Input/Output

Dr Deepayan Bhowmik Dr Andrea Bracciali

Introduction

- C's input/output library is the biggest and most important part of the standard library.
- The `<stdio.h>` header is the primary repository of input/output functions, including `printf`, `scanf`, `putchar`, `getchar`, `puts`, and `gets`.

- We've looked at some of these functions.
- … and will introduce new functions, most of which deal with files.

Introduction

- In C99, some I/O functions belong to the `<wchar.h>` header. They deal with wide characters rather than ordinary characters.

- Functions in `<stdio.h>` that read or write data are known as byte input/output functions.

- Similar functions in `<wchar.h>` are called wide-character input/output functions.

Streams

- In C, the term stream means any source of input or any destination for output.

- Many programs obtain all their input from one stream (the keyboard) and write all their output to another stream (the screen).
- Larger programs may need additional streams.

- Streams often represent files stored on various media.

File Pointers

- Accessing a stream is done through a file pointer, which has type `FILE *`.

- The `FILE` type is declared in `<stdio.h>`.
- Certain streams are represented by file pointers with standard names.
- Additional file pointers can be declared as needed:

```
FILE *fp1, *fp2;
```

Standard Streams and Redirection

- `<stdio.h>` provides three standard streams:

File Pointer	*Stream*	*Default Meaning*
`stdin`	Standard input	Keyboard
`stdout`	Standard output	Screen
`stderr`	Standard error	Screen

- These streams are ready to use—we don't declare them, and we don't open or close them.

Text Files versus Binary Files

- `<stdio.h>` supports two kinds of files: text and binary.

- The bytes in a text file represent characters, allowing humans to examine or edit the file.
  - The source code for a C program is stored in a text file.

- In a binary file, bytes don't necessarily represent characters.
  - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
  - An executable C program is stored in a binary file.

Text Files versus Binary Files

- Text files have two characteristics that binary files don't possess: carriage return (`'\r'`) and line feed (`'\n'`) – other encoding are used.

- Text files are divided into lines. Each line in a text file normally ends with one or two special characters.
  - Windows: carriage-return character (`'\x0d'`) followed by line-feed character (`'\x0a'`)
  - UNIX and newer versions of Mac OS: line-feed character
  - Older versions of Mac OS: carriage-return character

TRY IT!

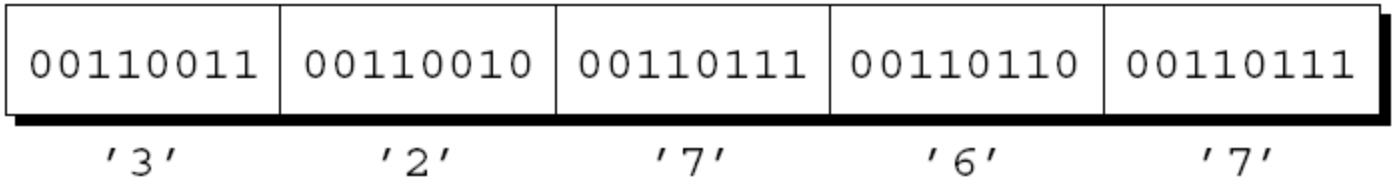
Text Files versus Binary Files

- Text files may contain a special "end-of-file" marker.
  - In Windows, the marker is `'\x1a'` (Ctrl-Z), but it is not required.
  - Most other operating systems, including UNIX, have no special end-of-file character.

- In a binary file, there are no end-of-line or end-of-file markers; all bytes are treated equally.

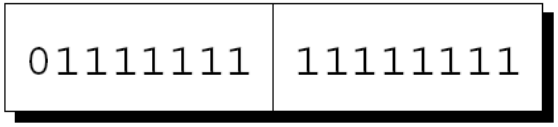
Text Files versus Binary Files

- When data is written to a file, it can be stored in text form or in binary form.
- One way to store the number 32767 in a file would be to write it in text form as the characters 3, 2, 7, 6, and 7:



Text Files versus Binary Files

- The other option is to store the number in binary, which would take as few as two bytes:



- Storing numbers in binary can often save space.

