Appendix A: Refresher on Files and the Shell

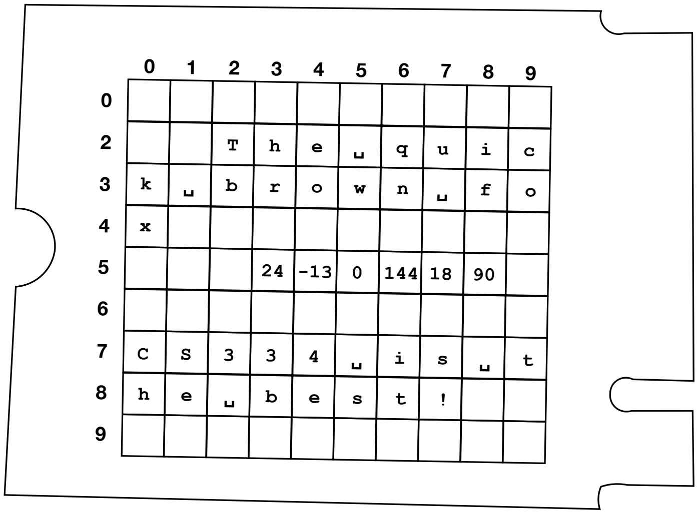
If you don't know what files are, or you feel a little fuzzy on them, read this chapter. Many students now studying CS grew up in a world where data is stored in the cloud, so they may never have needed to manage files directly. Files are a foundational concept in computer science. They are used in all but the most trivial programs, so it's worth your time to know what they are. This tutorial will show you what they are, where to find them, and how to work with them in a shell.

A *file* is an array of bytes stored a disk. A *disk* is just a physical device capable of storing files when a computer is powered off. To make things concrete, let's start by discussing disks and proceed from there.

Disks and Filesystems

The most common disk now used in consumer-grade computer hardware like laptops and phones is something called a *solid-state disk*, commonly referred to as an *SSD*. The term *sold-state* conveys the fact that the device is composed entirely of electronics and contains no moving parts. A typical SSD like the kind you might find in your laptop is shown in Figure 25.

Figure 25: A 256GB NVMe SSD from a Dell laptop. This device is roughly the size of the first digit of your thumb. In 2025, an SSD can store data at a cost of roughly $0.040 per billion bytes.



To a first approximation, the circuitry inside an SSD really does store data in a big array. In the diagram above, we have three files, a text file that contains the text `The quick brown fox`, a second file we'll pretend stores audio data, and a third file containing the (very obviously true) text `CS334 is the best!`

Usually, a file is a contiguous[121] sequence of bytes stored in the disk's array. A program in your computer's operating system, called the *filesystem*, usually decides where to place it.[122] A filesystem is an abstraction that provides a mapping from file *names* to disk locations. For example, I might have the following mapping for our hypothetical disk.

121 Contiguous means "right next to each other."

122 In practice, it is a little more complicated than this, as many modern disks themselves make data placement decisions.

Filename	Location	Size
`essay.txt`	22	19
`song.mp3`	35	6
`best_class.docx`	70	18

Table 4: A simple mapping from filenames to disk locations.

This mapping is also stored somewhere on disk (like at offset 0) so that when the computer starts up, the operating system can automatically fetch the name mappings. The above scheme is, of course, a simplification. Disks are vastly larger than the one shown in my diagram, files are usually much larger as well, and filesystems themselves have a lot more nuance in how they manage data. For example, we typically want to organize information *hierarchically*, in a recursive, tree-like structure. The basic principle remains the same, except that a special file called a *directory* makes hierarchies possible. A directory[123] file stores a mapping that says where one can recursively find the files "inside" the directory.

123 A directory is usually called a *folder* by computer users, but programmers tend to use the technical term because it is more precise.

It's worth mentioning that the cheapest storage, on a per-byte basis, is magnetic storage. *Magnetic tape* stores data on what looks like an audio cassette tape 26. A *hard disk drive* (HDD) stores data on spinning platters 27. SSDs are typically 3-4× the cost of tape or HDDs for the same amount of storage. Magnetic storage devices are machines with moving parts, and so they are sensitive to vibration and drops, which is why we rarely see them anymore in consumer devices. However, in protected environments, they can be both cheaper and more reliable than SSDs. Hard disks are an excellent medium to store backups of your most important files.[124]

[124] Thumb drives are SSDs. SSDs are a *terrible* medium for backups, because the physical property they use for data storage degrades over time. If you put data on a thumb drive, stick it in a desk drawer, and then expect the data to still be there, intact, several years later, you may be in for a shock.

