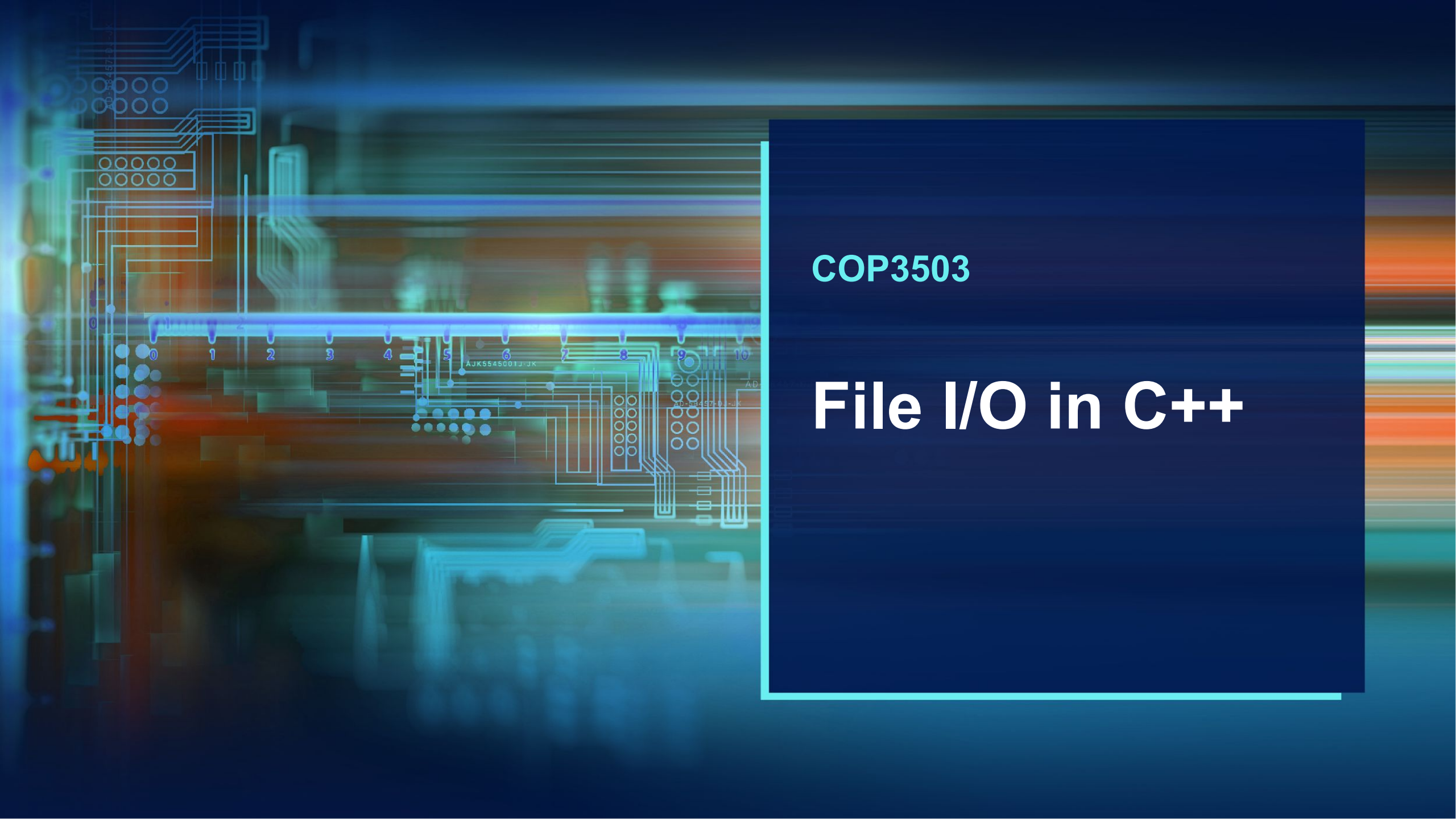


COP3503

# File I/O in C++

# File I/O Classes

- In C++, 3 classes will help with file input/output operations:
  - `ofstream` Output file stream, for writing data to a file
  - `ifstream` Input file stream, for reading data from a file
  - `fstream` File stream (Input OR output [or both!] depending on what you need)

```
#include <fstream>    // All 3 file stream classes are in here
using std::ofstream;
using std::ifstream;
using std::fstream;
```

# File I/O 101

1. Open the file.
2. Verify the file was opened. — Optional, but do it anyways.
3. Use the file.
   - **Read from it**: Copy data from the file into your program.
   - **Write to it**: Copy data from your program to the file.
   - If you're reading from the file, the data is now "in" your program—do whatever you want with it
4. Close the file. — Optional
   - You aren't **required** to close them in C++, but it **might** be necessary in some cases.

