C File Processing

CS 2060

Prof. Jonathan Ventura

Files and Streams

- Files are used for long-term storage of data
  - (on a hard drive rather than in memory).

Files and Streams

- Files are used for long-term storage of data
  - (on a hard drive rather than in memory).
- C views a file as a sequential stream of bytes, beginning at index zero.

Files and Streams

- Files are used for long-term storage of data
  - (on a hard drive rather than in memory).
- C views a file as a sequential stream of bytes, beginning at index zero.
- Files have dynamic length, meaning that they grow bigger as we add data.

FILE structure

- A file is represented by a `FILE` structure, defined in `stdio.h`:

```
FILE *f = NULL;
```

- The file processing functions all use pointers to `FILE`s.

End-of-file marker

- The end of the file is marked with an *end-of-file marker* (EOF).

0	1	2	3	EOF

Table: A file with four bytes

Opening a file

- To open a file, use `fopen`:

```
FILE *f = fopen("hello.txt","r");
```

- `fopen` returns a `FILE` pointer.
- The first argument is the file name.
- The second argument is the *file open mode*.

File open modes

- There are many file open modes; the most common are shown here:

Mode	Description	Notes
`"r"`	Read-only	Opens existing file
`"w"`	Write-only	Creates new file, deleting old file if it exists
`"a"`	Append	Begins with file pointer at end of file
`"r+"`	Read and write	Opens existing file
`"w+"`	Read and write	Creates new file, deleting old file if it exists

Opening a file

- If `fopen` fails to open the requested file, it will return `NULL`.

Opening a file

```
// try to open file
FILE *f = fopen(filename,"r");
// check if it succeeded
if ( f == NULL ) {
    // print error message
    printf( "Could not open %s for reading\n",
            filename );
    exit(1);      // bail
}
```

Opening a file

- `fopen` might fail for several reasons, e.g.:
  - Trying to open as read-only a file that does not exist
  - Trying to open a file at a path that is not reachable (directory does not exist)
  - Insufficient permissions

Closing a file

- When we are done with the file, we need to *close* it with `fclose`.

Closing a file

- When we are done with the file, we need to *close* it with `fclose`.
- On some platforms, only a limited number of files can be open at once.

Closing a file

- When we are done with the file, we need to *close* it with `fclose`.
- On some platforms, only a limited number of files can be open at once.
- Opened files will be closed when your program exits.

Closing a file

```
// open log file for appending
FILE * f = fopen("log.txt","a");
// ... append record to file ...
// close log file
fclose( f );
```

Reading from a file

- Reading from a file is similar to reading from the console.

Reading from a file

- Reading from a file is similar to reading from the console.
- You can use `fscanf` for formatted input from a file.

Reading from a file

- Reading from a file is similar to reading from the console.
- You can use `fscanf` for formatted input from a file.

- `fscanf(f,"%d",&int);`

Reading from a file

- Reading from a file is similar to reading from the console.
- You can use `fscanf` for formatted input from a file.

- `fscanf(f,"%d",&int);`

- Remember to specify the file as the first argument to `fscanf`.

Reading from a file

```
// Open hello.txt as read-only
FILE *f = fopen("hello.txt","r");
// Read five integers from file
int a[5];
for ( int i = 0; i < 5; i++ ) fscanf(f,"%d",a+i);
// Close file
fclose(f);
```

Reading characters

- `fgetc`: reads individual characters:

```
// Open input.txt
FILE *f = fopen("input.txt","r");

// Print characters from input.txt
// Stop when we see end-of-file marker
char c;
while ( ( c = fgetc(f) ) != EOF ) putchar(c);

// Close file
fclose(f);
```

Reading characters

- `fgets`: reads a line of text:

```
// Print lines from file
// Stop when we see end-of-file marker
char line[1024];
while ( 1 ) {
    // Read line from file
    char *ptr = fgets(line, 1024, f);

    // If result is NULL, either error occured or EOF
    if ( ptr == NULL ) break;

    // Print line
    printf("%s\n",ptr);
}
```

The file pointer

- The `FILE` struct maintains a *file pointer* which points to where we are in the file.