Tapes and hard disks encode the "big array" of data differently than SSDs. Filesystems mostly abstract over those differences so that you can write programs the same way regardless of the medium on which you store your data.

Figure 26: In 2025, an LTO-9 tape costs roughly $0.005 per billion bytes.

Figure 27: In 2025, a hard disk drive costs roughly $0.014 per billion bytes.

Files and the Shell

Let's open up a terminal and examine some files on your computer. Hopefully you have used a terminal before, but it you have not, a *terminal* is a text-only interface for your computer. Expert users tend to prefer working in a terminal over other interfaces because they can work more efficiently by keeping their hands on a keyboard. Some expert users go to great lengths to avoid using a touchpad or a mouse. For the most part, any computing task you can do with a mouse you can do faster in the terminal once you have gotten over the initial learning curve.

In the macOS, start a program called `Terminal`. If you are using Linux or some other UNIX-like operating system your terminal program may be called `Terminal`, `Console`, `Konsole`, `xterm`, or some other variation. Microsoft Windows also has a console, called the `Windows Terminal`, but by default it is configured to run `cmd.exe` or `PowerShell`, and both of these are very different environments than we use in our CS program. If you're using Windows, I recommend installing the Windows Subsystem for Linux (WSL). Then start the `Windows Terminal` and select your Linux distribution (typically `Ubuntu`).[125]

[125] Download WSL at https://learn.microsoft.com/en-us/windows/wsl/install

After starting a terminal, you should see an interactive program called the *shell*. You may hear people refer to the shell as the "command line"; most of the time these terms refer to the same thing. The first thing to know about the shell is that it is a programming language. Everything you can do in Python or Java you can also do in the shell. However, the shell is designed to make certain tasks easy at the expense of others, so although it is *possible* to write many programs in the shell, we tend to use general-purpose programming languages when writing complex programs.

A shell usually displays a *prompt* and waits for you to supply a *command*. The prompt on my computer as I write this looks like the following.

```
dbarowy@Tash textbook %
```

Your prompt may look different than mine, because the prompt's appearance can vary depending on a user's settings. To avoid confusion when discussing the shell we adopt a convention whereby the shell prompt is shown as a `$` symbol:

```
$
```

For simplicity, I will also use `$` throughout this reading.

Location

The second thing to know about the shell is that at any given time, your interactive session has a "location" in the filesystem. Typically, after opening the `Terminal`, that location is your *home directory*, which is a kind of "default location" for your account on your computer. We can find the location for our shell by typing the following.

```
$ pwd
/Users/dbarowy/Documents/Code/cs334-materials/textbook
```

(do not type the `$` sign)

The `pwd` command means "print working directory." As I write this, I am working in the directory for the course textbook, so the *path* that is returned for me is a subdirectory of my home directory. Try typing `pwd` yourself.

Listing Files

I can find out which files are in the working directory by typing `ls`, which is short for "list files."

```
$ ls
Makefile              graphics              supporting_code
Makefile.aux          handouts              todelete
Makefile.fdb_latexmk  macros.tex            tufte-book.cls
Makefile.fls          main.bbl              tufte-common.def
Makefile.log          main.tex              tuftefoot.sty
bibliography.bib      readings
file-offset.sh        sources
```

As you can see, I have a lot of files in my working directory. Some of these names are subdirectories, not files. By default, `ls` does not distinguish between the two, but you can add a "flag" to the `ls` command and it will give you more information.

```
$ ls -l
total 280
-rwxr-xr-x@  1 dbarowy  staff    646 Jan 27 13:19 Makefile
-rw-r--r--@  1 dbarowy  staff     32 Jan 27 13:20 Makefile.aux
-rw-r--r--@  1 dbarowy  staff    615 Jan 27 13:20 Makefile.fdb_latexmk
-rw-r--r--@  1 dbarowy  staff    261 Jan 27 13:20 Makefile.fls
-rw-r--r--@  1 dbarowy  staff  14343 Jan 27 13:20 Makefile.log
-rwxr-xr-x@  1 dbarowy  staff   1620 Jan 28 15:19 bibliography.bib
-rwxr-xr-x   1 dbarowy  staff    553 Jan 30 16:42 file-offset.sh
drwxr-xr-x  63 dbarowy  staff   2016 Jan 30 14:46 graphics
drwxr-xr-x   5 dbarowy  staff    160 Sep  9  2022 handouts
-rwxr-xr-x@  1 dbarowy  staff      0 Mar 28  2021 macros.tex
-rw-r--r--@  1 dbarowy  staff    443 Jan 30 16:55 main.bbl
-rwxr-xr-x@  1 dbarowy  staff  19123 Jan 30 15:26 main.tex
drwxr-xr-x  46 dbarowy  staff   1472 Jan 31 13:29 readings
drwxr-xr-x   8 dbarowy  staff    256 Jan 23  2024 sources
drwxr-xr-x  14 dbarowy  staff    448 Aug 16  2023 supporting_code
drwxr-xr-x   4 dbarowy  staff    128 Feb 26  2024 todelete
-rwxr-xr-x@  1 dbarowy  staff   2113 Mar 28  2021 tufte-book.cls
-rwxr-xr-x   1 dbarowy  staff  66821 Mar 28  2021 tufte-common.def
-rwxr-xr-x@  1 dbarowy  staff   2691 Mar 28  2021 tuftefoot.sty
```

