

Richard Litt

Scan QR code to add me as a friend.

Week 11 Lecture 1

XMUT-NWEN 241 - 2026 T2

Systems Programming

Richard Littauer

School of Engineering and Computer Science

Victoria University of Wellington

richard.littauer@vuw.ac.nz

Admin

- Assignment 3
  - Due date is 22 November.

Review



File Stream I/O

Introduction to File Input / Output

- I/O is the process of copying data between main memory and external devices, like terminals (keyboards), disk drives, networks, etc.

- In C, everything is abstracted as a file
  - Each file is simply a sequential stream of bytes
  - C imposes no structure on a file

- From the program's point of view, data input and data output are made possible through files

Accessing Files

- A file must first be opened properly before it can be accessed for reading or writing
- Opening a file establishes a “communication channel” between the program and the file



File Stream vs File Descriptor

- "Communication channel" can either be a file stream or file descriptor
- C provides functions for accessing files via file stream or file descriptor

	File Descriptor	File Stream
Content access	Primitive access: contents can be accessed as blocks of bytes	Rich access: contents can be formatted using format specifiers
Control operations	Allows setting of control parameters	Does not allow
Special I/O modes	Allows special access modes such as non-blocking	Does not allow
Buffering	None	Supports 3 modes of buffering

File Stream vs File Descriptor

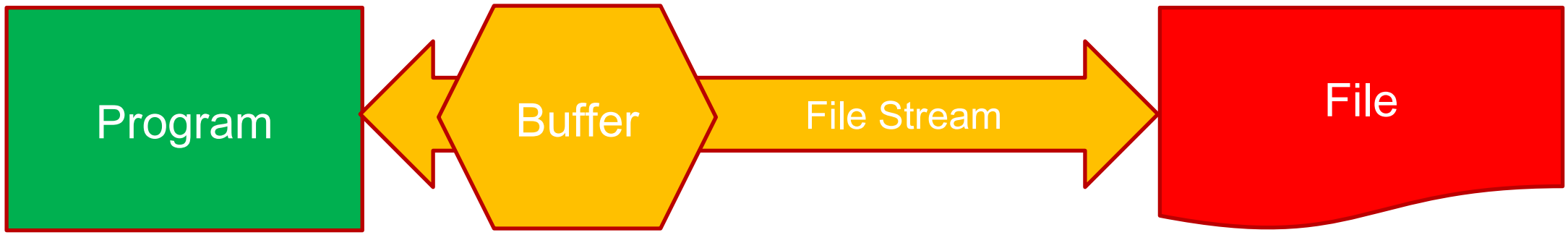
- File streams provide a higher-level interface, layered on top of the primitive file descriptor facilities

- For special files (e.g. I/O devices and sockets), file descriptor is the recommended approach

- For regular files (files on disk), file stream is the recommended approach

Stream Buffering

- One of the common pitfalls when dealing with file streams is buffering



- More problematic in interactive I/O streams
  - Data written by program to file does not appear immediately
  - Data read by program from file does not appear immediately

Illustration

```
char str[100];
scanf ("%s", str);
```

- If user types the string

```
The quick brown fox
```

- The string `str` will only be assigned "The"
- What happens to the rest?

Stream Buffering Modes

Mode	Description
Unbuffered	Characters written to or read from an unbuffered stream are transmitted individually to or from the file as soon as possible.
Line buffered	Characters written to a line buffered stream are transmitted to the file in blocks when a newline character is encountered.
Fully buffered	Characters written to or read from a fully buffered stream are transmitted to or from the file in blocks of arbitrary size.