Opening a File

- Opening a file for use as a stream requires a call of the `fopen()` function.
- Prototype for `fopen()`:

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

- `filename` is the name of the file to be opened.
  - This argument may include information about the file's location, such as a drive specifier or path.
- `mode` is a "mode string" that specifies what operations we intend to perform on the file.

Opening a File

- In Windows, be careful when the file name in a call of `fopen` includes the `\` character.
- The call

```
fopen("c:\project\test1.dat", "r")
```

will fail, because `\t` is treated as a character escape.

- One way to avoid the problem is to use `\\` instead of `\`:

```
fopen("c:\\project\\test1.dat", "r")
```

- An alternative is to use the `/` character instead of `\`:

```
fopen("c:/project/test1.dat", "r")
```

Opening a File

- `fopen` returns a file pointer that the program can (and usually will) save in a variable:

```
FILE* fp;
fp = fopen("in.dat", "r");
  /* opens in.dat for reading */
```

- When it can't open a file, `fopen` returns a null pointer.

Modes

- Factors that determine which mode string to pass to `fopen`:
  - Which operations are to be performed on the file
  - Whether the file contains text or binary data

Modes

- Mode strings for text files:

String	***Meaning***
`"r"`	Open for reading
`"w"`	Open for writing (file need not exist)
`"a"`	Open for appending (file need not exist)
`"r+"`	Open for reading and writing, starting at beginning
`"w+"`	Open for reading and writing (truncate if file exists/create)
`"a+"`	Open for reading and writing (append if file exists)

Modes

- Mode strings for binary files:

String	*Meaning*
`"rb"`	Open for reading
`"wb"`	Open for writing (file need not exist)
`"ab"`	Open for appending (file need not exist)
`"r+b"` or `"rb+"`	Open for reading and writing, starting at beginning
`"w+b"` or `"wb+"`	Open for reading and writing (truncate if file exists)
`"a+b"` or `"ab+"`	Open for reading and writing (append if file exists)

Modes

- Note that there are different mode strings for writing data and appending data.

- When data is written to a file, it normally overwrites what was previously there.
- When a file is opened for appending, data written to the file is added at the end.

Closing a File

- The `fclose` function allows a program to close a file that it's no longer using.
- The argument to `fclose` must be a file pointer obtained from a call of `fopen`.
- `fclose` returns zero if the file was closed successfully.
- Otherwise, it returns the error code `EOF` (a macro defined in `<stdio.h>`).

Closing a File

- The outline of a program that opens a file for reading:

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

#define FILE_NAME "example.dat"

int main(void)
{
  FILE *fp;

  fp = fopen(FILE_NAME, "r");
  if (fp == NULL) {
    printf("Can't open %s\n", FILE_NAME);
    exit(EXIT_FAILURE);
  }
  …
  fclose(fp);
  return 0;
}
```

A Note on opening a File

- It's not unusual to see the call of `fopen` combined with the declaration of `fp`:

```
FILE *fp = fopen(FILE_NAME, "r");
```

or the test against `NULL`:

```
if ((fp = fopen(FILE_NAME, "r")) == NULL) …
```

Obtaining File Names from the Command Line

- Ways to supply file names to a program:
  - Building file names into the program doesn't provide much flexibility.
  - Prompting the user to enter file names can be awkward.
  - Having the program obtain file names from the command line is often the best solution.

- An example that uses the command line to supply two file names to a program named `demo`:

