameter Passing
10. Dynamic and Static Memory
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Library I/O

Input/output (I/O) in C is accomplished using functions from the **stdio** component of the *C Standard Library*.

In order to use library I/O functions and associated symbols, a program must contain the following header include:

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

C **library I/O** is also referred to as **stream I/O** because an open file is referred to as a **stream**.

The **handles** for open files with library I/O are of type `FILE*` (a pointer to a `FILE` struct/structure).

A `FILE` struct contains information required for a *stream*, including its **file descriptor**, buffer (pointer), EOF/error indicators, etc.

Default I/O Streams

Three streams are automatically open for each process:

- **standard input** – symbol `stdin`;
- **standard output** – symbol `stdout`;
- **standard error** – symbol `stderr`.

By default, these three stream will all be associated with the *terminal*, so for example, input from `stdin` will come from the keyboard.

However, these three streams may be associated with other devices by the use of **piping** and **redirection** in the *shell command* that invokes a program (e.g., "`ls | ./prog > save`").

Library I/O Functions

Important library I/O functions:

- `fopen` – open a file
- `fclose` – close a stream/file
- `printf` – formatted output to `stdout`
- `fprintf` – formatted output to a stream
- `fscanf` – formatted input from a stream
- `getchar` – read a `char` from `stdin`
- `fgetc` – read a `char` from a stream
- `fgets` – read a line as a string from a stream
- `putchar` – write a `char` to `stdout`
- `fputc` – write a `char` to a stream
- `fputs` – write a string to a stream

Library I/O Functions (contd.)

Important library I/O functions (contd.):

- `fread` – binary input from a stream
- `fwrite` – binary output to a stream
- `fseek` – reposition read-write pointer/position in file stream
- `fflush` – flush buffer for a stream
- `feof` – check stream end-of-file status
- `ferror` – check stream error status

Library I/O Examples

Example of opening a file, reading a line, writing to file:

```
FILE *fptr = fopen("output.text","w");

char buffer[200];
printf("Enter line to save: ");
fgets(buffer,200,stdin);
int length = strlen(buffer);

fprintf(fptr,"Line is %d characters:\n%s\n",length,buffer);

fclose(fpntr);
```

I/O Buffering

A key property of library I/O is that I/O operations are **buffered** by default.

Being buffered means that reading/writing to disk/terminal is done in "chunks" for efficiency, even if the I/O functions are inputting or outputting individual characters.

C has three types of stream buffering:

- **fully buffered** – bytes are exchanged with the associated file/device in **blocks**
- **line buffered** – bytes are exchanged in **lines**, delimited by **newline chars** (`'\n'`)
- **unbuffered** – bytes are exchanged immediately

I/O Buffering (contd.)

By default, streams associated with *interactive* devices like *terminals* are set to be *line buffered*, while other devices (like files) are set to be *fully buffered*.

The buffering for a stream can be changed with `setvbuf()`:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

- `mode` must be one of:
  - `_IONBF` – unbuffered
  - `_IOLBF` – line buffered
  - `_IOFBF` – fully buffered

I/O Buffering (contd.)

Buffering means that when an output function like `fprintf()` is called, data will first be transferred to a buffer associated with the stream, and only under the right conditions will data from the buffer be output to the target file/device.

One consequence of buffering is that *prompts* that do not end with a *newline* may not be printed on the terminal immediately, causing confused output from a program.

The libary fuction `fflush()` can be used to flush buffered data:

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

- for output streams: all buffered data written to the target
- for input streams: all unconsumed buffered data discarded

EOF

An important stdio *constant symbol* is `EOF`.

`EOF` is a value that is returned by some stdio functions to indicate that **end-of-file** was encountered *or* that there was an *error*.

The two conditions can be distinguished with `feof()`/`ferror()`:

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

It is critical to understand that `EOF` is a value returned by some stdio functions—it is *not* a character/`char` (`'EOF'` is illegal).

Files *do not* have `EOF` or any other character as a **sentinel**.

Note that `EOF` is an `int`, typically defined as -1 (but test for `EOF`).

EOF (contd.)

The fact that `EOF` is an `int` and typically negative can lead to unexpected problems with **portable code**.

Whether `char` is signed or unsigned is *left to the implementation*, and can vary from one architecture to another.

This means that if we assign `EOF` (-1) to a `char`, it might be interpreted as -1 or it might be interpreted as 255.

To avoid this problem, always use `int` vars to capture the "character" return value of functions like `fgetc()` that can return `EOF`.

Formatted I/O

Among the most used library I/O functions are the "formatted output" functions, including:

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

The `format` parameter is a **format string**: a string constant that contains **format directives**.

Format directives:

- ordinary characters – copied directly to output, e.g., `"value:"`
- escape sequences – for special characters, e.g., `'\n'` (newline)
- **conversion specifications** – start with `%`, consume an argument, specify how to format argument, e.g., `%d`

Formatted I/O (contd.)

The most important part of a *conversion specification* is the **conversion specifier**: a single character that determines how the argument is to be formatted.