- Newly opened streams are fully buffered by default, except streams connected to interactive devices which are line buffered
- C provides functions for changing stream buffering mode

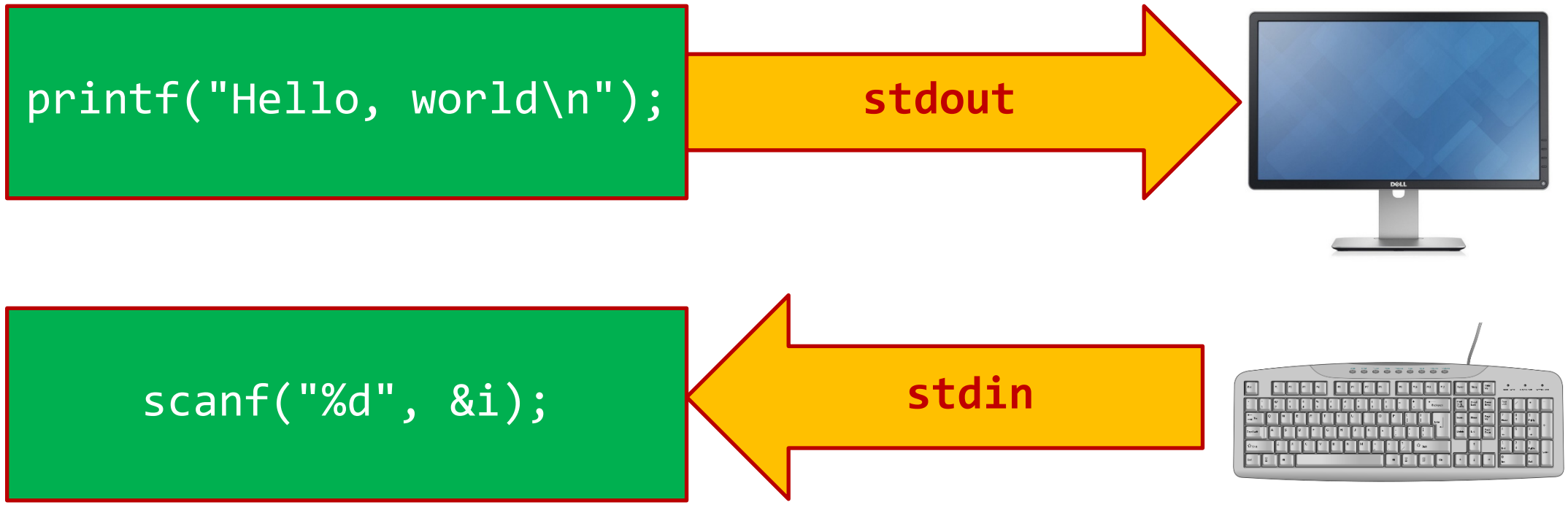
Built-in Streams (1)

- Every C program has access to 3 file streams: `stdin`, `stdout`, `stderr`

File	Description	Default	Buffering
`stdin`	Standard input stream	Connected to the keyboard	Line buffered
`stdout`	Standard output stream	Connected to the screen	Line buffered
`stderr`	Standard error stream	Connected to the screen	Unbuffered

Built-in Streams (2)

- You have already been using these streams without you knowing it!



File Stream (Stream for Short)

- The `<stdio.h>` header file provides types and functions for accessing streams
- **`FILE`** structure: a structure that holds information about a stream
- **`FILE`** facilitates stream I/O: C functions use **`FILE`** pointer to access files

Stream I/O Functions (1)

- `fopen` – open or create a file and associate a stream
- `fclose` – close a stream
- `fflush` – force to write all buffered data to file
- `fgetc` – read a single character from a stream
- `fputc` – write a single character to a stream

Stream I/O Functions (2)

- `fscanf` – read formatted input from stream
- `fprintf` – write formatted output to stream
- `fread` – read in binary mode from stream
- `fwrite` – write in binary mode to stream
- `fseek/rewind` – change position in stream
- `ftell` – determine position in stream

Opening a File

A file must be "opened" before it can be used

```
FILE *fp; // pointer to stream
          :::
fp = fopen (filename, mode);
```

"string" specifying the file name

"r" – open the file for reading only
"w" – open the file for writing only
"a" – open the file for appending data to it

returns a pointer to FILE (`FILE *`); used in all subsequent file operations.

Examples

- Open a file named mydata for reading:

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

- File is opened for reading only – file must exist
- File reading is positioned at the start of file

- Open or create a file named file.csv for writing:

```
FILE *fp;
fp = fopen ("file.csv", "w");
```

- Creates a new file for writing
- If file exists, contents (if any) are deleted
- File writing is positioned at the start of file

Examples

- Open or create a file named sample.txt for appending:

```
FILE *fp;
fp = fopen ("sample.txt", "a");
```

- Creates a new file for writing if does not exist
- File writing is positioned at the end of file

Did `fopen()` Succeed?

