Part 6: UNIX File I/O

Contents

[DOCUMENT NOT FINALIZED YET]

C++ : feature available in C++ but not in C

File Input/Output

Storing and retrieving data sets on a persistent storage medium such as harddrives, tapes, or USB memory sticks is essential to computing

Operating systems provide the interface between applications and storage hardware

C library `stdio` provides functions for creating, removing, and renaming files, as well as writing and reading data (`man stdio`)

C++ provides stream classes (see iostream)

In Unix all input and output (I/O) is done by reading or writing to files

Devices attached to the computer are controlled by files residing in directory `/dev` with special I/O semantics which is implemented in so called device drivers

E.g., reading key presses is accomplished by reading from file `/dev/pts/0`

File Operations

Before using files they need to be opened

- The operating system checks access permissions
- If OK, a file identifier is returned which is then used to access the file in subsequent operations

When done, files have to be closed. This operation ensures that all data is actually written to the storage medium. This seems to be surprising at first, but because storage media are much slower than main memory, it is beneficial to write to and read from fast memory buffers and postpone actual storage operations

For instance, consider writing millions of numbers to a file residing on a harddrive which stores data on spinning magnetic disks. Instead of positioning the read/write head for each number individually — which can take milliseconds — one could store thousands of numbers in a buffer and when it gets full, write the entire buffer to disk. This is much faster because head positioning is slow but once found, writing consecutive data is fast

Unbuffered vs. Buffered I/O in C

We distinguish unbuffered I/O functions such as

`open, read, write, close`

that access files using so-called file descriptors (fds) (small non-negative integers) from buffered I/O functions such as

`fopen, fread, fclose, fwrite`

that access files using FILE structs which contain a file descriptor and buffer information

Consult man pages to learn how these functions work

E.g., man 2 open and man 3 fopen

In what follows, some examples will be given and the most important I/O C library functions will be discussed

Example: Write Binary Data (Unbuffered)

```
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <unistd.h>

int main()
{
  const int N = 10'000;
  int *a = new int[N];
  for (int i=0; i < N; ++i) { a[i] = i; }
  // create file "data", truncate it, open it for write
  // operation, set user permissions to rw
  int fd = open("data", O_CREAT | O_TRUNC | O_WRONLY,
                S_IRUSR | S_IWUSR);
  if (fd < 0) {
    perror("encountered open error"); exit(10);
  }
  int length = N*sizeof(a[0]);
  if (write(fd, a, length) != length) {
    perror("encountered write error"); exit(10);
  }
  if (close(fd) < 0) {
    perror("encountered close error"); exit(10);
  }
  delete [] a;
}
// Check file content with od -t x1 data | less
// Are lab machines big or little endian?
```

Example: Read Binary Data (Unbuffered)

```
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <unistd.h>

int main()
{
  const int N = 10'000;
  int *a = new int[N];
  int fd = open("data", O_RDONLY);
  if (fd < 0) {
    perror("encountered open error"); exit(10);
  }
  int length = N*sizeof(a[0]);
  if (read(fd, a, length) != length) {
    perror("encountered read error"); exit(10);
  }
  if (close(fd) < 0) {
    perror("encountered close error"); exit(10);
  }
  for (int i=0; i < N; ++i) {
    printf("%d\n", a[i]);
  }
  delete [] a;
  return 0;
}
// How to write/read binary data so that it can be used
// irrespective of endianness?
```

Example: Write Text Data (Buffered)

```
#include <cstdio>
#include <cstdlib>

int main()
{
  int N = 10'000;
  FILE *fp = fopen("numbers", "w");

  if (!fp) {
    perror("encountered fopen error"); exit(10);
  }

  for (int i=0; i < N; ++i) {
    if (fprintf(fp, "%d ", i) < 0) {
      perror("encountered fprintf error"); exit(10);
    }
  }

  if (fclose(fp)) {
    perror("encountered fclose error"); exit(10);
  }
  return 0;
}
```