```
demo names.dat dates.dat
```

- In Netbeans, Project Properties > Run > Run Command, eg. `"${OUTPUT_PATH}" arg1 arg2`

Obtaining File Names from the Command Line

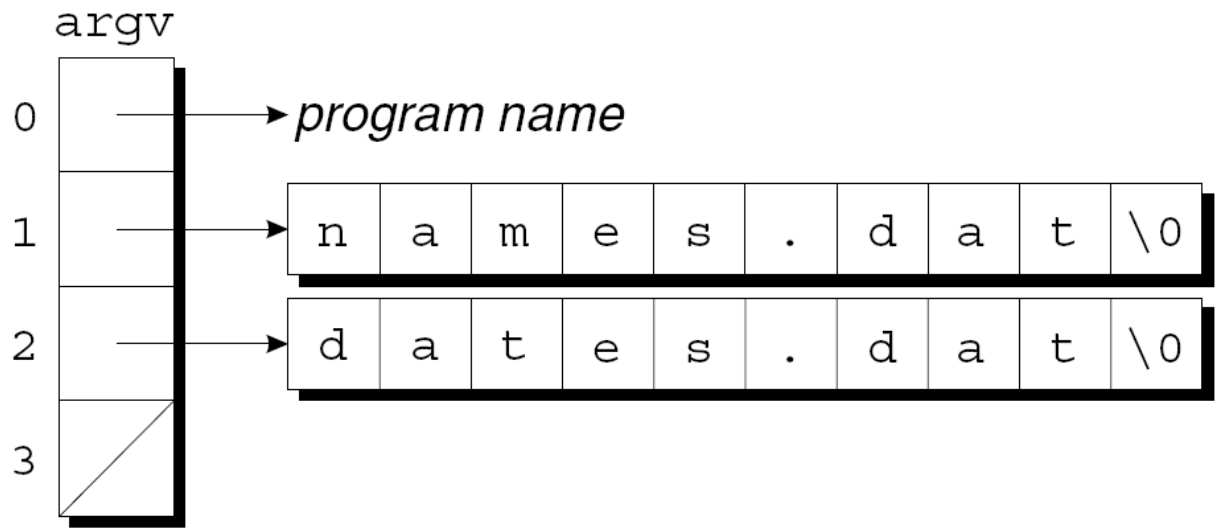
- Earlier lecture showed how to access command-line arguments by defining `main` as a function with two parameters:

```
int main(int argc, char *argv[])
{
  …
}
```

- `argc` is the number of command-line arguments.
- `argv` is an array of pointers to the argument strings.

Obtaining File Names from the Command Line

- `argv[0]` points to the program name, `argv[1]` through `argv[argc-1]` point to the remaining arguments, and `argv[argc]` is a null pointer.
- In the `demo` example, `argc` is 3 and `argv` has the following appearance:



Program: Checking Whether a File Can Be Opened

- The `canopen.c` program determines if a file exists and can be opened for reading.

- The user will give the program a file name to check:

  `canopen` *file*
- The program will then print either *file* `can be opened` or *file* `can't be opened`.
- If the user enters the wrong number of arguments on the command line, the program will print the message `usage: canopen filename`.

canopen.c

```
/* Checks whether a file can be opened for reading */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;

  if (argc != 2) {
    printf("usage: canopen filename\n");
    exit(EXIT_FAILURE);
  }

  if ((fp = fopen(argv[1], "r")) == NULL) {
    printf("%s can't be opened\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  printf("%s can be opened\n", argv[1]);
  fclose(fp);
  return 0;
}
```

Formatted I/O

- The next group of library functions use format strings to control reading and writing.
- `printf` and related functions are able to convert data from numeric form to character form during output.
- `scanf` and related functions are able to convert data from character form to numeric form during input.

The ...`printf` Functions

- The `fprintf` and `printf` functions write a variable number of data items to an output stream, using a format string to control the appearance of the output.

- The prototypes for both functions end with the `...` symbol (an ***ellipsis***), which indicates a variable number of additional arguments:

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

- Both functions return the number of characters written; a negative return value indicates that an error occurred.

The ...`printf` Functions

- `printf` always writes to `stdout`, whereas `fprintf` writes to the stream indicated by its first argument:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- A call of `printf` is equivalent to a call of `fprintf` with `stdout` as the first argument
- TRY IT

The ...`scanf` Functions

- `fscanf` and `scanf` read data items from an input stream, using a format string to indicate the layout of the input.

- After the format string, any number of pointers—each pointing to an object—follow as additional arguments.
- Input items are converted (according to conversion specifications in the format string) and stored in these objects.

The ...`scanf` Functions

- `scanf` always reads from `stdin`, whereas `fscanf` reads from the stream indicated by its first argument:

```
scanf("%d%d", &i, &j);
  /* reads from stdin */
fscanf(fp, "%d%d", &i, &j);
  /* reads from fp */
```

- A call of `scanf` is equivalent to a call of `fscanf` with `stdin` as the first argument.

Character I/O

- Library functions to read and write single characters.
- These functions work equally well with text streams and binary streams.
- The functions treat characters as values of type `int`, not `char`.

Character I/O

- Library functions to read and write single characters.

- `putchar` writes one character to the `stdout` stream:

```
putchar(ch);     /* writes ch to stdout */
```

- `fputc` write a character to an arbitrary stream:

```
fputc(ch, fp);  /* writes ch to fp */
```

Input Functions

- `getchar` reads a character from `stdin`:

```
ch = getchar();
```

- `fgetc` read a character from an arbitrary stream:

```
ch = fgetc(fp);
```

  The function treats the character as an `unsigned char` value (which is then converted to `int` type before it's returned).
- As a result, they never return a negative value other than `EOF`.

Program: Copying a File

- The `fcopy.c` program makes a copy of a file (code next!).
- The names of the original file and the new file will be specified on the command line when the program is executed.
- An example that uses `fcopy` to copy the file `f1.c` to `f2.c`:

```
fcopy f1.c f2.c
```

- `fcopy` will issue an error message if there aren't exactly two file names on the command line or if either file can't be opened.

Program: Copying a File

- Using `"rb"` and `"wb"` as the file modes enables `fcopy` to copy both text and binary files.

- If we used `"r"` and `"w"` instead, the program wouldn't necessarily be able to copy binary files.

fcopy.c

```
/* Copies a file */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *source_fp, *dest_fp;
  int ch;

  if (argc != 3) {
    fprintf(stderr, "usage: fcopy source dest\n");
    exit(EXIT_FAILURE);
  }
```

```
  if ((source_fp = fopen(argv[1], "rb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[1]);
    exit(EXIT_FAILURE);
  }

  if ((dest_fp = fopen(argv[2], "wb")) == NULL) {
    fprintf(stderr, "Can't open %s\n", argv[2]);
    fclose(source_fp);
    exit(EXIT_FAILURE);
  }

  while ((ch = fgetc(source_fp)) != EOF)
    fputc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Line I/O

- Library functions in the next group are able to read and write lines.
- These functions are used mostly with text streams, although it's legal to use them with binary streams as well.

Output Functions

- The `puts` function writes a string of characters to `stdout`:

```
puts("Hi, there!");  /* writes to stdout */
```

- After it writes the characters in the string, `puts` always adds a new-line character.

Output Functions

- `fputs` is a more general version of `puts`.
- Its second argument indicates the stream to which the output should be written:

```
fputs("Hi, there!", fp);  /* writes to fp */
```

- Unlike `puts`, the `fputs` function doesn't write a new-line character unless one is present in the string.
- Both functions return `EOF` if a write error occurs; otherwise, they return a nonnegative number.

Input Functions

- The `gets` function reads a line of input from `stdin`:

```
gets(str); /* reads a line from stdin */
```

- `gets` reads characters one by one, storing them in the array pointed to by `str`, until it reads a new-line character (which it discards).

- `fgets` is a more general version of `gets` that can read from any stream.
- `fgets` is also safer than `gets`, since it limits the number of characters that it will store.