- If the file was not able to be opened, then the value returned by the `fopen()` is `NULL`
- Always check return value of `fopen()`

```
FILE *fp;
fp = fopen ("mydata", "r");
if (fp == NULL) {
   printf ("File open failed.\n");
   return 1;
}
```

Reasons for opening failure:

- File does not exist
- File is already open
- File cannot be created
- File cannot be accessed (insufficient permissions)

Closing a File

- After completing all operations on a file, it must be closed to ensure that all file data stored in the buffer are written to the file
- General format:

```
fclose (file_pointer);
```

```
FILE *fp; // pointer to data type FILE

        :::
fp = fopen (filename, mode);
        :::
fclose (fp); // close the file
```

Flushing Buffer Contents

- To force writing of buffer content to file without closing it, call the `fflush()` function
- General format:

```
fflush (file_pointer);
```

```
FILE *fp; // pointer to data type FILE
        :::
fp = fopen (filename, mode);
        :::
fflush (fp); // write buffer to file
        :::
```

Read/Write Operations on Files

- Simplest file input-output (I/O) function: `fgetc()` & `fputc()`

```
int ch;
FILE *fp;
   :::
ch = fgetc(fp);
```

- **`fgetc()`** reads one character from stream
- **`fgetc()`** returns an end-of-file marker `EOF`, when the end of the file has been reached

```
getchar() -> fgetc(stdin)
```

Read/Write Operations on Files

```
char ch;
FILE *fp;
   :::
fputc(ch, fp);
```

- `fputc()` is used to write a character to a stream

```
putchar(ch) -> fputc(ch,stdout)
```

Example with `fgetc()` and `fputc()`

```c
int main(void)
{
    FILE  *ifp, *ofp;
    int  c;

    ifp = fopen ("ifile.dat","r");
    ofp = fopen ("ofile.dat","w");

    while ((c = fgetc (ifp)) != EOF)
         fputc (toupper(c), ofp);
    fclose (ifp);
    fclose (ofp);
}
```

ifile.dat:

```
Hello nwen241!
```

ofile.dat:

```
HELLO NWEN241!
```

fgetc() vs getc()

- Both routines read a character from a stream
- `fgetc()` is implemented as a function while `getc()` is implemented as a function-like macro
- An argument in `getc()` should not be an expression with side effects
- Example: `fgetc(*p++)` works but `getc(*p++)` fails

`fputc()` vs `putc()`

- Both routines write a character to a FILE stream
- `fputc()` is implemented as a function while `putc()` is implemented as a function-like macro
- Same considerations as `fgetc()` and `getc()`

Recall: `scanf()`

- Reads user input from keyboard (`stdin` stream)

- Consider:

```
int a, b;
scanf("%d %d", &a, &b);
```

`scanf()` [and `printf()`] are variadic functions: the number of arguments they accept is not fixed

Format specifier expects 2 integers in decimal

2 numbers entered by user on keyboard will be stored here

fscanf()

- Same as `scanf()` except need stream (`FILE *`) as an argument
  - `scanf()` reads formatted input from `stdin` stream
  - `fscanf()` reads formatted input from specified stream
- Example:

```
int a, b;
FILE *fp;
fp = fopen ("datafile", "r");
fscanf(fp, "%d %d", &a, &b);
```

`fscanf()` would read values from the stream pointed by `fp` and assign those values to **a** and **b**

```
scanf("%d", &a) -> fscanf(stdin, "%d", &a)
```

Example (1)

- Consider:

```
int a, b;
FILE *fp;
fp = fopen ("datafile", "r");
fscanf(fp, "%d %d", &a, &b);
```

- Contents of `datafile`:

```
100 200
```

- What is the value assigned to a and b?