# Creating Output Files with ofstream

```
#include <fstream>
using std::ofstream;  // Or using namespace std; etc..

// Elsewhere...
// Pass a string to its constructor to open a file
ofstream someFile("SomeFile.txt");

// Or use the open() function
// The constructor just calls open() if you pass it a string
ofstream otherFile;
otherFile.open("data/file1.xyz");

// Verify the file is open...

// If so, do some stuff with said file
```

# Check to See if Files Are Opened!

- Use the `is_open()` function.

```
ofstream someFile("data/ImportantFile.txt");

// Check if the file is actually open...
if (someFile.is_open())
{
    cout << "File opened successfully!" << endl;
    // Do something with the file
}
else
{
    cout << "Error! File not opened!" << endl;
    // Throw an exception, return an error code, etc
}
```

A simple verification before any other work can save you a lot of time!

You might spend a lot of time debugging only to discover the file was never opened!

**Why** it didn't open… that's another matter!

# Why Might a File Fail to Open?

- **Most likely reason**: You got a file path wrong (wrong name, wrong directory, etc)
  - `ofstream` objects will create the file if it doesn't exist—a "wrong-but-still-valid" path might produce a file!
- You may be trying to access some protected directory.
- The file may be already opened by something else!
  - Another program or another object in **your** program.
  - Operating systems may not allow more than one "thing" to access a file at the same time.

```
// Typos -- if you tried to open "data/scores.txt"...
ofstream someFile("data/scors.txt");     // You may create the wrong file...
ofstream someFile2("daat/scores.txt");   // Cannot open, no "daat/" directory
```

```
// Multiple objects trying to access the same file
ofstream file1("data.txt");  // Opens! (Assuming a proper path)
ofstream file2("data.txt");  // Won't open, "data.txt" is already in use
```

# Opening Files With Specific IO Modes

- There are several values (stored in an enum) you can pass as a second parameter to open() or a constructor.
- They affect how the file is opened, and how read/write operations work.

| | |
|---|---|
| `ios_base::in` | Input (`ifstream` uses this automatically) |
| `ios_base::out` | Output (`ofstream` uses this automatically) |
| `ios_base::trunc` | Truncate (delete) the contents (`ofstream` uses this by default) |
| `ios_base::app` | Append, add to a file's contents |
| `ios_base::ate` | Seek to the end after opening (**A**t **T**he **E**nd) |
| `ios_base::binary` | Perform operations in binary mode |

```
fstream someFile("file.ext", ios_base::out | ios_base::binary | ios_base::app);
```

Combine multiple options using the bitwise **or** operator – open this file in output mode, binary mode, **and** preserve the file contents.

# Writing to Files With ofstream

- Use the **insertion operator**: <<
- Just like printing something to the screen with **cout**

```
ofstream someFile("ExampleFile.txt");

// Check if the file is actually open…
if (someFile.is_open())
{
    // Write some data (just like using cout)
    const float mmmPI = 3.1415f;
    someFile << "Hello World, file edition.";
    someFile << "Value of pi: " << mmmPI;
}
```

Contents of “ExampleFile.txt”

```
Hello World, file edition.Value of pi: 3.1415
```

Space or a newline might be helpful here…

You can write tabs (with **\t**) or newline characters (with **endl** or **\n**) to format the file.