The most used conversion specifiers are:

- `d`, `i` – `int` argument formatted in decimal
- `f`, `F` – `double` argument formatted in decimal notation style `[-]ddd.ddd`
- `e`, `E` – `double` argument formatted in decimal notation style `[-]d.ddde±dd`
- `s` – `char*` string `char`'s written up to '\0'
- `c` – `int` converted to `unsigned char`, output ASCII character

Formatted I/O (contd.)

A *conversion specification* can also contain multiple *optional components* between the `%` and the *conversion specifier* character.

Optional components include (in the following order):

- zero or more **flags** (`#`, `0`, `-`, *space*, `+`, `'`)
- minimum **field width**
- **. precision**
- **length/type modifiers** (`h`, `hh`, `l`, `ll`, etc.)

Examples:

- `%8.2f` – print floating point in 8-wide field, 2 digits to right of decimal point
- `%05d` – print integer in (minimum) 5-wide field, padding with 0's on left if less than 5 digits

Formatted I/O (contd.)

Formatted *input* functions include:

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

While the format conversion specifications are simpler, the basic structure is similar.

For more detailed information see "`man fprintf`" or "`man fscanf`".

Formatting Strings

The I/O library provides related functions for formatting strings:

```
int sprintf(char *str, const char *format, ...)
int snprintf(char *str, size_t size, const char *format, ...)
```

These are generally a better choice for constructing complex strings than using string library functions such as `strncat`.

Example of constructing a file path string:

```
char filepath[PATH_MAX];
sprintf(filepath,"%s/out%d.data", getcwd(NULL,0), count);
```

Just remember that these functions *do not automatically allocate space* for the resulting string; you must do that ahead of time as with an array or `malloc()`.

Reading Lines

It not be too suprising to find out that C does *not* have a function that can read an *arbitrary length line* in as a string.

The closest function it does provide is `fgets`:

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

`fgets()` will read in a line as a string, but you must provide the string memory of the *required size*:
"`fgets()` reads in at most one less than `size` characters from `stream` and stores them into the buffer pointed to by `s`. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte (`'\0'`) is stored after the last character in the buffer."

Reading Lines (contd.)

If you know the maximum length of a line, `fgets()` works well:

```
char line[102];  //100-char line + newline + null-char
fgets(line,102,stdin);
```

If you do not, a complex loop and **dynamic memory** are required:

```
int incr = 10;
int total = incr + 1;
char *line = malloc(total);
int start = 0;
while (fgets(line+start,incr+1,stdin)!=NULL &&
       line[strlen(line)-1] != '\n') {
  start = total-1;
  total += incr;
  realloc(line,total);
}
```

getline() and getdelim()

While C does not provide functions that support reading lines of arbitrary length, two function were added to POSIX (2008) that do (so are available on Linux/UNIX systems).

They are: `getline()` and `getdelim()`:

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

- `getline()` reads an entire line from `stream`, storing the address of the buffer containing the text into `*lineptr`.
- `getdelim()` works similarly except that `delim` argument can specify a delimiter (other than newline).
- The buffer ends up null-char terminated and includes the newline/`delim` character (if one was found before file-end).

getline() and getdelim() (contd.)

They can allocate dynamic memory themselves, or accept pointers to already allocated dynamic memory (possibly expanding it):

- If `*lineptr` is NULL and `*n` is 0 when called, a new buffer will be allocated for storing the line
- Alternatively, `*lineptr` can contain a pointer to a `malloc()` allocated buffer of `*n` bytes. If the buffer is not large enough to hold the line, it will be resized (with `realloc()`). This may result in `*lineptr` and `*n` being changed to reflect updated address and size.
- If new buffer is to be allocated automatically, it must eventually be `free()`'d by the user program. Note that this is required even if `getline()`/`getdelim()` fails.

getline() and getdelim() (contd.)

Return values:

- Returns indicate the number of characters read, including the delimiter character, but not including the terminating null char.
- On failure to read a line (including file-end), -1 is returned; `errno` is set on error.

Though not technically part of the C standard I/O library, must `#include <stdio.h>` to use these functions.

You will probably also require a **feature test macro**, such as:

```
#define _POSIX_C_SOURCE 200809L
```

or

```
#define _GNU_SOURCE
```

getline() and getdelim() (contd.)

Example use of `getline()` from man page:

```
int main(void)
{
  FILE *stream;
  char *line = NULL;
  size_t len = 0;
  ssize_t read;

  stream = fopen("/etc/motd", "r");
  if (stream == NULL)
    exit(EXIT_FAILURE);

  while ((read = getline(&line, &len, stream)) != -1) {
    printf("Retrieved line of length %zu :\n", read);
    printf("%s", line);
  }

  free(line);
  fclose(stream);
  exit(EXIT_SUCCESS);
}
```

getline() and getdelim() (contd.)

The "`char **lineptr`" parameter declaration can be confusing.

It is a *pointer to a string* (`char*`) because this is how these functions pass back the buffer containing the line—i.e., it is a string *reference parameter* (in C++ terminology).

In the above example code, `line` is ultimately to contain the read line—i.e., to be a pointer to a `char` array containing the line.

This means the functions must make `line` point to the dynamic array they have newly allocated (holding the read line).

To do this, they must be given the address where `line`'s value (a pointer to `char` array) is stored, so they can set it to a new value: the pointer to the new dynamic array.