Example (1)

- Consider:

```
int a, b;
FILE *fp;
fp = fopen ("datafile", "r");
fscanf(fp, "%d %d", &a, &b);
```

- Contents of `datafile`:

```
100 200
```

- What is the value assigned to a and b?

a = 100, b = 200

Detecting End of File using **EOF**

- End-of-file indicator EOF informs the program when there are no more data (no more bytes) to be processed
- `fscanf()` returns EOF if end-of-file is reached, or errors were encountered when reading from stream
- Example:

```
int ret, var;
ret = fscanf (fp, "%d", &var) ;
if (ret == EOF) {
      printf ("End-of-file encountered.\n");
}
```

Detecting End of File using `feof()`

- Use the `feof(fp)` function which returns a non-zero value (true) or zero (false) condition
  - True if EOF was reached
  - False otherwise
  - It reports end of file only. Use `ferror(fp)` to check for read errors.

```
int var;
if (fscanf(fp, "%d", &var) != 1) {
   if (feof(fp))
      printf("End-of-file happened.\n");
   else if (ferror(fp))
      printf("Read error.\n");
   else
      printf("Input wasn't a number.\n");
}
```

Recall: `printf()`

- Writes to screen (`stdout` stream)

- Consider:

```
int a = 1, b = 2;
printf("%d %d", a, b);
```

Format specifier will write 2 integers in decimal

2 numbers to be written to screen

fprintf()

- Same as `printf()` except need stream (`FILE *`) as an argument
  - `printf()` writes formatted output to `stdout` stream
  - `fprintf()` writes formatted output to specified stream
- Example:

```
int a = 100, b = 200;
FILE *fp;
fp = fopen ("datafile", "w");
fprintf(fp, "%d %d", a, b);
```

`fprintf()` would write the values stored in a and b to the stream pointed to by `fp`

```
printf("%d", a) -> fprintf(stdout, "%d", a)
```

Example (1)

```
int a = 100, b = 200;
FILE *fp;
fp = fopen ("datafile", "w");
fprintf(fp, "%d %d", a, b);
```

- What will be the contents of `datafile` after running this code?

Example (1)

```
int a = 100, b = 200;
FILE *fp;
fp = fopen ("datafile", "w");
fprintf(fp, "%d %d", a, b);
```

- What will be the contents of `datafile` after running this code?

```
100 200
```

Handling Binary Files

- Same as dealing with text files except in the opening step

```
FILE *fp; // pointer to stream
        :::
fp = fopen (filename, mode);
```

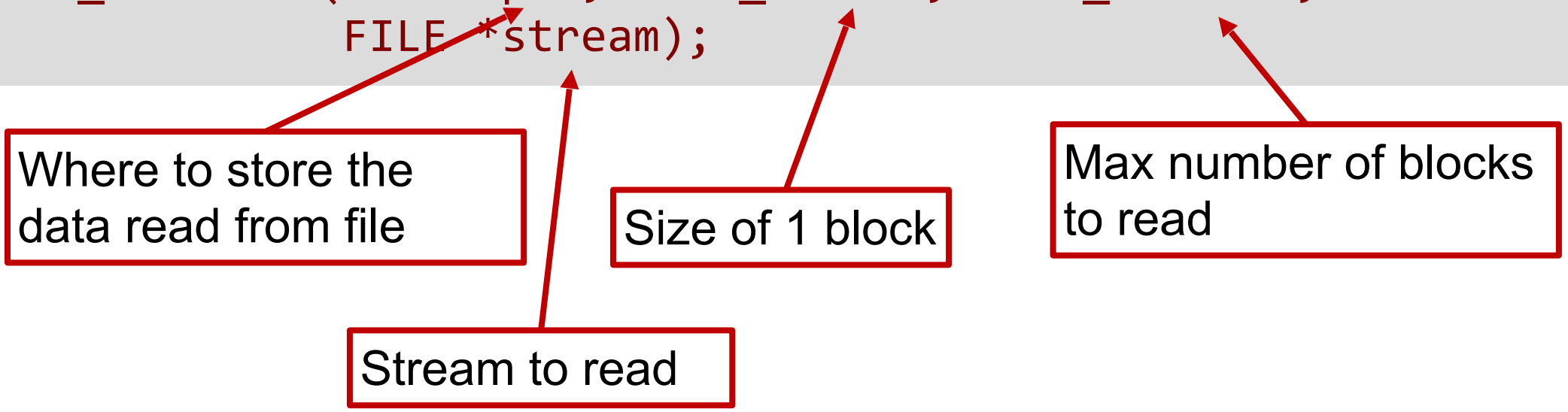


"rb" – open the file in binary mode for reading only
"wb" – open the file in binary mode for writing only
"ab" – open the file in binary mode for appending data to it

