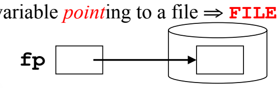
Input/Output

Based on slides from K. N. King and Dianna Xu

Bryn Mawr College
CS246 Programming Paradigm

Streams

- In C, the term stream means any source of input or any destination for output.
- Accessing a stream is done through a file pointer, which has type `FILE *`.
  - A variable *pointing* to a file ⇒ **`FILE *fp;`**



  - The `FILE` type is declared in `<stdio.h>`.
  - Certain streams are represented by file pointers with standard names – **`stdin`**, **`stdout`** and **`stderr`**

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.

Standard Streams and Redirection

- The I/O functions discussed in previous chapters obtain input from `stdin` and send output to `stdout`.
- Unix allows changing of default meanings through redirection**.**
- Input redirection forces a program to obtain its input from a file instead of from the keyboard:

```
demo <in.dat
```

- Output redirection is similar:

```
demo >out.dat
```

All data written to `stdout` will now go into the `out.dat` file instead of appearing on the screen.

Standard Streams and Redirection

- Input redirection and output redirection can be combined:

```
demo <in.dat >out.dat
demo < in.dat > out.dat
demo >out.dat <in.dat
```

- Output redirection: everything written to `stdout` is put into a file.
- Writing error messages to `stderr` instead of `stdout` guarantees that they will appear on the screen even when `stdout` has been redirected.

Text Files vs Binary Files

- `<stdio.h>` supports two kinds of files:
  - Text file: a sequence of bytes that represent characters, allowing humans to examine or edit the file.
    - E.g., the source code for a C program.

text	00000011	0000010	00000111	00000110	00000111
	'3'	'2'	'7'	'6'	'7'

  - Binary file: bytes don't necessarily represent characters.
    - Groups of bytes might represent other types of data, such as integers and floating-point numbers.
    - E.g., an executable C program.

binary	01111111	11111111

Text Files vs Binary Files

- Text files have two characteristics that binary files don't possess.
- 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

Text Files vs 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.
- In this lecture we cover text file I/O.

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.
  - 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.
- Returns the null pointer NULL (zero) on error, i.e. trying to read a file that doesn't exist.

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:

```
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
- 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)
"a+"	Open for reading and writing (append if file exists)

Modes

- Special rules apply when a file is opened for both reading and writing.
  - Can't switch from reading to writing without first calling a file-positioning function unless the reading operation encountered the end of the file.
  - Can't switch from writing to reading without either calling `fflush` or calling a file-positioning function.

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` or `freopen`.
- `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;
}
```

Closing 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) …
```

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;
}
```

File Buffering

- It takes time to transfer the buffer contents to or from disk, but one large "block move" is much faster than many tiny byte moves.
- A call that flushes the buffer for the file associated with `fp`:

```
fflush(fp);   /* flushes buffer for fp */
```

- A call that flushes *all* output streams:

```
fflush(NULL);  /* flushes all buffers */
```

- `fflush` returns zero if it's successful and `EOF` if an error occurs.

Formatted I/O

- Reading – returns number of matches or EOF

  `int fscanf(FILE *fp, "...",` *variableList*`);`
- Writing – returns number of chars written

  `int fprintf(FILE *fp, "...",` *variableList*`);`
- **`scanf`** is equivalent to **`fscanf`** with **`stdin`**
- **`printf`** to **`fprintf`** with **`stdout`**

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.

The …printf Functions

- `fprintf` works with any output stream.
- One of its most common uses is to write error messages to `stderr`:

```
fprintf(stderr, "Error: data file can't be opened.\n");
```

- Writing a message to `stderr` guarantees that it will appear on the screen even if the user redirects `stdout`.

