I/O Demo Session

Lydia Duncan

October 7, 2025

Outline

- The Basics
- The IO Library
- Opening A File
- Reading and Writing
- Formatted I/O
- I/O Transactions
- Sir Not-Appearing-In-This-Presentation

The Basics

Basic I/O

- Can print to console without a ‘use’ or ‘import’ statement

```
writeln("Hello, world!");
```

- Can also print partial lines using ‘write’

```
write("The first 5 primes are: ");
for prime in getPrimes(numPrimes=5) {
  write(prime, " ");
}
writeln();
// Prints “The first 5 primes are: 2 3 5 7 11 ” on a single line
```

- And print formatted output using ‘writef’

```
writef("%i in octal is %oi\n", 16, 16);  // Prints “16 in octal is 20”
```

Basic I/O: Types

- Basic Chapel types are easily written
  - No string cast necessary!

```
writeln("Hello, world!"); // E.g., strings ...
writeln(1..5);            // ranges ...
writeln(3.14);            // reals ...
writeln([1, 2, 4, 8]);    // arrays, etc.
```

- Classes and records default to printing out their fields

```
class Foo {
  var x: int;
  var y: bool;
}

var f = new Foo(3, false);
writeln(f); // prints "{x = 3, y = false}". A record would use ( ) instead of { }
```

See Ben's demo tomorrow on how to control the way types are printed!

Basic I/O: The Libraries

- For a lot of programs, this is enough
- When you want something more, then it's time to turn to the IO libraries!

```
use IO;
```

- The 'IO' library provides more extensive operations
  - File operations
  - Specialized reading and writing

  - It also defines a submodule, 'FormattedIO', for more format string operations
    - Formatted reading/writing to files
    - Format methods on the 'string' and 'bytes' types
    - And some regex operations

- There's also the 'ParallelIO' module, which is currently unstable
  - Won't cover today, but worth knowing about and exploring

Quick Exercise 1

- Take a couple of minutes to create a short Chapel program that writes an array to the console
- Now adjust that program to print the index before each value

The IO Library

The IO Library

- The IO library is mainly focused around three types:
  - ‘file’
  - ‘fileReader’
  - ‘fileWriter’

- The ‘file’ type represents a file in the operating system

```
var f = open(“myFile.txt”, ioMode.rw);
```

- The ‘fileReader’ and ‘fileWriter’ types allow reading and writing from an associated file

```
var r = f.reader();
var w = f.writer();
```

‘file’ vs. ‘fileReader’/’fileWriter’

- Why separate types?
  - Better supports parallel operations
    - One ‘file’ instance can have multiple ‘fileReader’/’fileWriter’s associated with it
    - If file regions covered don’t overlap, this is parallel safe!

- If don’t need multiple for a file, can open directly

```
var r = openReader(“myFile.txt”);
var w = openWriter(“myOtherFile.txt”);
```

  - A ‘file’ will still be created, but it’ll only be owned by the ‘fileReader’/’fileWriter’

stdout/stdin/stderr

- Our previous examples were writing to ‘stdout’ (standard output)
  - Chapel also provides ‘stdin’ (standard input) and ‘stderr’ (standard error)
  - Represent the normal programming concepts
  - ‘stderr’ can be written to by calling its various ‘write*’ methods
    - And ‘stdin’ can be read from by calling its ‘read*’ methods or the free standing ‘read’ functions in the ‘IO’ library

- These are instances of the ‘fileWriter’ and ‘fileReader’ types, provided by the ‘IO’ library

Opening a File

Opening a File

```
var f = open("myFile.txt", ioMode.rw);
```

The path to the file

The permissions to open the file with

ioMode option	What it means
r	read
cw	create and write
rw	read and write
cwr	create, read and write
a*	append

- 'open( )' also can take an 'ioHintSet' argument
  - Used to specify optimization hints to the file system
  - 'ioHintSet's can be combined using '|' and '&'
  - And compared using '==' or '!='
  - If one isn't provided, defaults to 'empty'

- This function can throw errors

* append is currently unstable

Opening fileReaders/fileWriters

- The file 'reader( )' and 'writer( )' methods don't need arguments
  - But there are some optional ones to provide

```
var r = f.reader(locking, region, hints, deserializer);
var w = f.writer(locking, region, hints, serializer);
```

Whether it should lock

The part of the file to read/write

Optimization hints

How to read/write the file's contents

- 'openReader'/'openWriter' also can take these arguments
  - But since there's no associated file, they require the path

Again, see Ben's demo tomorrow on serializers/deserializers!