Reading Binary Files

- Read blocks of binary data from stream

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



- `fread()` returns the actual number of elements read

Example

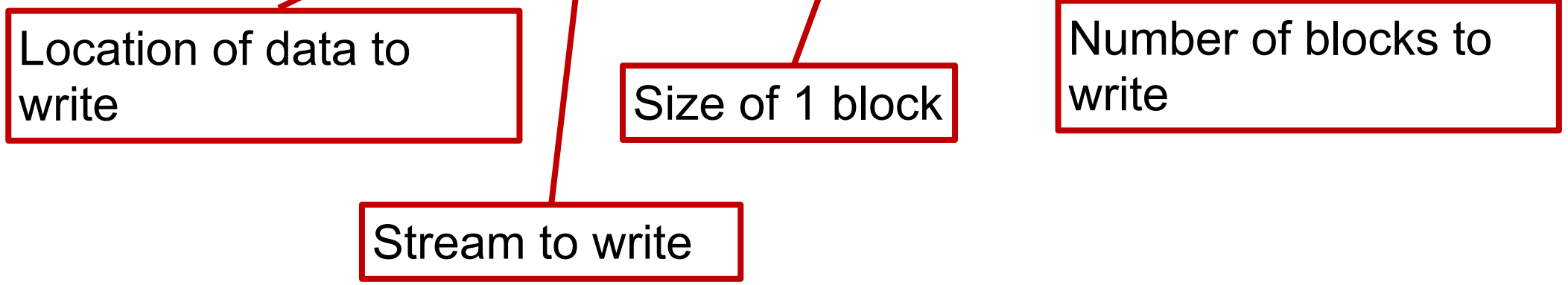
```
FILE *fp;
unsigned char buffer[10];

fp = fopen("file1.exe", "rb");
fread (buffer, sizeof(buffer), 1, fp);
```

- Will read the first 10 bytes of `file1.exe` and store them in `buffer`

Writing Binary Files

- Writes blocks of binary data to stream

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



- `fwrite()` returns the actual number of elements written

Example

```
FILE *fp;
int data[4] = {15, 31, 63, 127};

fp = fopen("datafile", "wb");
fwrite (data, sizeof(int), 4, fp);
```

- Will write the data array to datafile

Example

- In Linux, you can use hexdump to view contents of binary file

- `hexdump -d datafile` will display the contents of datafile in decimal

`datafile:`

```
00015   00000   00031   00000   00063   00000   00127   00000
```

15 31 63 127

Random Access

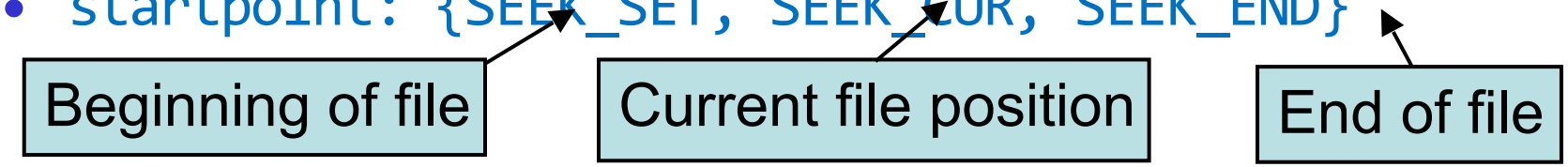
- After opening a file, read/write position is either at start or end of file
- To change position, use either `fseek()` or `rewind()`
- To know current position, use `ftell()`

fseek()

- fseek() allows repositioning within a file

```
int fseek(FILE *stream, long int offset, int startpoint);
```

