NWEN 241
Getting closer to the system

Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

Victoria
UNIVERSITY OF WELLINGTON
*Te Whare Wānanga
o te Ūpoko o te Ika a Māui*

CAPITAL CITY UNIVERSITY

Command Line Arguments

- Command line arguments are parameters supplied to a program, when the program is invoked.
- How do these parameters get into the program?
- Every C program has a **`main`** function.
- **`main`** can 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 char*s (that is, an array of ‘c’ strings)
- `argv[0]` → program name
- `argv[1]` → first argument
- …
- `argv[argc-1]` → last argument

Example

```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 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.

Input / Output & `stdio.h`

- In general, **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 a ***file***; each file is simply a sequential stream of bytes; C imposes no structure on a file.
- Defined in **`stdio.h`** is the **`struct FILE`** that comprises a file descriptor and a file control block.
- A ***file*** must first be opened properly before it can be accessed for reading or writing. When a file is opened, a ***stream*** is associated with the file. Pointer to (i.e. address of) the "file" is returned.

Input / Output & `stdio.h`

- Every UNIX/Linux process begins with three open files corresponding to the standard input, output and error streams, macros defined in **`stdio.h`**:

Macro	Name	Physical relationship
`stdin`	standard input *file*	connected to keyboard
`stdout`	standard output *file*	connected to screen
`stderr`	standard error *file*	connected to screen
`EOF`	end-of-file	special –ve integer constant

- Also defined in **`stdio.h`** are three variable types (including **`FILE`**), several macros (including above) and various functions for performing input / output, e.g. **`printf()`**, **`scanf()`**, **`getchar()`** , **`gets()`**, **`putchar()`**, **`puts()`**, etc.

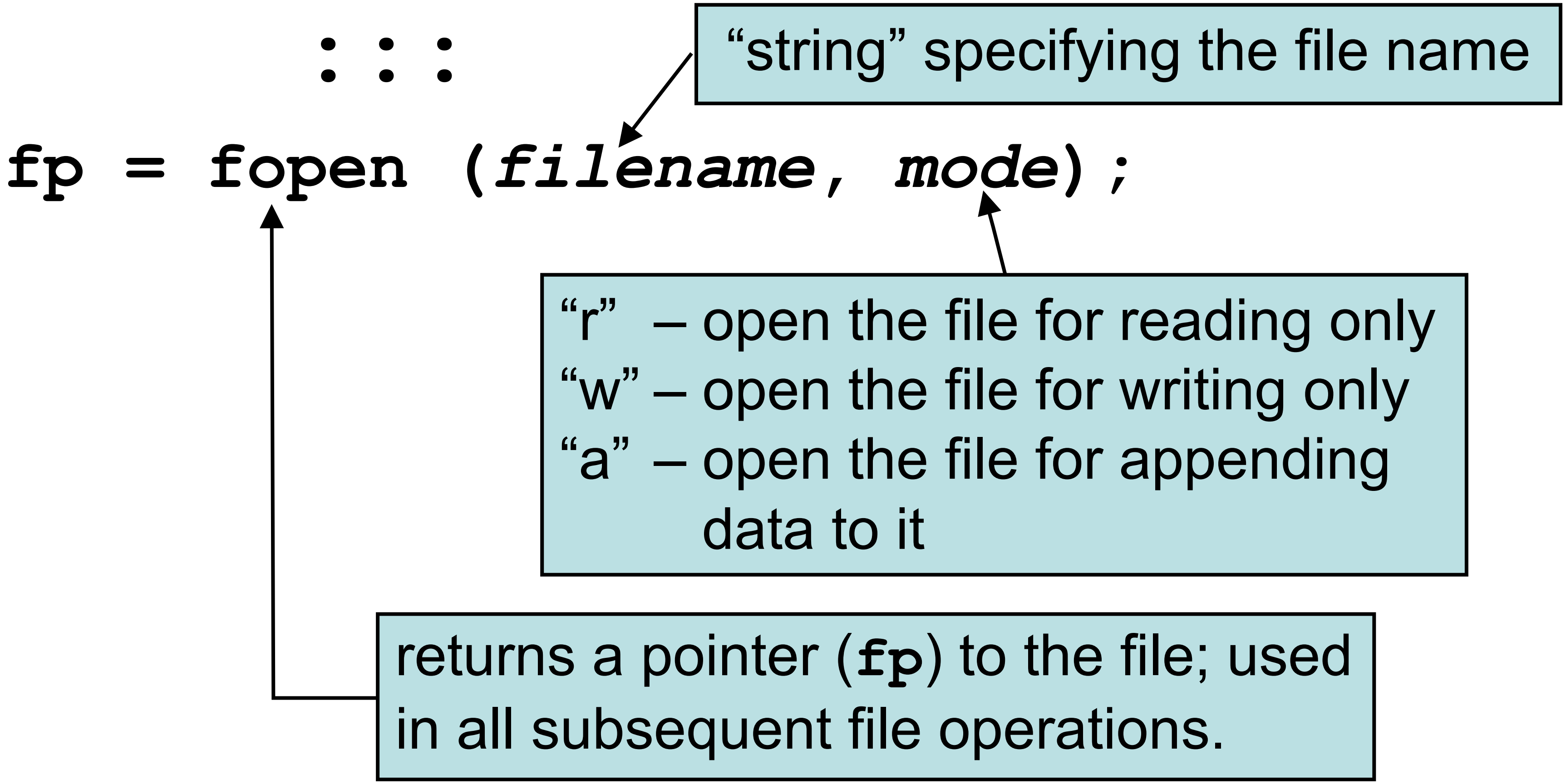
File operations

- Creating a new file
- Opening an existing file
- Writing data to a file
- Reading data from a file
- Closing a file
- Random access operations

Declaring `FILE` pointer and Opening file

A file must be “opened” before it can be used.

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



```
          : : :
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 (`fp`) to the file; used in all subsequent file operations.

