NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Command line arguments
- File IO - C
- File IO – C++

C and C++

C

C++

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 they get into the program?

- The main function can 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 they get into the program?

- The *Command line arguments* get into the program because

  - 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

Command line arguments

General format of command line arguments:

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

- argc
  - Number of arguments (including program name)
- argv
  - Array of strings (character arrays)
  - argv[0] program name
  - argv[1] first argument
  - … –

  argv[argc-1] last argument

Example

```
#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 abstracted as a file
  - Each file is simply a sequential stream of bytes
  - C imposes no structure on a file

Input / Output & stdio.h

- Defined in stdio.h is the structure 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 FILE is returned

Input / Output & stdio.h

- In C stdio.h provides functions with input and output capability
- From the program's point of view, data input and data output are made possible through files
- Every C program has access to 3 such files: **stdin, stdout, stderr**

stdin	stdin	stdin
stdin	Standard input file	Connected to the keyboard
stdout	Standard output file	Connected to the screen
stderr	Standard error file	Connected to the screen

- 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

C

- 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

C

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 a text file for reading only
“w” – open a text file for writing only
“a” – open a text file for appending data to it

returns a pointer (**fp**) to the file; used in all subsequent file operations.

Opening modes

C

In addition to the modes "r", "w", and "a", a file may also be “opened” in *read-write* modes

- "r+":
  - Open in read-write mode
  - Return NULL if file does not exist
- "w+":
  - Open in read-write mode
  - Create file if it does not exist, or empty file if it exists
- "a+":
  - Open in read-write mode
  - Create file if it does not exist, writing will start at the end of file

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

C

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 ; // pointer to data type FILE
fptr = fopen ("mydata", "r") ;
if (fptr == NULL)
{
   printf ("File open failed.\n");
}
```

Closing a file

C

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

C

The 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.

getchar() is equivalent to getc(stdin) – getchar() can only get input from stdin

Read/Write Operations on Files

C

putc is used to write a character to a file

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

putchar(c) is equivalent to putc(c,stdout) – putchar can only write to stdout

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);
    Return 0;
}
```

fgetc() and fputc()

fgetc() vs getc()

- Both routines read a character from a FILE stream
- fgetc is implemented as a function while getc is implemented as a **macro**
- fgetc function runs more slowly than getc but takes less disk space.
- Benefit: fgetc(*p++) works but getc(*p++) fails

fputc() vs putc()

- Both routines write a character to a FILE stream
- fputc is a function while putc is a macro
- same considerations for fputc as fgetc

Example

- It could lead to strange behavior in some (not very useful) cases, e.g.:

- FILE *my_files[10] = {...}, *f=&my_files[0];
- for (i=0; i<10; i++) {
- int c = getc(f++); // Parameter to getc has side effects!
- }

- If getc evaluates f++ more than once, it will advance f more than once per iteration. In comparison, fgetc is safe in such situations.

fscanf()

C

Same as scanf except needs a file pointer as an argument.

Example:

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

fscanf reads values from the file "pointed" to by fptr1 and assign those values to a and b.

End of File using EOF

C

- 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

Example:

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

End of File using feof()

C

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

Example:

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

fprintf()

C

- Same as printf except need to use file pointer as an argument:

Example:

```
int a=5, b=20;
FILE *fptr2;
fptr2 = 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()

C

```
#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

C

- 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
- Likewise, binary files can be opened in read-write mode using "rb+", "wb+", and "ab+"
- Example:

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

Reading binary files

C

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

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

C

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

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

C

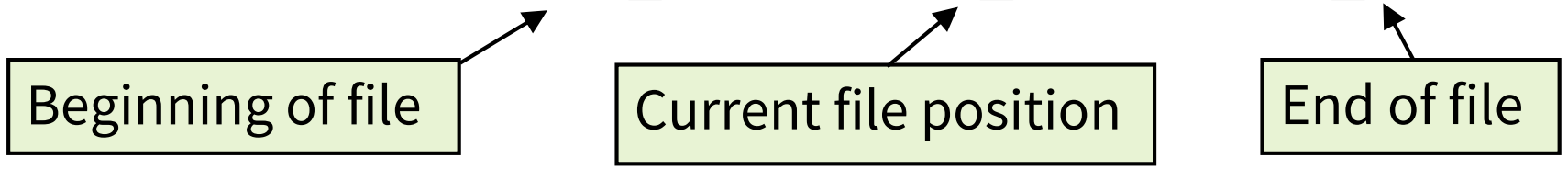
- Most often used with binary files using **`fseek`**, **`ftell`** and **`rewind`**
- **`fseek`** allows repositioning within a file.

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

The 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}`



Random Access (2)

C

- **`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).
```

```c
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  FILE *fp;
  char data[60];
  fp = fopen ("testra.c","w");
  fputs("NWEN241 is a systems programming
  course using C++ and C", fp);
  fclose(fp);
  fp = fopen ("testra.c","r");
  fgets (data, 60, fp );
  printf("Before fseek - %s \n", data);
  fseek(fp, 21, SEEK_SET);
  fgets ( data, 60, fp );
  printf("After SEEK_SET to 21 - %s\n",data);
  fseek(fp, -10, SEEK_CUR);
  fgets ( data, 60, fp );
  printf("After SEEK_CUR to -10 - %s\n", data);
  fseek(fp, -7, SEEK_END);
  fgets ( data, 60, fp );
  printf("After SEEK_END to -7 - %s\n", data);
  fseek(fp, 0, SEEK_SET);
  fgets ( data, 60, fp );
  printf("After SEEK_SET to 0 - %s\n", data);
// Can use rewind(fp); also
  fclose(fp);
  return 0;
}
```

```
Before fseek - NWEN241 is a systems programming course using C++ and C
After SEEK_SET to 21 - programming course using C++ and C
After SEEK_CUR to -10 -  C++ and C
After SEEK_END to -7 - + and C
After SEEK_SET to 0 - NWEN241 is a systems programming course using C++ and C
```