- New position in the file is determined by:
  - offset – byte count (possibly negative) relative to the position specified by startpoint where
    - startpoint: {SEEK_SET, SEEK_CUR, SEEK_END}



ftell()

- `ftell()` returns the current file position:

```
long ftell(FILE *stream);
```

- This may be saved and later passed to `fseek()`:

```
long file_pos;
file_pos = ftell(fp);
…
fseek(fp, file_pos, SEEK_SET);
/* return to previous position */
```

`rewind()`

- Reposition reading/writing to start of file

- `rewind(fp)` is equivalent to:

```
fseek(fp, 0, SEEK_SET)
```

Review

- Assume this file: `ABCDEFGHIJKLMNOPQRSTUVWXYZ\n`
- `fp` is `alphabet.txt`, opened for reading. What do the three `fgetc` calls return?

```
fseek(fp, 10, SEEK_SET);
printf("%d\n", fgetc(fp));
fseek(fp, -3, SEEK_CUR);
printf("%d\n", fgetc(fp));
fseek(fp, 0, SEEK_END);
printf("%d\n", fgetc(fp));
```

Review

- What does this print, and why?

```
FILE *fp = fopen("data.txt", "r");

if (fprintf(fp, "x") < 0)
    printf("failed\n");
```

- Your user types `help me please I am stuck in a computer` and presses return. What does `line` contain afterwards, and where is the rest of the input?

```
char line[5];
fgets(line, sizeof(line), stdin);
```

Review

- The user types `3`, return, `hello`, return. What is in `c`?

```c
int n, c;

scanf("%d", &n);
c = getchar();
```

- Given `int x = -42;` How many bytes does `fprintf(fp, "%d", x)` write?

- How many does `fwrite(&x, sizeof x, 1, fp)` write?

Content

Command Line Arguments

Quick Tutorial on File Stream I/O and Command Line Arguments

Pointer Application 1: Accessing Array Elements

- An array name is a pointer to its first element
- `a[i]` is just shorthand for `*(a + i)`

Pointer Application 1: Accessing Array Elements

- ~~An array name is a pointer to its first element~~
- An array name converts to a pointer to its first element
- `a[i]` is just shorthand for `*(a + i)`

Decay

- Decay is an automatic conversion made by C
- In most expressions, an array gets converted to a pointer to its first element.

```
int a[10];
```

- What type is a?

Decay

- Decay is an automatic conversion made by C
- In most expressions, an array gets converted to a pointer to its first element.

```
int a[10];
```

- a has the type int[10]: ten ints in one block of 40 bytes
- When you write a + i, or pass a to a function, or assign int *p = a;
  - Then the compiler converts a into &a[0], which has type int *

Decay

- There are three exceptions:
  - sizeof a gives 40, the size of the whole array
  - &a gives a pointer to the whole array, type int (*)[10]. It has the same address as &a[0].
    - What happens if you add +1 to this?

Decay

- There are three exceptions:
  - sizeof a gives 40, the size of the whole array
  - &a gives a pointer to the whole array, type int (*)[10]. It has the same address as &a[0].
    - Adding 1 to it moves 40 bytes, not 4.
  - A string literal used to initialise an array, like char s[] = "hi";, copies the characters into s.

Decay

- What happens if you do this?

```
void f(int a[]) {
   sizeof a; // or
   sizeof *a;
}
```

- What about this?

```
i[a]
```

Decay

- Compare these: sizeof?

```
int a[3] = {1, 2, 3};
int *p = malloc(3 * sizeof(int));
```

Decay

- Compare these: sizeof?

```
int a[3] = {1, 2, 3};
int *p = malloc(3 * sizeof(int));
```

- a is 12 bytes: [1][2][3]
- p is 8 bytes, because it's just an address, pointing to 12 bytes somewhere on the heap

Decay

- When things decay:

```
a + 1               // converted
a[2]                // converted (it's *(a + 2))
f(a)                // converted
int *q = a;         // converted
printf("%p", a)     // converted
sizeof a            // not converted: 12
&a                  // not converted: pointer
```

Decay

- Why decay?
- C has no way to treat a whole array as one value.
  - Can’t copy an array with =
  - Can’t pass an array to a function
  - Can’t return an array by value
- So C hands around the address of the first element instead.