# Writing to Files With ofstream

```
ofstream someFile("Numbers.txt");

// Check if the file is actually open...
if (someFile.is_open())
{
    for (int i = 2; i <= 16; i *=2)
        someFile << i << endl;

    // Close the file when finished
    // You MUST close it if you want to open another file,
    // using the SAME stream object
    someFile.close();
}

someFile.open("Numbers2.txt");
for (int i = 2; i <= 16; i *=2)
    someFile << i << ' ';
```

Contents of “Numbers.txt”

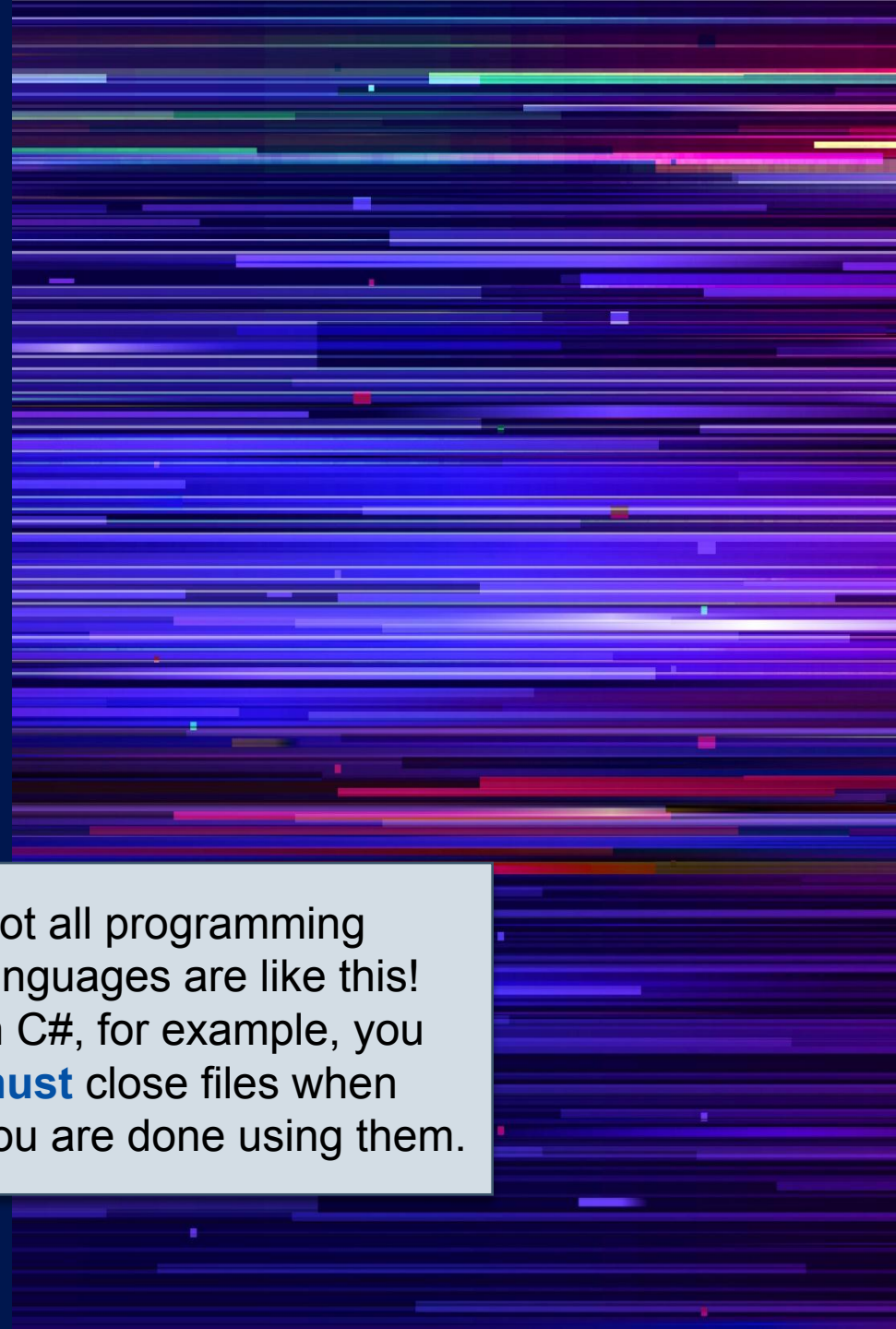
```
2
4
8
16
```

Contents of “Numbers2.txt”

```
2 4 8 16
```

# Do You Ever Need to Call `close()`?

- For the most part, no—the **destructors** of file stream objects call close automatically for you
- This is part of a concept called **Resource Acquisition Is Initialization**, or **RAII**.
  - When an object is initialized, it handles the acquisition of some resource (a file, in this case)
  - When that object is destroyed, it handles freeing that resource (i.e calling close())
  - As long as the object is around, the resource is valid
- If you do call `close()`? Nothing bad will happen



Not all programming languages are like this! In C#, for example, you **must** close files when you are done using them.