- These functions can also throw errors

Cleaning Up

- When you're done with all these types, you can close them

```
r.close();
f.close();
```

- But all fileReaders and fileWriters must be closed before their files are closed
  - All these types will close themselves automatically when they go out of scope

```
proc someFunc() {
  var f = open("myFile.txt", ioMode.rw);
  …
  return somethingUnrelated;
}
```

'f' will be closed after this return

Quick Exercise 2

- Make a new Chapel program that opens a file to “oldMcDonald.txt” for reading
  - This file is defined in ‘https://github.com/chapel-lang/chapelcon25-demos/IO/’
- Now open a ‘fileReader’ on that file
- Can you write an ‘openReader’ call that does the same thing?

ioMode option	What it means
r	read
cw	create and write
rw	read and write
cwr	create, read and write
a*	append

Reading and Writing

Reading From A File

- ‘fileReader’ provides many methods for reading from a file
  - We’ll focus on ‘read’ and ‘readLine’
    - ‘read’ can take in one or more arguments to fill

```
var w: int;
r.read(w); // returns ‘false’ if nothing was read
```

    - Or it can take one or more type arguments:

```
var x = r.read(int);
var (a, b, c) = r.read(int, bool, bool);
```

    - ‘readLine’ will read through a newline and place the result in either the ‘string’, ‘bytes’, or array of ‘bytes’ argument

```
var s: string;
r.readLine(s);
```

Can also specify the length and whether the newline character itself should be included

    - Alternatively, can request the type to return (must be either ‘string’ or ‘bytes’)

```
var s = r.readLine(string);
```

Ditto

Reading From A File, cont.

- ‘fileReader’ also provides a ‘lines’ iterator
  - This allows you to traverse the entirety of the file

```
for line in r.lines() {
  …
}
```

  - It defaults to requiring no arguments, but can also specify:
    - Whether the newline characters should be removed from the line

```
for line in r.lines(stripNewline = true) { … }
```

    - Whether the line should be a ‘string’ or ‘bytes’

```
for line in r.lines(t = bytes) { … }
```

Writing To A File

- ‘fileWriter’ provides several methods for writing to a file
  - We’ll focus on ‘write’ and ‘writeln’
    - These are the same as the standalone versions from earlier
    - Just instead defined as methods

```
w.write("Hello, ”);
var s = "world!";
w.writeln(s);
```

Quick Exercise 3

- Grab your program that opened “oldMcDonald.txt” for reading
  - Let’s extend it to read the contents of the file and print it to the console

- Did you accidentally print a newline in between each line of the file? ☺

Formatted I/O

Formatted I/O: Motivation

- In many cases, the ability to comma-separate intended output will be sufficient

```
writeln("Here I am interspersing an int, ", 5, ", with some text");
```

- But sometimes you want more extensive control of *how* your output looks
  - Or what parts of the output you want to read in

- Our focus here will be 'readf( )'/'fileReader.readf( )', 'fileWriter.writef( )', and the 'format' methods
  - But the library also contains 'fileReader' methods for using regex

Formatted I/O

- ‘readf’/’fileReader.readf’ take the pattern, followed by the arguments to read

```
var month: string;
var day: int;
var year: int;
r.readf("Today is %s %i, %i\n", month, day, year);
```

- ‘fileWriter.writef’ is specified the same, but the arguments passed will be used to fill in the pattern

```
var month: string = “October”;
var day: int = 7;
var year: int = 2025;
w.writef("Today is %s %i, %i\n", month, day, year);
```

- Instead of sending the pattern in as an argument, the format methods are called on the pattern itself

```
var fmtStr = "Today is %s %i, %i\n";
var res = fmtStr.format(month, day, year);
```

Format Specifiers

- Using a format string starts with a base format specifier for the type
- The letter used tries to be the most natural for that type
  - Though it's not always possible – see 'imag' and 'complex'