Testing end-of-file

- `feof` tests if the file pointer is at the end-of-file marker:

```
FILE *f = fopen("data.txt","r");
// Read characters until we see the end-of-file marker
while ( !feof(f) ) {
    char c = fgetc(f);
    putchar(c);
}
fclose(f);
```

Rewind

- To move the file pointer back to the beginning of the file, use `rewind`:

```
FILE *f = fopen("data.txt","r");
// Read characters until we see the end-of-file marker
while ( !feof(f) ) {
    char c = fgetc(f);
    putchar(c);
}
// Rewind back to the beginning
rewind(f);
// ...
```

Formatted print

- `fprintf` writes formatted data to a file (analogous to `printf`):

```
// Open output.txt
FILE *f = fopen("output.txt","w");

// Write out some messages
fprintf(f,"Hello, world!\n");
fprintf(f,"2 + 5 = %d\n", 2 + 5 );

// Close the file
fclose(f);
```

Writing characters

- `fputc` writes individual characters:

```
// Open input.txt and output.txt
FILE *f = fopen("input.txt","r");
FILE *fout = fopen("output.txt","w");

// Copy characters from input.txt to output.txt
// Stop when we see end-of-file marker
char c;
while ( ( c = fgetc(f) ) != EOF ) fputc(c,fout);

// Close files
fclose(f);
fclose(fout);
```

Reading/writing raw data

- So far we have only considered text files.

Reading/writing raw data

- So far we have only considered text files.
- We can also store "raw" data in a file as bytes, integers, floats, etc.

Reading/writing raw data

- So far we have only considered text files.
- We can also store "raw" data in a file as bytes, integers, floats, etc.
- For example, reading/writing an integer as four bytes rather than as an ASCII string is faster and saves storage space.

Reading/writing raw data

- So far we have only considered text files.
- We can also store "raw" data in a file as bytes, integers, floats, etc.
- For example, reading/writing an integer as four bytes rather than as an ASCII string is faster and saves storage space.
- We can also read/write `structs`, etc.

Reading raw data with `fread`

- `fread` reads a specified number of bytes from a file:

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

- `ptr` is where the data will be written to.
- `size` is the number of bytes *in one item*.
- `nitems` is the number of items to read.
- The return value is the number of items read.

Reading integers stored as binary

Reading raw integers with `fread`:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

// How to read one integer from the file?
int value;

// Close the file
fclose(f);
```

Reading integers stored as binary

Reading raw integers with `fread`:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

int value;
fread( &value, sizeof(int), 1, f );

// Close the file
fclose(f);
```

Reading integers stored as binary

Reading raw integers with `fread`:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

int value;
if ( fread( &value, sizeof(int), 1, f ) != 1 ) {
    printf("Read failed.\n");
}

// Close the file
fclose(f);
```

Reading integers stored as binary

Reading raw integers with `fread`:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

// How to read 10 integers?

// Close the file
fclose(f);
```

Reading integers stored as binary

Reading raw integers with `fread`:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

int values[10];
if ( fread( values, sizeof(int), 10, f ) != 10 ) {
    printf("Read failed.\n");
}

// Close the file
fclose(f);
```

Reading a struct

Reading a `struct` with `fread`:

```
struct Point { int index, float x, float y };

// ...
struct Point pt;
fread( &pt, sizeof(struct Point), 1, f );
```

Reading many structs

Reading many `structs` with `fread`:

```
struct Point { int index, float x, float y };

// ...
struct Point pts[10];
fread( pts, sizeof(struct Point), 10, f );
```

Writing raw data with `fwrite`

- `fwrite` writes a specified number of bytes to a file:

```
int fwrite( void *ptr, size_t size, size_t nitems, FILE *stream );
```

- `ptr` is where the data will be read from.
- `size` is the number of bytes *in one item*.
- `nitems` is the number of items to write.
- The return value is the number of items written.

Writing integers stored as binary

Writing raw integers with `fwrite`:

```
// Open the file for writing
FILE *f = fopen("integers.dat","w");

// How to write one integer to the file?
int value = 10;

// Close the file
fclose(f);
```

Writing integers stored as binary

Writing raw integers with `fwrite`:

```
// Open the file for writing
FILE *f = fopen("integers.dat","w");

int value = 10;
fwrite( &value, sizeof(int), 1, f );

// Close the file
fclose(f);
```

Writing integers stored as binary

Writing raw integers with `fwrite`:

```
// Open the file for writing
FILE *f = fopen("integers.dat","w");

int value;
if ( fwrite( &value, sizeof(int), 1, f ) != 1 ) {
    printf("Write failed.\n");
}

