fopen(“Lecture08”,”r”);

Steven R Bagley

Reading a number

- How would we read a number from the keyboard?
- Could use `getchar()`
- And build up the number digit by digit
- C provides us with functions to do this

demo creation

Keyboard Input

- Coursework two asks you to use `scanf()` to read from the keyboard
- It has a syntax similar to `printf`

```
scanf("%d", &height);
```

- But also has all these `&` because `scanf` can read more than one value
- But functions can only return one…

I/O

- So far we've looked at reading from keyboard and printing to the screen
- By default, `getchar()`, `printf()`, `scanf()` etc. all refer to these
- But UNIX also provides a second output, the standard error output (`stderr`)
- Generally used for all error output…

`stderr`

- To write to the `stderr` output we need to specifically tell C to output to it
- UNIX treats everything as files
- And so to write to `stderr` we need to understand file I/O in C

C's File I/O

- Fortunately, C's file I/O is very simple
- Uses similar functions to those we've used `fscanf()`, `fgetc()`, `fprintf()`, etc...
- The ones already seen are just specific versions of them
- Must specify which `FILE` to read/write to

`FILE` struct

- C's file I/O is based around a new type `FILE`
- Or more specifically a pointer to it
- This is an example of a user-defined type
- Defined by `stdio.h` — technically part of the C librarys, not the language
- Will see how to define our own types later…

Writing to a `FILE`

- Write to a `FILE` using `fprintf()`
  `int fprintf(FILE *fp, char *format, ....)`
- Works in the same way as `printf()` except we need to provide a `FILE` pointer as the first argument
- Either use a standard one, or create one with `fopen()`

`stdin`	Standard input	read only
`stdout`	Standard output	write only
`stderr`	Standard error output	write only

fprintf(stdout -- same as printf

Go demo it…

`stdin`	Standard input	read only
`stdout`	Standard output	write only
`stderr`	Standard error output	write only

} default for `scanf`/`printf`

fprintf(stdout -- same as printf

Go demo it…

Reading from a `FILE`

- Again, an equivalent to `scanf()`
  `int fscanf(FILE *fp, char *format, ....)`
- And for `getchar()`
  `int fgetc(FILE *fp)`
  `int getc(FILE *fp)`
- Same functionality, but the latter is a `#define` macro rather than a function
- But what do we read from?

getc() Faster, but might not work in all situations…

Opening a `FILE`

- C provides `fopen()` to open files
  `FILE *fopen(char *path, char *mode);`
- File to open given in string `path`
- The `mode` string specifies how to open file
- Returns a pointer to a `FILE`
- Can't open a non-existant file for reading, so returns `NULL`

We don't know where the FILE is in memory, created and returned by fopen
Need to close the file with fclose later
Only if the FILE

mode	Meaning
r	Open for reading only.
r+	Open for reading and writing
w	Open for writing only. Truncates file to zero length or creates file
w+	Open for reading and writing. Truncates file to zero length or creates file
a	Append; open for writing only. File created if it doesn't exist
a+	Append; open for reading and writing. File created if it doesn't exist

append means writes always happen at end of file…

```
FILE *fp;
int x;

fp = fopen(“file”, “r”);
fscanf(fp, “%d”, &x);
fclose(fp);
```

Open a file for reading
Need to close it.
Go and demo -- modify celsius to fahrenheit program to read values from file.

Open and Close

- Must close a `FILE` when finished with it
  `fclose(fp)`
- Otherwise, data may be lost
- C uses buffered I/O — so data isn't written to the file straight away
- Can use `fflush()` to force it to be written

Closing files

- All open files will be closed when the program finishes
- But *very* good practice to close them as soon as you have finished
- Limit to the number of open files in OS
- Also saves memory

Function	Meaning
`int getc(FILE *fp)` `int fgetc(FILE *fp)`	reads a character from a file
`int putc(int c, FILE *fp)` `int fputc(int c, FILE *fp)`	Write a character to a file
`int fscanf(FILE *fp, char *format, ... )`	As `scanf` — but reads from the file `fp`
`int fprintf(FILE *fp, char *format, ...)`	As `printf` — but writes to the file `fp`
`int feof(FILE *fp)`	Returns true if end of file has been reached
`int ungetc(int c, FILE *stream)`	Ungets a character. The next character read will be the ungotten character.
`int fputs(char *str, FILE *stream)`	Write the string `str` to the `stream`
`char *fgets(char *str, int size, FILE *stream)`	Read a string into `str` of maximum `size` characters

Go modify our c2f program…

Random-access

- By default, you'll move sequential through a file from beginning to end
- Indeed, for some 'files' this is the only thing that makes sense (e.g. keyboard, network)
- However, for regular files on disk C lets you have *random access* to any part of the file

Remember, in UNIX everything is a file…

Seeking

- C provides the `fseek()` function to move to a particular location in the file
  `int fseek(FILE *fp, long offset, int whence)`
- Move to `offset` bytes from `whence`
- Positive `offset` moves forward, negative `offset` move back
- What's `whence`? Specifies where the `offset` is relative to

whence	Meaning
SEEK_SET	Move relative to the start of the file
SEEK_CUR	Move relative to the current position in the file
SEEK_END	Move relative to the end of the file

All #defines for actual values

```
fseek(fp,   4, SEEK_SET); /* Move to the fifth character in the file */

fseek(fp,   4, SEEK_CUR); /* Move four characters ahead */
fseek(fp,  -4, SEEK_CUR); /* Move four characters back */

fseek(fp,   0, SEEK_SET); /* goto the start – see also rewind() */
fseek(fp,   0, SEEK_END); /* goto the end */

fseek(fp, -10, SEEK_END); /* goto 10 characters before the end */
```

Function	Meaning
`int fseek(FILE *stream, long offset, int whence)`	Moves to specified point in the file. Returns `0` if successful
`long ftell(FILE *stream)`	returns current offset within `stream`
`void rewind(FILE *stream)`	Go to the start of the file, identical to `(void)fseek(stream, 0, SEEK_SET)`
`int fgetpos(FILE *stream, fpos_t *pos);`	Alternative for moving around files, can cope with file sizes bigger than can be stored in a `long`
`int fsetpos(FILE *stream, fpos_t *pos);`	

fpos_t is another user-defined type
Write a program to read a file backwards :)

Binary

- So far, we've been reading ASCII from files
- `fscanf()`/`fprintf()` read/write ASCII versions of `int`s etc.
- What if we want to read the *binary* representation?

Opening Binary files

- Need to open a file for binary access
- Put a b in the mode string

```
fp = fopen("file", "rb");
fp = fopen("file", "w+b");
```

- Alters the way C processes certain characters
- Need to use binary access functions

Function	Meaning
`size_t fread(void *p, size_t size, size_t nitems, FILE *fp)`	Read `nitems` of size `size` from `fp` to memory pointed to by `p`
`size_t fwrite(void *p, size_t size, size_t nitems, FILE *fp)`	Write `nitems` of size `size` to `fp` from memory pointed to by `p`
`size_t sizeof(`*`TYPE`*`)`	returns the size of the type TYPE in memory in bytes

size_t is just an integer value

```
int x = 42;
FILE *fp = fopen(“bin”, “wb”);

fwrite(&x, sizeof(int), 1, fp);

fclose(fp);
```

Go write a program to read in the header of a TARGA file…

Read and write

- Often see it with size set to `1` and use `nitems` to specify the number of bytes to read
- Care should be taken with binary
- Particular host endianness when transferring files across machines

Memory blocks

- Real power comes from reading and writing large memory blocks of data
- Such as an array…