Example: Read Text Data (Buffered)

```
#include <cstdio>
#include <cstdlib>

int main()
{
  int N = 10'000;
  FILE *fp = fopen("numbers", "r");

  if (!fp) {
    perror("encountered fopen error"); exit(10);
  }

  for (int i=0; i < N; ++i) {
    int x;
    if (fscanf(fp, "%d", &x) != 1) {
      printf("encountered read error\n"); exit(10);
    }
    printf("%d\n", x);
  }

  if (fclose(fp)) {
    perror("encountered fclose error"); exit(10);
  }
  return 0;
}
```

Standard Input, Output, Error

Special file descriptors `0,1,2` refer to standard input, standard output, and standard error, resp.

The command shell connects fds 0,1,2 with the console (input: keyboard, output: text window). All other files have to be opened

When redirecting program I/O to and from files using > , >>, and <, fds 0,1,2 are associated with files. E.g.,

```
./prog < infile > outfile
```

associates fd 0 (standard input) with file `infile` and and fd 1 (standard output) with file `outfile`

Normally, file descriptor 2 remains attached to the console to display error messages

Output to standard error can also be redirected. Syntax is shell-dependent. E.g., bash: `./prog &> xxx`

This redirects both standard output and standard error to file `xxx`

FILE structs stdout, stdin, stderr

The C stdio library provides pointers to FILE structs for buffered access to the standard input, output, and error streams:

`stdin, stdout, stderr`

These are global objects that are initialized before `main` is executed

To write text to and read text from those streams, we can use the generalized versions of `printf` and `scanf`: `fprintf` and `fscanf` that take a FILE struct pointer as parameter:

```
#include <cstdio>

fprintf(stdout, "text sent to standard output");
// equivalent to printf("text sent to standard output");

fscanf(stdin, "%d", &x);
// equivalent to scanf("%d", &x);

fprintf(stderr, "text sent to standard error");
// will be printed to terminal, even if program output
// is redirected to file
```

Some Unbuffered I/O Functions

```
#include <fcntl.h>
#include <unistd.h>
int open(char *filename, int flags);
int open(char *filename, int flags, mode_t mod);
int close(int fd);
```

Opens file and returns fd for subsequent file accesses

flags: what operation we want to perform. OR combination of: `O_WRONLY`, `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC` ...

mode: file access permissions. bits: rwx rwx rwx (user/-group/others) (when file is created using `O_CREAT`)

open return value: ≥ 0: file descriptor, < 0 : error

close returns 0 iff successful

```
int fd = open("foo",
              O_CREAT | O_TRUNC | O_WRONLY,
              0666); // ugo:rw

int fd2 = open("foo2", O_RDONLY);
```

```
#include <unistd.h>
ssize_t write(int fd, void *buf, size_t count);
ssize_t read (int fd, void *buf, size_t count);
```

write count bytes to stream with file descriptor fd from buffer buf

read count bytes from stream with file descriptor fd to buffer buf

On success write returns the number of bytes written. On error, -1 is returned

On success read returns the number of bytes read. 0 indicates EOF. On error, -1 is returned

In case of an error occurring in C library functions global variable `errno` contains error code (`man errno`). `perror` prints it in readable form

```
write(fd, "hello", strlen("hello")+1);
read(fd, input, 10); // input must hold 10 bytes
```

Some Buffered I/O Functions

```
#include <cstdio>
FILE *fopen(const char *filename,
                              const char *mode);
int fclose(FILE *fp);
```

`fopen` opens a file for buffered access

mode:

`"r"`:	read	`"r+"`:	read & write
`"w"`:	write (truncate)	`"w+"`:	read & write (trunc.)
`"a"`:	append		

`fopen` returns 0 if something went wrong (`errno` contains error code)