Input Functions

- A call of `fgets` that reads a line into a character array named `str`:

```
fgets(str, sizeof(str), fp);
```

- `fgets` will read characters until it reaches the first new-line character or `sizeof(str)` – 1 characters have been read.
- If it reads the new-line character, `fgets` stores it along with the other characters.

Input Functions

- Both `gets` and `fgets` return a null pointer if a read error occurs or they reach the end of the input stream before storing any characters.

- Otherwise, both return their first argument, which points to the array in which the input was stored.
- Both functions store a null character at the end of the string.

Block I/O

- The `fread` and `fwrite` functions allow a program to read and write large blocks of data in a single step.

- `fread` and `fwrite` are used primarily with binary streams, although—with care—it's possible to use them with text streams as well.

Block I/O

- `fwrite` is designed to copy an array from memory to a stream.
- Arguments in a call of `fwrite`:
  - Address of array
  - Size of each array element (in bytes)
  - Number of elements to write
  - File pointer
- A call of `fwrite` that writes the entire contents of the array `a`:

```
fwrite(a, sizeof(a[0]),sizeof(a) / sizeof(a[0]), fp);
```

Block I/O

- `fwrite` returns the number of elements actually written.
- This number will be less than the third argument if a write error occurs.

Block I/O

- `fread` will read the elements of an array from a stream.
- A call of `fread` that reads the contents of a file into the array `a`:

```
n = fread(a, sizeof(a[0]),
          sizeof(a) / sizeof(a[0]), fp);
```

- `fread`'s return value indicates the actual number of elements read.
- This number should equal the third argument unless the end of the input file was reached or a read error occurred.

Block I/O

- `fwrite` is convenient for a program that needs to store data in a file before terminating.

- Later, the program (or another program) can use `fread` to read the data back into memory.
- The data doesn't need to be in array form.

- A call of `fwrite` that writes a structure variable `s` to a file:

```
fwrite(&s, sizeof(s), 1, fp);
```

String I/O

- The next functions can read and write data using a string as though it were a stream.

- `sprintf` and `snprintf` write characters into a string.
- `sscanf` reads characters from a string.

Output Functions

- The `sprintf` function writes output into a character array (pointed to by its first argument) instead of a stream.
- A call that writes `"9/20/2010"` into `date`:

```
sprintf(date, "%d/%d/%d", 9, 20, 2010);
```

- `sprintf` adds a null character at the end of the string.
- It returns the number of characters stored (not counting the null character).

Output Functions

`sprintf`

- can be used to format data, with the result saved in a string until it's time to produce output.
- it is also convenient for converting numbers to character form.

Output Functions

- The `snprintf` function (new in C99) is the same as `sprintf`, except for an additional second parameter named `n`.

- No more than `n` – 1 characters will be written to the string, not counting the terminating null character, which is always written unless `n` is zero.
- Example:

```
snprintf(name, 13, "%s, %s", "Einstein", "Albert");
```

The string `"Einstein, Al"` is written into `name`.

Output Functions

- `snprintf` returns the number of characters that would have been written (not including the null character) had there been no length restriction.
- If an encoding error occurs, `snprintf` returns a negative number.
- To see if `snprintf` had room to write all the requested characters, we can test whether its return value was nonnegative and less than `n`.

Input Functions

- The `sscanf` function is similar to `scanf` and `fscanf`.

- `sscanf` reads from a string (pointed to by its first argument) instead of reading from a stream.
- `sscanf`'s second argument is a format string identical to that used by `scanf` and `fscanf`.

Input Functions

- `sscanf` is handy for extracting data from a string that was read by another input function.

- An example that uses `fgets` to obtain a line of input, then passes the line to `sscanf` for further processing:

```
fgets(str, sizeof(str), stdin);
  /* reads a line of input */
sscanf(str, "%d%d", &i, &j);
  /* extracts two integers */
```