Examples of …printf Conversion Specifications

- Examples showing the effect of flags on the `%d` conversion:

Conversion Specification	*Result of Applying Conversion to 123*	*Result of Applying Conversion to –123*
`%8d`	`•••••123`	`••••-123`
`%-8d`	`123•••••`	`-123••••`
`%+8d`	`••••+123`	`••••-123`
`% 8d`	`•••••123`	`••••-123`
`%08d`	`00000123`	`-0000123`
`%-+8d`	`+123••••`	`-123••••`
`%- 8d`	`•123••••`	`-123••••`
`%+08d`	`+0000123`	`-0000123`
`% 08d`	`•0000123`	`-0000123`

Examples of …printf Conversion Specifications

- Examples showing the effect of the minimum field width and precision on the `%s` conversion:

Conversion Specification	*Result of Applying Conversion to* **`"bogus"`**	*Result of Applying Conversion to* **`"buzzword"`**
`%6s`	`•bogus`	`buzzword`
`%-6s`	`bogus•`	`buzzword`
`%.4s`	`bogu`	`buzz`
`%6.4s`	`••bogu`	`••buzz`
`%-6.4s`	`bogu••`	`buzz••`

Examples of …printf Conversion Specifications

- The * character allows us to specify minimum field width and/or precision as argument(s) after the format string.
- A major advantage of * is that it allows us to use a macro to specify the width or precision:

```
printf("%*d", WIDTH, i);
```

- The width or precision can even be computed during program execution:

```
printf("%*d", page_width / num_cols, i);
```

- Calls of printf that produce the same output:

```
printf("%6.4d", i);
printf("%*.4d", 6, i);
printf("%6.*d", 4, i);
printf("%*.*d", 6, 4, i);
```

Examples of …printf Conversion Specifications

- The %p conversion is used to print the value of a pointer:

```
printf("%p", (void *) ptr);
  /* displays value of ptr */
```

  - The pointer is likely to be shown as an octal or hexadecimal number.
- The %n conversion is used to find out how many characters have been printed so far by a call of printf.
  - After the following call, the value of len will be 3:

```
printf("%d%n\n", 123, &len);
```

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.

The …scanf Functions

- The …scanf functions return the number of data items that were read and assigned to objects.
- They return EOF if no more input characters could be read before any data items can be read.
- Loops that test scanf's return value are common.
- A loop that reads a series of integers one by one, stopping at the first sign of trouble:

```
while (scanf("%d", &i) == 1) {
  …
}
```

…scanf Format Strings

- The format string represents a pattern that a … scanf function attempts to match as it reads input.
  - If the input doesn't match the format string, the function returns.
  - The input character that didn't match is "pushed back" to be read in the future.

…scanf Format Strings

- The format string "ISBN %d-%d-%ld-%d" specifies that the input will consist of:
  - the letters ISBN
  - possibly some white-space characters
  - an integer
  - the - character
  - an integer (possibly preceded by white-space characters)
  - the - character
  - a long integer (possibly preceded by white-space characters)
  - the - character
  - an integer (possibly preceded by white-space characters)

`scanf` Examples

- Examples that combine conversion specifications, white-space characters, and non-white-space characters:

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%d%d", &i, &j);`	~~12•~~,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d,%d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged
`n = scanf("%d ,%d", &i, &j);`	~~12•,•34~~¤	n: 2 i: 12 j: 34
`n = scanf("%d, %d", &i, &j);`	~~12~~•,•34¤	n: 1 i: 12 j: unchanged

`scanf` Examples

- Examples showing the effect of assignment suppression and specifying a field width:

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%*d%d", &i);`	~~12•34~~¤	n: 1 i: 34
`n = scanf("%*s%s", str);`	~~My•Fair~~•Lady¤	n: 1 str: "Fair"
`n = scanf("%1d%2d%3d", &i, &j, &k);`	~~12345~~¤	n: 3 i: 1 j: 23 k: 45
`n = scanf("%2d%2s%2d", &i, str, &j);`	~~123456~~¤	n: 3 i: 12 str: "34" j: 56

…`scanf` Conversion Specifications

- `%[`*set*`]` matches any sequence of characters in *set* (the ***scanset***) , where *set* can be any set of characters.
- `%[^`*set*`]` matches any sequence of characters not in *set*.
- Examples:
  `%[abc]` matches any string containing only `a`, `b`, and `c`.
  `%[^abc]` matches any string that doesn't contain `a`, `b`, or `c`.