// Close the file
fclose(f);
```

Writing integers stored as binary

Writing raw integers with `fwrite`:

```
// Open the file for writing
FILE *f = fopen("integers.dat","w");

// How to write 10 integers?
int values[10] = { 0,1,2,3,4,5,6,7,8,9 };

// Close the file
fclose(f);
```

Writing integers stored as binary

Writing raw integers with `fwrite`:

```
// Open the file for writing
FILE *f = fopen("integers.dat","w");

int values[10] = { 0,1,2,3,4,5,6,7,8,9 };
if ( fwrite( values, sizeof(int), 10, f ) != 10 ) {
    printf("Write failed.\n");
}

// Close the file
fclose(f);
```

Writing a struct

Writing a `struct` with `fwrite`:

```
struct Point { int index, float x, float y };

// ...
struct Point pt = { 0, 1.f, 1.f };
fread( &pt, sizeof(struct Point), 1, f );
```

Writing many structs

Writing many `structs` with `fwrite`:

```
struct Point { int index, float x, float y };

// ...
struct Point pts[10];
pts[0].index = 0;   pts[0].x = 10;  pts[0].y = 10;
// ...
fwrite( pts, sizeof(struct Point), 10, f );
```

Random Access Files

- A *random access file* contains a sequence of identically-sized data records.

Starting byte #:	0	100	200	300	...
Record #:	0	1	2	3	...

Table: A random access file with 100-byte records

Random Access Files

- Because of this structure, we can easily jump to any records without having to read through the entire file.

Starting byte #:	0	100	200	300	...
Record #:	0	1	2	3	...

Table: A random access file with 100-byte records

Moving the file pointer with `fseek`

- `fseek` moves the file pointer to a specified location:

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

- `stream` is the file
- `offset` is the offset to jump to
- `whence` indicates the reference location for the `offset`
  - `SEEK_SET` to offset from the beginning
  - `SEEK_END` to offset from the end
  - `SEEK_CUR` to offset from the current position

Random Access Files

- How do we calculate the offset for record n ?

Starting byte #:	0	100	200	300	...
Record #:	0	1	2	3	...

Table: A random access file with 100-byte records

Random Access Files

- Record n is at offset $100 \times n$.
- Or: `sizeof(struct Record)*n`.

Starting byte #:	0	100	200	300	...
Record #:	0	1	2	3	...

Table: A random access file with 100-byte records

Initializing a random access file

```
/** ClientData
 * Stores data for a bank client
 */
typedef struct {
    unsigned int account_num;    // account number
    char last_name[15];          // account last name
    char first_name[10];         // account first name
    double balance;              // account balance
} ClientData;
```

Initializing a random access file

```
// Create a random access file with room for 100 records
int main() {
    // try to open file
    FILE *f = fopen( "accounts.dat", "w" );

    // test if file open failed (f is NULL)
    if ( !f ) {
        puts("Could not open accounts.dat for writing.");
        return 1;
    }
```

Initializing a random access file

```
    // create empty record
    ClientData empty_record = { 0, "", "", 0.0 };

    // output 100 blank records to file
    for ( int i = 0; i < 100; i++ ) {
        fwrite( &empty_record, sizeof(ClientData), 1, f );
    }

    // close file
    fclose(f);
}
```

Initializing a random access file

```
    // create empty record
    ClientData empty_record = { 0, "", "", 0.0 };

    // output 100 blank records to file
    // why not this: ?
    fwrite( &empty_record, sizeof(ClientData), 100, f );

    // close file
    fclose(f);
}
```

Reading a random access file

```
int main() {
    // open file (will skip failure test here for brevity)
    FILE *f = fopen( "accounts.dat", "r" );

    // ask for record number
    puts("Enter record number:");
    int record_num = 0;
    scanf( "%d", &record_num );
```

Reading a random access file

```
// Move file pointer to requested record
fseek( f, record_num*sizeof(ClientData), SEEK_SET );

// Read requested record
ClientData client_data;
fread( &client_data, sizeof(ClientData), 1, f );
```

Reading a random access file

```
        // Print requested record
        printf( "Account number: %u", client_data.account_num );
        printf( "Last name: %s", client_data.last_name );
        printf( "First name: %s", client_data.first_name );
        printf( "Account balance: %lf", client_data.balance );

    }
```