`fclose` closes a file that was opened by `fopen` and returns 0 iff no error occurred.

```
FILE *fp_pread = fopen("read_rile", "r");
FILE *fp_write = fopen("write_file", "w");
fclose(fp_write);
fclose(fp_read);
```

Some Buffered I/O Functions (Continued)

```
#include <cstdio>
int fgetc(FILE *fp);
int fputc(int c, FILE *fp);
int feof(FILE *fp);
int ferror(FILE *fp);
void fflush(FILE *fp);
```

`fgetc` reads next character from stream. Returns value (≥ 0) on success or `EOF` in case of end-of-file or error

`fputc` writes character `c` to stream, returns `EOF` iff error occurred.

`feof` returns value $!= 0$ iff end of file reached

`ferror` returns value $!= 0$ iff error occurred

`fflush` forces a write of all buffered data to device/file. Returns value $!= 0$ iff error occurred.

As usual, global variable `errno` contains error code.
`perror("Remark");` prints error description

Some Buffered I/O Functions (Continued)

```
#include <cstdio>
size_t fwrite(void *p, size_t size, size_t n, FILE *fp);
size_t fread (void *p, size_t size, size_t n, FILE *fp);
```

`fwrite` writes `n` items of length `size` each to file `*fp` starting at address p

`fread` reads `n` items of length `size` each from file `*fp` and stores them beginning at p

`fwrite`/`fread` return number of successfully written/read items

Use `feof` and `ferror` to distinguish end-of-file and read errors

```
T a[N];
fread (a, sizeof(a[0]), N, fp);
fwrite(a, sizeof(a[0]), N, fp);
```

Writing and Reading Binary Data

If binary data is written and read by different computers it is important to use a hardware architecture and compiler independent data format, because struct/class layout and endianness may be different!

How can this be accomplished?

For each struct/class that needs to be written or read from a file, write serialize/deserialize functions that write respectively read the data using a general binary format

Example:

```
// error handling omitted for clarity

// write int to file (low-order byte first)
void serialize_int(int x, FILE *fp)
{
  to_little_endian(x);  // now low-order byte stored first
  fwrite(&x, sizeof(x), 1, fp); // store all bytes in file
}
// read int from file (low-order byte first)
void deserialize_int(int &x, FILE *fp)
{
  fread(&x, sizeof(x), 1, fp); // read all bytes from file
  from_little_endian(x); // restore byte order
}

struct X
{
  int a, b;

  // write struct data to file in binary
  int serialize(FILE *fp) {
    serialize_int(a, fp);
    serialize_int(b, fp);
  }
  // read struct data from file in binary
  int deserialize(FILE *fp) {
    deserialize_int(a, fp);
    deserialize_int(b, fp);
  }
};
```

Writing/Reading Objects From Files

```
// error handling omitted for clarity

// write X object to file on computer A
X u;
FILE *fp = fopen("save_u", "w");
u.serialize(fp);
fclose(fp);

// read X object from file on computer B
// that has access to the file
X u;
FILE *fp = fopen("save_u", "r");
u.deserialize(fp);
fclose(fp);
```

Formatted Output

```
typedef const char *ccptr;

int printf(ccptr format, ...);

  = fprintf(stdout, format, ...);

int fprintf(FILE *fp, ccptr format, ...);
```

- formatted data output
- variable # of parameters to be printed, must match format string. Modern compilers check that
- e.g. `printf("%d %d %f\n", i, j, real);` prints two integers and a double value in readable form to stdout