- If you don't know the type, you can use:
  - '%n' if you know it's a number
  - '%?' if you don't know the type, or know it's a record or class
    - This will use the fileWriter/fileReader's associated serializer/deserializer

Have I convinced you to see Ben's demo yet? ☺

Type	Format Specifier
int	%i
uint	%u
real	%r
imag	%m
complex	%z
string	%s
Single character	%c

Conversion Specifiers

- Many of the types can be modified for greater control
  - Padding the output (e.g., '%17')
    - Aligning the value (see table)
    - Padding with zeroes (e.g., '%0')
  - Converting to binary, octal, or hexadecimal (see table)
    - We used '%o' in our 'writef' example
  - Prefixing positive numbers with '+' (e.g., '%+')

Alignment specifier	Meaning
<	Left justify, e.g. "15 "
^	Center justify, e.g. " 15 "
>	Right justify, e.g. " 15"

Numeric conversion	Meaning
b	binary
o	octal
x	hexadecimal
d	decimal

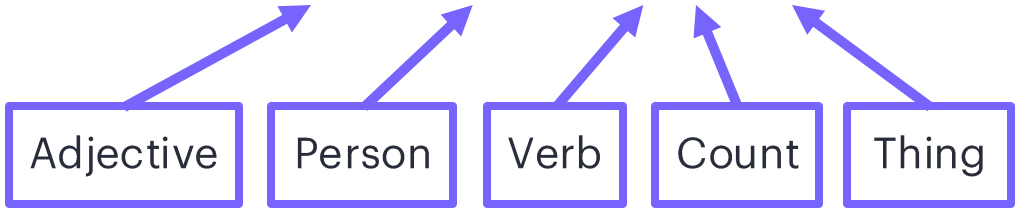
Conversion Specifiers Applicability

- Many of the types can be modified for greater control
  - Padding the output (e.g., ‘%17’)
    - Aligning the value (see previous slide)
    - Padding with zeroes (e.g., ‘%0’)
  - Converting to binary, octal, or hexadecimal (see previous slide)
  - Prefixing positive numbers with ‘+’ (e.g., ‘%+’)

Adjustment specifier	Applicable type specifiers
<, ^, >	i, u, r, m, z, s
b, o	i, u
x, d	i, u, r, m, z
+	i, u, r, m, z
0 (zero)	i, u, r, m

Quick Exercise 4

- Grab your program that opened “oldMcDonald.txt” for reading again
  - Instead of reading the whole file, let’s get specific parts out of the first line
  - The first line says “Old McDonald had a farm”



- Make a ‘readf’ call to get each of these components out of the first line

- How would you use ‘writef’ to print with a number instead of ‘a’ for the count?
  - Hint: you could use ‘%i’ to print an integer instead of a string

I/O Transactions

I/O Transactions

- Chapel offers a way to speculatively perform operations involving a ‘fileReader’/’fileWriter’
  - If something goes wrong, can fall back to the earlier state
    - E.g., if expected next value read to be something specific and it wasn’t

- This involves “transactions”
  - Try a sequence of actions on the ‘fileReader’/’fileWriter’
    - If they work, ‘commit’ and continue
    - If they don’t, ‘revert’ and try something else

```
r.mark();
…
if success {
  r.commit();
} else {
  r.revert();
  …
}
```

Exercise 5

- The oldMcDonald text file has a bad animal noise in it
  - Use transactions to find and print the line with the bad animal noise in it

Sir Not-Appearing-In-This-Presentation

Not Covered

- IO library features
  - Other ways to open a file: Alternate initializers, openTempFile, openMemFile
  - openStringReader, openBytesReader,
  - Specific ioHintSets
  - Other read methods: readLiteral, readNewline, matchLiteral, matchNewline, readThrough, readTo, readAll, readString, readBytes, readBits, readCodepoint, readByte, readBinary, readln
  - Other write methods: writeLiteral, writeNewline, writeBits, writeCodepoint, writeByte, writeString, writeBytes, writeBinary
  - IO transaction methods: offset, advance, advanceThrough, advanceThroughNewline, advanceTo
  - seek
  - File properties: isOpen, path, size
  - flush
  - assertEOF
- FormattedIO library methods: extractMatch, search, matches
- ParallelIO library

Thank You