Did the `fopen(...)` command succeed?

If the file was not able to be opened, then the value returned by the **`fopen`** routine is **`NULL`**.

For example, if the file **`mydata`** does not exist, then:

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

Closing a file

After completing all operations on a file, it must be closed to ensure that **all** file data stored in memory buffers 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
```

Read/Write Operations on Files

Simplest file input-output (I/O) function: **`getc`** & **`putc`**

```
char ch;
FILE *fp;
    :::
ch = getc(fp);
```

`getc` will return an end-of-file marker **`EOF`**, when the end of the file has been reached.

`putc` is used to write a character to a file.

```
char ch;
FILE *fp;
    :::
putc(c, fp);
```

Example

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

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

    while ((c = getc (ifp)) != EOF)
         putc (toupper(c), ofp);
    fclose (ifp);
    fclose (ofp);
}
```

`fgetc()` and `fputc()`

`fgetc()` vs **`getc()`**

- **`fgetc`** is a subroutine that performs the same function as the **`getc`** **macro**; **`fgetc`** is NOT a macro.
- **`fgetc`** subroutine runs more slowly than **`getc`** but takes less disk space.
- Benefit:

  `fgetc(*p++)` works but **`getc(*p++)`** fails

`fputc()` vs **`putc()`**

- **`fputc`** is a subroutine while **`putc`** is a macro;
- same considerations for **`fputc`** as **`fgetc`**.

`fscanf()`

- Same as `scanf` except need to file pointer as an argument.
- Example:

```
int a, b;
FILE *fptr1;
fptr1 = fopen ("datafile", "r");
fscanf( fptr1, "%d%d", &a, &b);
```

- `fscanf` would read values from the file "pointed" to by `fptr1` and assign those values to `a` and `b`.

End of File using **EOF**

- The end-of-file indicator **EOF** informs the program when there are no more data (no more bytes) to be processed.
- Check the value returned by the **fscanf** function:

```
int istatus, var;

istatus = fscanf (fptr1, "%d", &var) ;

if ( istatus == EOF )
{
    printf ("End-of-file encountered.\n") ;
}
```

End of File using `feof()`

- Use the **`feof`** function which returns a **true** or **false** condition:

```
fscanf (fptr1, "%d", &var) ;
if ( feof (fptr1) )
{
    printf ("End-of-file encountered.\n");
}
```

fprinf()

- Same as **printf** except need to file pointer as an argument.
- Example:

```
int a=5, b=20;
FILE *fptr2;
fptr1 = fopen ("results", "w");
fprintf (fptr2, "%d %d\n", a, b);
```

- **fprintf** functions would write the values stored in **a** and **b** to the file "pointed" to by **fptr2**.

Example using fscanf() & fprintf()

```
#include <stdio.h>
int main ( )
{
   FILE *outfile, *infile ;
   int b = 5, f ;
   float a = 13.72, c = 6.68, e, g ;
   outfile = fopen ("testdata", "w") ;
   fprintf (outfile, "%6.2f%2d%5.2f", a, b, c) ;
   fclose (outfile) ;
   infile = fopen ("testdata", "r") ;
   fscanf (infile,"%f %d %f", &e, &f, &g) ;
   printf ("%6.2f,%2d,%5.2f\n", e, f, g) ;
   fclose (outfile) ;
}
```

Handling binary files

- Same as dealing with text files except in the opening step.
- Need to open the file as a binary file using the binary mode identifier, e.g.
  - "rb" **r** for read and **b** for binary
  - "wb" **w** for write and **b** for binary
  - ”ab" **a** for append and **b** for binary
- Example:

```
FILE *ptr;
ptr = fopen ("file1.exe","rb");
```

Reading binary files

- `fread` reads a block of binary data, up to `nmemb` elements of size `size` from `stream`, storing them at the address specified by `ptr`.

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

- `fread` returns the actual number of elements read.
- Example:

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

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

Writing binary files

- **fwrite** writes a block of binary data comprising **nmemb** elements of size **size** from **ptr** to **stream**.

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

- **fwrite** returns the number of elements written.
- Example:

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

write_ptr = fopen("file2.exe","wb");
fwrite (buffer,sizeof(buffer),1,write_ptr);
```

Random Access (1)

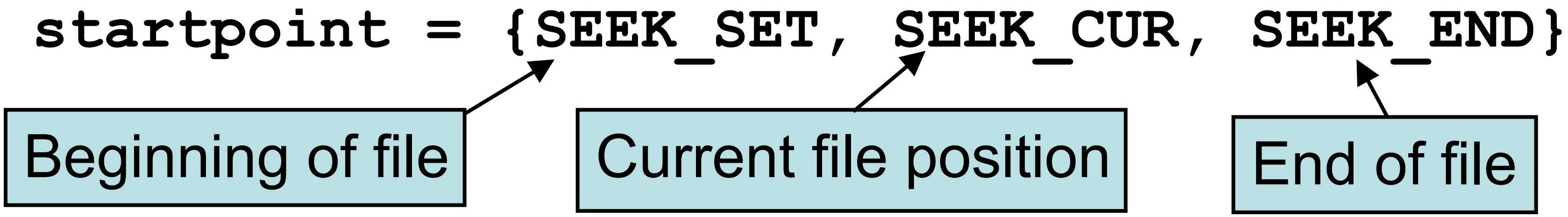
Most often used with binary files using **`fseek`**, **`ftell`** and **`rewind`**.

`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}
```

Beginning of file

Current file position

End of file

Random Access (2)

ftell returns the current file position:

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

This may be saved and later passed to fseek:

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

rewind(fp) is equivalent to:

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