Format String

- %c : character
- %s : C-string
- %d : integer number
- %f : double precision floating point number
- %e : -"- , scientific notation
- ... many more: man fprintf
- %% = %
- general:
  % [flags] [width] [prec] [len-mod] conv-spec

printf Examples

```
#include <cstdio>

char c = 'x';
int i = 12345;
double f = 3.1415926535;
char s[] = "foo";

printf("%% c=%c i=%d f=%f s=%s", c, i, f, s);
// "% c=x i=12345 f=3.141593 s=foo"

printf("|%d TEST|", i);           // |12345 TEST|
printf("|%8dTEST|", i);           // |   12345TEST|
printf("|%08dTEST|", i);          // |00012345TEST|
printf("|%-8dTEST|", i);          // |12345   TEST|

printf("|%f TEST|", f);           // |3.141593 TEST|
printf("|%.1f TEST|", f);         // |3.1 TEST|
printf("|%7.2f TEST|", f);        // |   3.14 TEST|
printf("|%+13.8f TEST|", f);      // |  +3.14159265 TEST|

printf("|%.3e|", f);              // |3.142e+00|
```

Formatted Input

`int scanf(ccptr format, ...);`

= `fscanf(stdin, format, ...);`

`int fscanf(FILE *fp, ccptr format, ...);`

- formatted data input
- variable number of pointers to variables to be read, must match format string
- returns number of successfully read values
- `fscanf(fp, "%d %d %f", &i, &j, &real);` reads two integers and a double value and returns 3 if OK
- DANGEROUS! Types and number of format specifiers and pointers passed on to the functions must match. Hopefully the compiler reports type errors …

Input Examples

```
#include <cstdio>

int a, b, c;

if (scanf("%d %d %d", &a, &b, &c) != 3) {
  // fewer than 3 values read from stdin => error
}

int c = fgetc(fp);     // read one byte from file
if (c == EOF) {        // end of file reached or error
  if (feof(fp)) {      // end of file
    ...
  } else {             // error
    ...
  }
}

char buffer[N];
// read line from stdin until EOF, '\n', or length = N-1
int r = fgets(buffer, N, stdin);
if (r == 0) // nothing read or error
```

Random File Access

Imagine a scenario in which you would like to store data in a file and you also would like to frequently change individual data items

One option is to read the entire file content into memory and after making a few adjustments writing everything back to the file

Depending on the length of the file, this operation can be very time consuming, because reading files is much slower than changing individual items

A faster solution is to make use of C library function `fseek` that sets the read/write position of an opened file:

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

Modes:

`SEEK_SET`: set position to `offset`
`SEEK_CUR`: set position to current position + `offset`
`SEEK_END`: set position to file end position + `offset`

`fseek` returns 0 if successful, and -1 otherwise

fseek Example

```
// error handling omitted for clarity

#include <cstdio>

// open for read/write access
FILE *fp = fopen("foo", "r+");
int x;
int pos = 4;

// increment value stored at byte position pos

// read element - changes position in file
fseek(fp, pos, SEEK_SET);
fread(&x, sizeof(x), 1, fp);

x++;

// write it back - changes position in file
fseek(fp, pos, SEEK_SET);
fwrite(&x, sizeof(x), 1, fp);

fclose(fp);
```

fseek Application: FileArray.h

```
// class representing an int array stored in a file
#include <cstdio>

class FileArray
{
public:
  // open array file
  FileArray(const char *filename);

  // create array file holding N integers
  FileArray(const char *filename, int N);

  // close file
  ~FileArray();

  // retrieve element i from file
  int get(int i);

  // set element i in file to x
  void set(int i, int x);

  // return number of elements
  int size() { return n; }

private:
  FILE *fp; // array file handle
  int n;    // number of elements
};
```

fseek Application: FileArray.cpp (1)

```
#include <cstdio>
#include <cstdlib>
#include <cassert>
#include "FileArray.h"

// Format of array file of size N:
// N               (4 bytes)
// array: N ints   (N*4 bytes)

// open array file
FileArray::FileArray(const char *filename)
{
  fp = fopen(filename, "r+");
  if (!fp) {
    fprintf(stderr, "can't open file %s\n", filename);
    exit(10);
  }

  if (fread(&n, sizeof(n), 1, fp) != 1) {
    fprintf(stderr, "can't read array size from file %s\n", filename);
    exit(10);
  }

  if (n <= 0) {
    fprintf(stderr, "corrupt length %d in file %s\n", n, filename);
    exit(10);
  }
}
```

