Week 6 Lecture 2

NWEN 241
Systems Programming

Jyoti Sahni

Jyoti.sahni@ecs.vuw.ac.nz

Content

Continuation of File Stream I/O

Command Line Arguments

Recap: 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

Recap: 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

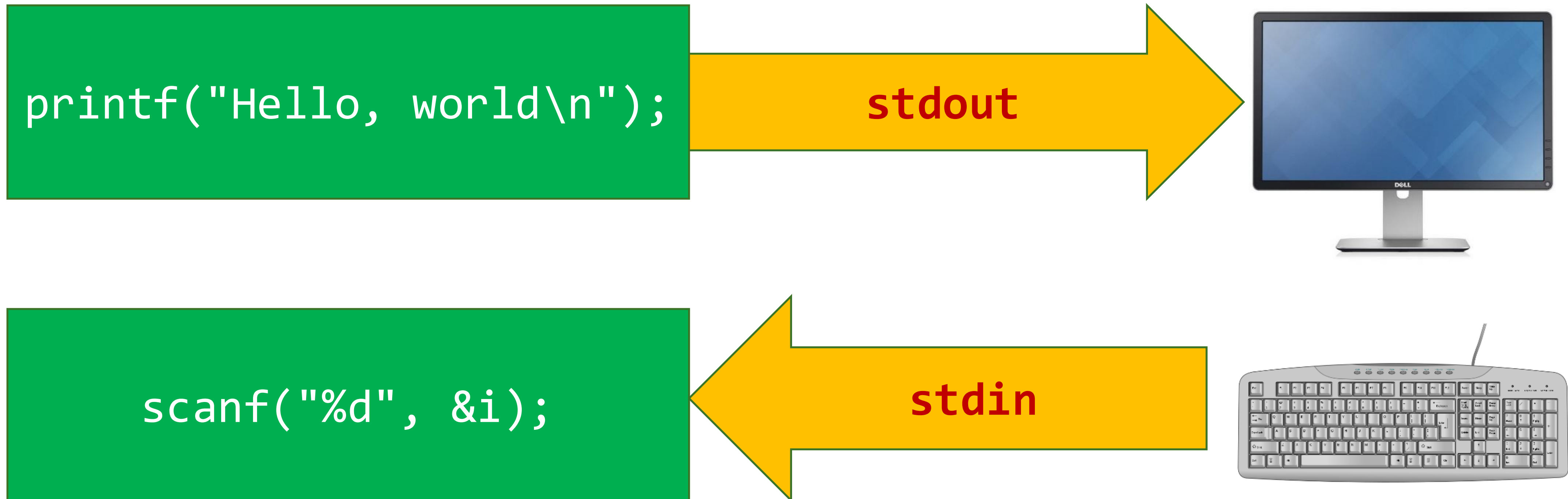
Builtin Streams (1)

- 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

Builtin 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 0;
}
```

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()` return 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(c, fp);
```

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

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

Example with fgetc() and fputc()

```
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**
- **Argument to** `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);
```

Format specifier expects 2 integers in decimal

2 numbers entered by user on keyboard will be stored here

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

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? a = 100, b = 200

Example (2)

- Consider:

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

- Contents of `datafile`:

```
100 200
```

200 is taken as a hexadecimal number

- What is the value assigned to a and b?

a = 100, **b = 512**

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()` function which returns a non-zero value (true) or zero (false) condition
  - *True* if EOF is reached, or errors were encountered during read operation
  - *False* otherwise
- Example:

```
int var;
fscanf (fp, "%d", &var) ;
if (feof(fp)) {
     printf ("End-of-file encountered.\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 prinf() 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?

```
100 200
```

Example (2)

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

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

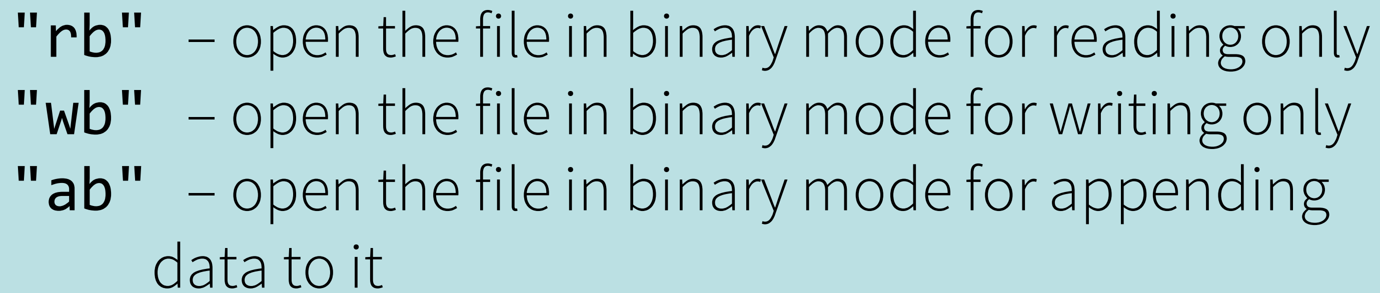
```
100 c8
```

c8 is the hexadecimal representation of 200

Handling Binary Files

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

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



Reading Binary Files

- Read blocks of binary data from stream

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

Size of 1 block

Max number of blocks to read

Stream to read

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

Location of data to write

Stream to write

Size of 1 block

Number of blocks to write

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

Example

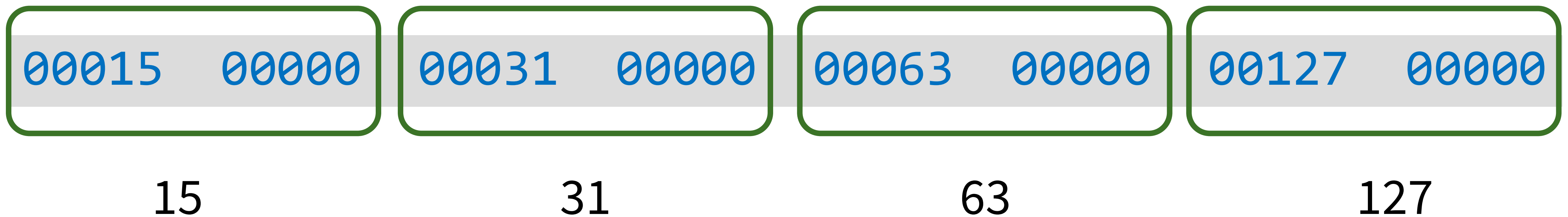
```
FILE *fp;
int data[4] = {15, 31, 63, 127};

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:`



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 -ve) 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)
```

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 may 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 invoked

- 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

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

- Friday tutorial
- Alvin Valera will deliver the second half of the course