Command Line Arguments

Command Line Arguments

- Command line arguments are parameters supplied to a program when it is invoked

- Example:
  - When invoking the command **cd** to change directory, you have to specify the directory that you want to go to as an **argument**:

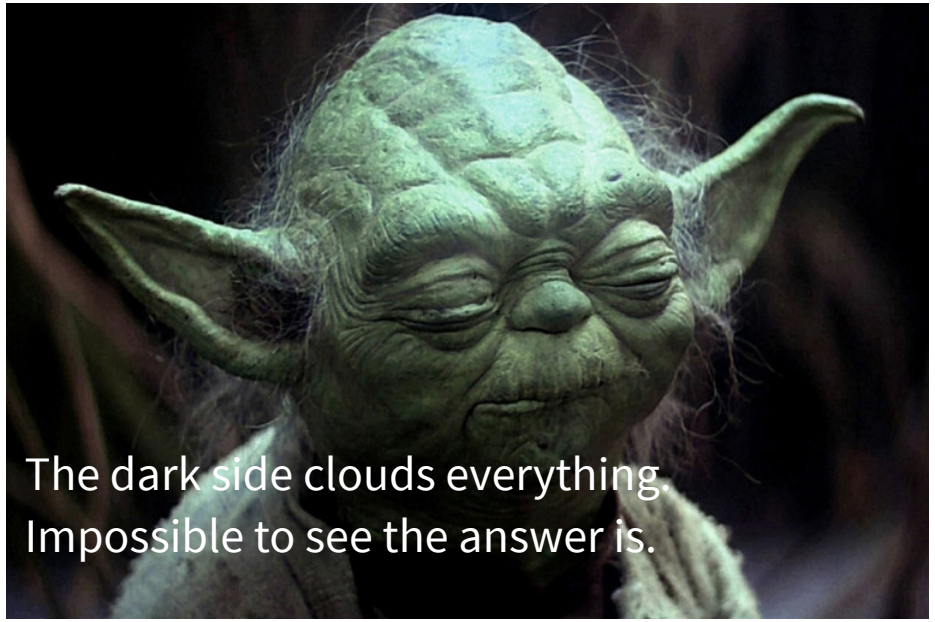
```
$ cd /home/yoda/padawan_grades
```

Command line argument

Command Line Arguments

- Command line arguments are parameters supplied to a program when it is invoked

- How do these parameters get into the program?



main() Function

- The main function can actually be implemented in two ways

```
int main(void)
{
    ...
}
```



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

Command Line Arguments

- Command line arguments are parameters supplied to a program when it is called

- How do these parameters get into the program?
  - Every C program has a `main()` function
  - `main()` can actually take 2 arguments, conventionally called `argc` and `argv`
  - Command line arguments are passed to the program through `argc` and `argv`

Passing Arguments to `main()`

- General format of command line arguments:

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

  - `argc`
    - Number of arguments (including program name)
  - `argv`
    - Array of strings
    - `argv[0]` → program name
    - `argv[1]` → first argument
    - …
    - `argv[argc-1]` → last argument

Passing Arguments to `main()`

- `char *argv[]` is the same as `char **argv`
- If we have `./prog first second`, then:
  - What is `argv[1]`?
  - What is `argv[1][0]`?
- `argv[argc]` will always equal `NULL`