There's a lot more information here, so I'm not going to discuss all of it. The most important thing to know is that the first letter in the first column tells you what kind of file the name represents. For example, for the name `graphics`, the column containing `drwxr-xr-x` starts with a `d`, so it is a directory. For the name `Makefile`, the column shows `-rwxr-xr-x@`, and the starting character `-` means "regular file." There's a lot of information packed in here, mostly about who is allowed to read or modify files.[126]

[126] If you're curious, Google "`unix permissions bits`" or type `man ls` to access the *manual page* for the `ls` command.

You can also use the `file` command to get the same information in a more verbose form.

```
$ file graphics
graphics: directory
$ file Makefile
Makefile: makefile script text, ASCII text
```

For any file containing "ASCII text", I can use the `cat` command to print it out.

Viewing Files

```
$ cat Makefile
all: main.pdf

main.pdf: main.tex $(wildcard readings/*.tex)
	latexmk -xelatex main.tex

handouts: handout-intro_to_c.pdf handout-memory_management.pdf handout-passing_pointers.pdf

handout-intro_to_c.pdf: readings/intro_to_c.tex handouts/handout-intro_to_c.tex
	latexmk -xelatex handouts/handout-intro_to_c.tex

handout-memory_management.pdf: readings/memory_management.tex handouts/handout-memory_management.tex
	latexmk -xelatex handouts/handout-memory_management.tex

handout-passing_pointers.pdf: readings/passing_pointers.tex handouts/handout-passing_pointers.tex
	latexmk -xelatex handouts/handout-passing_pointers.tex

clean:
	latexmk -C
```

Suppose that `Makefile` was too big to fit on my screen. `cat` will print the entire file, which would be a pain to view without scrolling. Instead, I can *pipe* the output to a *pager* program that makes it easier to view the output.

```
$ cat Makefile | less
```

To quit the `less` program type `q`. You can also use the up and down arrows or the `Page Up`/`Page Down` keys to navigate. Experienced shell users are already screaming at me because they know that you can use `less` directly with a file, like so:

```
$ less Makefile
```

However, I show you the `cat Makefile | less` version above because it is a nice recipe that you can apply whenever you have *any* program that produces lots of output. The recipe is

```
$ program | less
```

For example, take something big and complicated like,

```
$ find . -iname "*.tex" -print -exec grep -iH "program" {} \; | less
```

I don't expect you to know the command above. But in case you're curious, it finds files with names ending in `.tex`, then searches inside them for the string `program`, and then prints the line containing each match out. Since I'm searching in the directory that has the course textbook, `program` shows up a lot, so I pipe the output to the `less` pager.