fseek Application: FileArray.cpp (2)

```
// create array file holding N integers
FileArray::FileArray(const char *filename, int N)
{
  assert(N > 0);
  n = N;

  fp = fopen(filename, "w");
  if (!fp) {
    fprintf(stderr, "can't open file %s\n", filename);
    exit(10);
  }

  if (fwrite(&n, sizeof(n), 1, fp) != 1) {
    fprintf(stderr, "can't write array size to file %s\n", filename);
    exit(10);
  }

  for (int i=0; i < n; ++i) {
    int x = 0;
    if (fwrite(&x, sizeof(x), 1, fp) != 1) {
      fprintf(stderr, "can't write data to file %s\n", filename);
      exit(10);
    }
  }
}

// close file
FileArray::~FileArray()
{
  fclose(fp);
}
```

fseek Application: FileArray.cpp (3)

```
// retrieve element i
int FileArray::get(int i)
{
  assert(i >= 0 && i < n);
  if (fseek(fp, (i+1) * sizeof(int), SEEK_SET)) { // why +1?
    perror("get fseek error");
    exit(10);
  }
  int x;
  if (fread(&x, sizeof(x), 1, fp) != 1) {
    fprintf(stderr, "read error %d\n", i);
    exit(10);
  }
  return x;
}


// set element i to x
void FileArray::set(int i, int x)
{
  assert(i >= 0 && i < n);
  if (fseek(fp, (i+1) * sizeof(int), SEEK_SET)) {
    perror("get fseek error");
    exit(10);
  }
  if (fwrite(&x, sizeof(x), 1, fp) != 1) {
    fprintf(stderr, "read error %d\n", i);
    exit(10);
  }
}
```

fseek Application: FileArrayMain.cpp

```
#include <cstdlib>
#include <ctime>
#include "FileArray.h"

int main()
{
  const int N = 1’000’000;

  // first create array file
  {
    FileArray fa("test", N);
    // file closed here
  }

  // then use it
  FileArray fa("test");

  printf("array size %d\n", fa.size());

  // random seed based on time
  srandom(time(0));

  // increment random elements
  for (int i=0; i < 1’000’000; ++i) {
    // almost uniformly distributed good enough for our purpose
    // but slightly biased when (RAND_MAX+1) % N != 0
    int j = random() % N;
    int x = fa.get(j);
    x++;
    fa.set(j, x);
  }
  return 0;
}
```

C++ Input/Output

Thus far we have concentrated on C input/output functions

C++ introduces stream classes that are easier to handle, safer, and more extensible

Not all is perfect, though. Formatted output, for instance, is a bit awkward, in that manipulators have to be used which change the state of output streams and have to be reset for subsequent outputs

```
// Input examples

#include <iostream>
using namespace std;  // This is where the compiler
                      // looks for symbols
                      // Can omit std:: now
                      // Only use in .cpp files!
double x;
int i;

cin >> x >> i;    // read double, then read int
                  // skip white space
if (!cin) {       // read didn't succeed
  if (cin.eof()) // end of input reached
  ...
```

```
// Ouput examples

#include <iostream>
using namespace std;

double x;
int i;

cout << x << ' ' << i << '\n';
// First write double, then space,
// then int, then newline character

// formatted C output
printf("%11.6f", x);

// Formatted C++ output using stream manipulators.
// Problem: stream state changes persist
#include <iomanip>
cout << fixed << setw(11) << setprecision(6) << x;

// If you like C format strings use boost::format.
// The program will be aborted if the parameters
// don't match the format string.
#include <boost/format.hpp>
cout << boost::format("%11.6f") % x;
```

Mixing C and C++ input/output is discouraged because of possible buffering issues

For more information on C++ input/output visit
`www.cplusplus.com/reference/iostream/`