Example

- Consider the C program `main_arg.c`:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  printf("%d arguments\n", argc);
  for(i = 0; i < argc; i++)
    printf("    %d: %s\n", i, argv[i]);
  return 0;
}
```

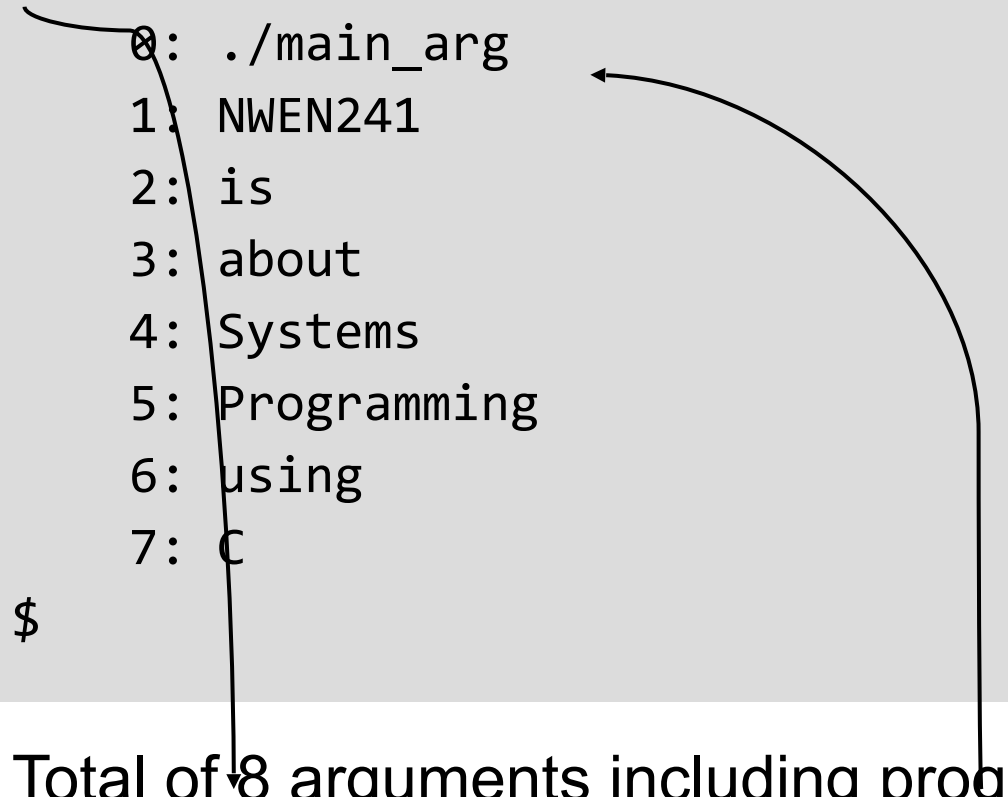
Example

- Compile and generate executable file `main_arg`

```
gcc main_arg.c -o main_arg
```

Program Output

```
$ ./main_arg NWEN241 is about Systems Programming using C
8 arguments
    0: ./main_arg
    1: NWEN241
    2: is
    3: about
    4: Systems
    5: Programming
    6: using
    7: C
$
```



Total of 8 arguments including program name itself.

Arguments are read in as strings.

Next Lecture

- Introduction to System Calls
- Process Management

Review

Q: You fwrite an int to a file, then open the file in a text editor. Why does it look like garbage?

Q: How many bytes does fprintf(fp, "%d", 7) write? And fprintf(fp, "%d", 7000000)?

Q: How many bytes does fwrite(&x, sizeof x, 1, fp) write when x is 7? And when x is 7000000?

Review

Q: fread(buffer, 10, 1, fp) reads all 10 bytes. Does it return 1 or 10?

Q: You ask fread for 100 ints, but the file only holds 30. What does it return?

Q: You open a file with "rb" and call fwrite on it. What happens?

Review

Q: Right after fopen, with "r", what does ftell(fp) return?

Q: A file is 27 bytes long. After fseek(fp, 0, SEEK_END), what does ftell(fp) return?

Q: You've read a file to the end and want to read it again from the top. What do you call?

Review

Q: The user types 42 and presses return. scanf("%d", &n) reads the number. What's still waiting in the input?

Q: char line[5]; fgets(line, sizeof(line), stdin); The user types hello and presses return. What's in line?

Review

Q: You run ./prog one two three. What is argc?

Q: Same command. What is argv[0]?

Q: Same command. Which argument is argv[argc - 1]?

Q: You run ./prog hello. What is argv[1][1]?

Review

Q: You run ./prog "hello world". What is argc?

Q: You run ./prog 5. Is argv[1] the number 5?

Q: You run ./prog with no arguments. What is argv[1]?

Q: In ls -l /tmp, what are argc and argv[2] for ls?