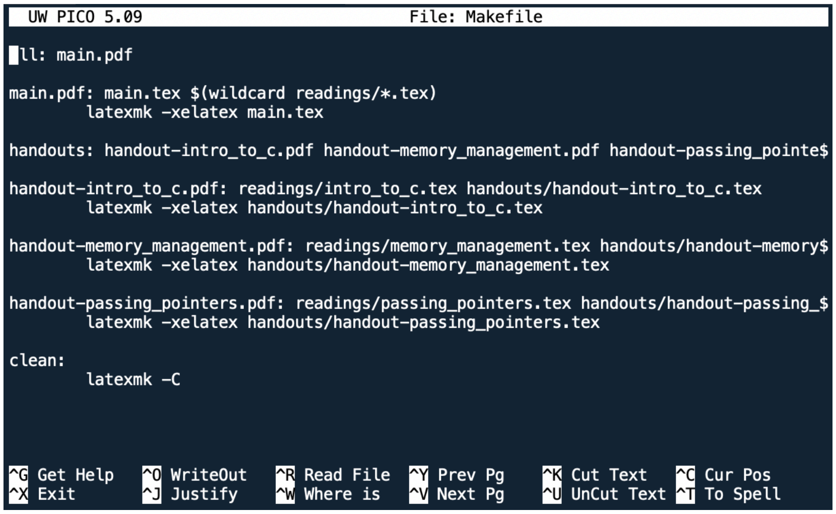
Editing Files

If your file contains ordinary text data, you can change the contents of a file using a `text editor`. A text editor is an interactive program that lets you easily modify the contents of a file. Some common text editors are `TextEdit` on the macOS, `Notepad` on Windows, or `gedit` on Linux. There are hundreds of text editors available. Many programmers like to use a variant of a text editor called an *integrated development environment* (IDE) that has productivity-enhancing features specifically for programming tasks. Some widely-used IDEs are `Visual Studio Code`, `XCode`, and `Eclipse`, and there are dozens, if not hundreds of IDEs as well. For this section, we will stick to a simple text editor for the terminal called `nano`.

If you type `nano` *`filename`*, the `nano` program will open a file called *`filename`* or create it if it does not exist. For example, I will open the `Makefile` in my working directory.

```
$ nano Makefile
```

Because `Makefile` already exists, I will see something like this:



You can navigate this file using the arrow keys, and typing on the keyboard will directly insert characters in the file. The bottom two rows show shortcuts for a set of available commands. You can run a command by pressing ctrl + *key*. For example, the shortcut to save a file is `^O` `WriteOut`, and what this means is that we should press ctrl + o to save. The shortcut `^X` `Exit` tells us that we can press ctrl + x to quit. Be sure to pay attention to the interactive prompt that appears at the bottom of the screen. Go ahead and quit to return to the shell.

Changing Locations

Finally, we can *change directory* to another location using `cd`. For example, we know that `graphics` is a directory, so I should be able to "go" there.

```
$ cd graphics
```

This is a good time to mention that, in UNIX, the convention is that if you don't explicitly ask for output, but the command succeeds, nothing is printed out. So it is probably not obvious that I just changed into the `graphics` directory. But we can check with a command we already learned, right?

```
$ pwd
/Users/dbarowy/Documents/Code/cs334-materials/textbook/graphics
```

Optional: Peeking Behind the Abstractions

When you program in a computer, you're usually seeing data through many abstractions. Those abstractions are there to help you, but sometimes they can get in the way. To finish off the tutorial, I thought I'd show you a little more to help you "see through" the abstractions provided by the filesystem. To be clear, this section is optional reading. However, since you presumably opted into studying computer science willingly, why not take the time to read on? The information in this section is interesting and will serve you well in the future.

Let us pose the following question: if files really are just big arrays of bytes on a physical disk, can we see those bytes? The answer is yes, definitely.

We will start by determining the names of the physical disk devices in my computer.

```
$ mount
/dev/disk3s1s1 on / (apfs, sealed, local, read-only, journaled)
devfs on /dev (devfs, local, nobrowse)
/dev/disk3s6 on /System/Volumes/VM (apfs, local, noexec, journaled, noatime, nobrowse)
/dev/disk3s2 on /System/Volumes/Preboot (apfs, local, journaled, nobrowse)
/dev/disk3s4 on /System/Volumes/Update (apfs, local, journaled, nobrowse)
/dev/disk1s2 on /System/Volumes/xarts (apfs, local, noexec, journaled, noatime, nobrowse)
/dev/disk1s1 on /System/Volumes/iSCPreboot (apfs, local, journaled, nobrowse)
/dev/disk1s3 on /System/Volumes/Hardware (apfs, local, journaled, nobrowse)
/dev/disk3s5 on /System/Volumes/Data (apfs, local, journaled, nobrowse, protect, root data)
map auto_home on /System/Volumes/Data/home (autofs, automounted, nobrowse)
```

Each unique prefix `/dev/disk`i is a distinct physical disk. You can see above there are two unique prefixes, so I have two physical disks in my computer, `/dev/disk1` and `/dev/disk3`. The next part of the name, `s`j, denotes the *logical disk*. A logical disk is an abstraction that lets me treat a *slice* of the array on the disk as if it were its own physical disk. For example, `/dev/disk3s5` says that I should treat the fifth slice on disk three as its own physical disk. It also says that the `/dev/disk3s5` logical disk should appear in the filesystem as `/System/Volumes/Data`.