`scanf` *Call*	*Input*	*Variables*
`n = scanf("%[0123456789]", str);`	123abc¤	n: 1 str: "123"
`n = scanf("%[0123456789]", str);`	abc123¤	n: 0 str: unchanged
`n = scanf("%[^0123456789]", str);`	abc123¤	n: 1 str: "abc"

`fscanf` and `fprintf`

- Reading – returns number of matches or EOF
  `int fscanf(FILE *fp, "…",` *variableList***`);`**
- Writing – returns number of chars written
  `int fprintf(FILE *fp, "…",` *variableList***`);`**
- **`scanf`** is equivalent to **`fscanf`** with **`stdin`**
- **`printf`** to **`fprintf`** with **`stdout`**

Character I/O

- Reading – returns char read or EOF

```
int fgetc(FILE *fp)
int getc(FILE *fp) // macro
int getchar() <==> int fgetc(stdin)
```

- Writing – returns char written

```
int fputc(int c, FILE *fp)
int putc(int c, FILE *fp) // macro
int putchar(int c) <==> int fputc(…,
  stdin)
int ungetc(int c, FILE *fp)
```

Character I/O

- `getchar` reads a character from `stdin`:
  `ch = getchar();`
- `fgetc` and `getc` read a character from an arbitrary stream:
  `ch = fgetc(fp); ch = getc(fp);`
- All three functions treat 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`.

Character I/O

- One of the most common uses of `fgetc`, `getc`, and `getchar` is to read characters from a file.
- A typical `while` loop for that purpose:

```
while ((ch = getc(fp)) != EOF) {
  …
}
```

- Always store the return value in an `int` variable, not a `char` variable.
- Testing a `char` variable against `EOF` may give the wrong result.

Character I/O

- The `ungetc` function "pushes back" a character read from a stream and clears the stream's end-of-file indicator.
- A loop that reads a series of digits, stopping at the first nondigit:

```
while (isdigit(ch = getc(fp))) {
  …
}
ungetc(ch, fp);
  /* pushes back last character read */
```

Character I/O

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

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

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

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

- File copy by Char:

```
FILE *in, *out;
// open both src and dest files as
// in and out, respectively
while ((c = fgetc(in)) != EOF) {
  fputc(c, out);
}
```

Line I/O

- Reading – returns pointer to string read, NULL if end of file

`char* fgets(char *buf, int max, FILE *fp)`

- Strings are character arrays in C
- **`max`** indicates the maximum number of characters to be read.
- **`max`** should be **`1`** less than the length of **`buf`**!
- **`gets`** is equivalent to **`fgets(…, stdin)`**
- Writing – returns number of chars written

`int fputs(char *buf, FILE *fp)`

Example: File Copy by Line

```
int main() {
  char buf[BUFLEN], inFile[BUFLEN], outFile[BUFLEN];
  FILE *in, *out;
  printf("Enter source filename: ");
  fgets(inFile,BUFLEN-1,stdin);
  inFile[strlen(inFile)-1] = '\0';
  // get outFile as well from user

  in = fopen(inFile, "r");
  out = fopen(outFile, "w");
  if ((in == NULL) || (out == NULL)) {
    printf("*** File open error\n");
    return;
  }
  /* NULL returned at EOF */
  while (fgets(buf, BUFLEN-1, in) != NULL) {
    fputs(buf, out);
  }
  fclose(in);  fclose(out);
  return 0;
}
```

File Positioning

- Each file has an associated file position
- When a file is opened, the file position is set either at the beginning or the end

`SEEK_SET` – beginning of file
`SEEK_CUR` – current file position
`SEEK_END` – end of file

```
int fseek(FILE *fp, long offset, int
whence)
void rewind(FILE *fp)
  rewind(fp) <==> fseek(fp, 0L,
  SEEK_SET)
```

String I/O

- Read and write data using a string as though it were a stream.
- 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).

String I/O

- `sscanf` reads characters from a string.
- 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 */
```

- `sscanf` returns the number of data items successfully read and stored.
- `sscanf` returns `EOF` if it reaches the end of the string (marked by a null character) before finding the first item.

String I/O

- One advantage of using `sscanf` is that we can examine an input line as many times as needed.
- This makes it easier to recognize alternate input forms and to recover from errors.
- Consider the problem of reading a date that's written either in the form *month*/*day*/*year* or *month*-*day*-*year*:

```
if (sscanf(str, "%d /%d /%d", &month, &day, &year) == 3)
  printf("Month: %d, day: %d, year: %d\n", month, day,
   year);
else if (sscanf(str, "%d -%d -%d", &month, &day, &year) ==
   3)
  printf("Month: %d, day: %d, year: %d\n", month, day,
   year);
else
  printf("Date not in the proper form\n");
```