And hey, check it out! I can use `ls` to see what's in there:

```
$ ls /System/Volumes/Data
Applications            Volumes                 private
Library                 cores                   sw
MobileSoftwareUpdate    home                    usr
System                  mnt
Users                   opt
```

So far, we have used `cat`, `less`, and `nano` to examine files. These programs strongly assume that the data stored in a file is textual. But files can contain any kind of data. The data could be image data, or video data, or audio data, or whatever. As mentioned in the preface, one of the astounding facts about computers is that they don't actually know about any of these data types. In reality, *all data* is stored as an array of numbers on a computer. When you run `cat` or `less`, that numeric data is *interpreted* as text data. We can actually look at the raw numeric data if we want.

```
$ hexdump -C graphics/sciam-1973.png | less
00000000  89 50 4e 47 0d 0a 1a 0a  00 00 00 0d 49 48 44 52  |.PNG........IHDR|
00000010  00 00 04 c6 00 00 06 94  08 06 00 00 00 1b d4 10  |................|
00000020  19 00 00 01 56 69 43 43  50 49 43 43 20 50 72 6f  |....ViCCPICC Pro|
00000030  66 69 6c 65 00 00 28 91  6d 90 c1 4a 02 61 14 85  |file..(.m..J.a..|
00000040  8f 65 28 66 60 e1 2a 0a  84 20 24 54 c2 14 da aa  |.e(f`.*.. $T....|
00000050  81 45 2e 26 35 a8 68 d1  38 9a 06 3a fd fd 4e 84  |.E.&5.h.8..:..N.|
00000060  d0 43 b4 8f 76 6d a2 17  08 57 ed da b4 0b 8a a0  |.C..vm...W......|
00000070  1e a0 65 04 22 94 4c 67  9c 4a ad 2e 5c ee c7 e1  |..e.".Lg.J..\...|
00000080  dc cb e5 00 43 6e 55 88  aa 13 40 4d 37 64 36 9d  |....CnU...@M7d6.|
00000090  0c 6c 6c 6e 05 5c 2f 70  c0 03 37 c2 08 aa 5a 5d  |.lln.\/p..7...Z]|
000000a0  24 14 25 43 0b be e7 60  b5 ee e9 66 dd 85 ad 5b  |$.%C...`...f...[|
000000b0  73 33 c9 cb f1 c9 e7 8b  d7 5a 46 cf b5 7d e7 7f  |s3.......ZF..}..|
000000c0  fd 03 e5 29 96 ea 1a e7  07 3b a2 09 69 00 8e 10  |...).....;..i...|
000000d0  59 39 32 84 c5 c7 64 bf  e4 53 e4 13 8b cb 36 5b  |Y92...d..S....6[|
000000e0  77 fd 05 9b af ba 9e 7c  36 45 be 25 fb b4 8a 5a  |w......|6E.%...Z|
000000f0  24 3f 91 43 85 3e bd dc  c7 b5 ea a1 f6 f5 83 f5  |$?.C.>..........|
00000100  bd b7 a4 af e7 38 27 d8  53 c8 20 8d 00 96 b1 84  |.....8'.S. .....|
...
```

The `hexdump` program prints each byte in the file as a two-character hexadecimal number. The column on the left shows the location (aka "offset") within the file where the leftmost byte starts. The output also shows the interpretation of the bytes as text on the right in case that ends up being meaningful. For this file, it doesn't mean anything, because the file contains only image data. But were I to run `hexdump` on a text file, I would see the text on the right along with the corresponding bytes.

If you really want to see the binary data, we can do that too, although we rarely do this because it's usually better to work with a hexadecimal or textual interpretation of the data.

```
$ xxd -b graphics/sciam-1973.png | less
00000000: 10001001 01010000 01001110 01000111 00001101 00001010  .PNG..
00000006: 00011010 00001010 00000000 00000000 00000000 00001101  ......
0000000c: 01001001 01001000 01000100 01010010 00000000 00000000  IHDR..
00000012: 00000100 11000110 00000000 00000000 00000110 10010100  ......
00000018: 00001000 00000110 00000000 00000000 00000000 00011011  ......
0000001e: 11010100 00010000 00011001 00000000 00000000 00000001  ......
00000024: 01010110 01101001 01000011 01000011 01010000 01001001  ViCCPI
0000002a: 01000011 01000011 00100000 01010000 01110010 01101111  CC Pro
...
```

With a bit more digging, you can even find the offset in the logical disk where a file starts. I will leave this as an exercise to the reader.[127]

[127] Another good exercise for the reader is why I said "offset for the logical disk" and not "offset for the physical disk."

Summary

There's a lot more to know about files, filesystems, and disks. If you know everything in this chapter, you will be well positioned to understand everything you need to know for this course (and for many other CS courses).