# Writing Complex Objects

- You would write them the same as you would print them to the screen.

```
class Vehicle
{
    string  _make;
    string  _model;
    int     _miles;
    double  _price;
public:
    // Accessors
    string GetMake();
    string GetModel();
    int GetMiles();
    double GetPrice();
};
```

```
ofstream someFile("VehicleData.txt");
Vehicle car; // Assume initialization...

// Check if the file is actually open…
if (someFile.is_open())
{
    someFile << car.GetMake() << endl;
    someFile << car.GetModel() << endl;
    someFile << car.GetMiles() << endl;
    someFile << car.GetPrice() << endl;
}
```

How you create an output file depends on your data—there's no one standard.

Contents of "VehicleData.txt"

```
Ford
Mustang
100
27350.72
```

# Reading Files With `ifstream`


- Reading data can be slightly more complicated.
- How you read depends on the format of the data.
  - Just like using **cin** and **getline()**

Contents of “VehicleData.txt”

```
Ford
Mustang
100
27350.72
```

If you wanted to read this as user input, how would you do that?

**What kind of data?**
Make (string)
Model (string)
Mileage (int)
Price (double)

```
string make;
string model;
int miles;
double price;
```

```
getline(cin, make);
getline(cin, model);
cin >> miles;
cin >> price;
```

Reading from a file just changes `cin` to some `ifstream` object.

# Reading Files With `ifstream`

```
ifstream someFile("VehicleData.txt");

string make;
string model;
int miles;
double price;

// Check if the file is actually open...
if (someFile.is_open())
{
    // Read data using the ifstream object
    getline(someFile, make);
    getline(someFile, model);
    someFile >> miles;
    someFile >> price;
}
```

Once the data has been read from the file, it's in your program (wherever you stored it).

After that, it's up to you to do whatever you want with the data.

# Assembling Complex Objects From a File

- We don't read "objects" from a file.
  - We read small pieces of data, one at a time.
- **To "read an object" from a file**:
  Read all the parts, then construct an object.

```
// Check if the file is actually open…
if (someFile.is_open())
{
    // Read data using the ifstream object
    getline(someFile, make);
    getline(someFile, model);
    someFile >> miles;
    someFile >> price;

    // Construct an object from individual pieces
    Vehicle car(make, model, miles, price);

    // Do something with the car object, store it for later?
    someVectorOfVehicles.push_back(car);
}
```

# Reading Unknown Quantities of Data

- We don't always know how much data is in a file.
- We could write code that "reads as long as there is data".
  - This is good for simple data – one value from a file is one "object" in a program

```
ifstream inFile("Numbers.txt");
int numberFromFile;
vector<int> numbers;

if (inFile.is_open())
{
    // While a value was successfully read...
    while (inFile >> numberFromFile)
    {
        // Store the value for later use
        numbers.push_back(numberFromFile);
    }
    // File finished reading, do something with the data
}
```

Contents of "Numbers.txt"

```
10
20
30
50
100
```

5 numbers in a file or 5,000, this loop can work no matter the count.

# Some Files Contain Data About the Data

- Some files might contain quantity values to describe the contents of a file.
- We can read and use this data to help read other data.

Contents of “Numbers.txt”

```
5
10
20
30
50
100
```

The first value is a count for the “real” data in the file.

Contents of “VehicleData.txt”

```
2
Ford
Mustang
100
27350.72
Mazda
CX-5
1602
24993.85
```

```
ifstream inFile("SomeFile.txt");
int numberCount;
inFile >> numberCount; // Read the count first

// Loop according to the count
for (int i = 0; i < numberCount; i++)
{
     // Read the one "thing" from the file,
     // according to its complexity
}
```

**You will often encounter this with files**:

1. Read some count/quantity
2. Read that quantity of

# There Are Lots of Ways to Structure Files

Homogeneous lists

```
141
2834
2354
718
59981
67120
563
2344
17
```

One line, one value

Interleaved data types on separate lines

```
2
Ford
Mustang
14231
28147.39
Tesla
Model 3
79
46212.41
```

Multiple lines
to make one object

Data on the same line, separated by a **delimiter**

```
Ford Mustang 14231 28147.39
Dodge Neon 23429 12212.41
```

```
Ford,Mustang,14231,28147.39
Dodge,Neon,23429,12212.41
```

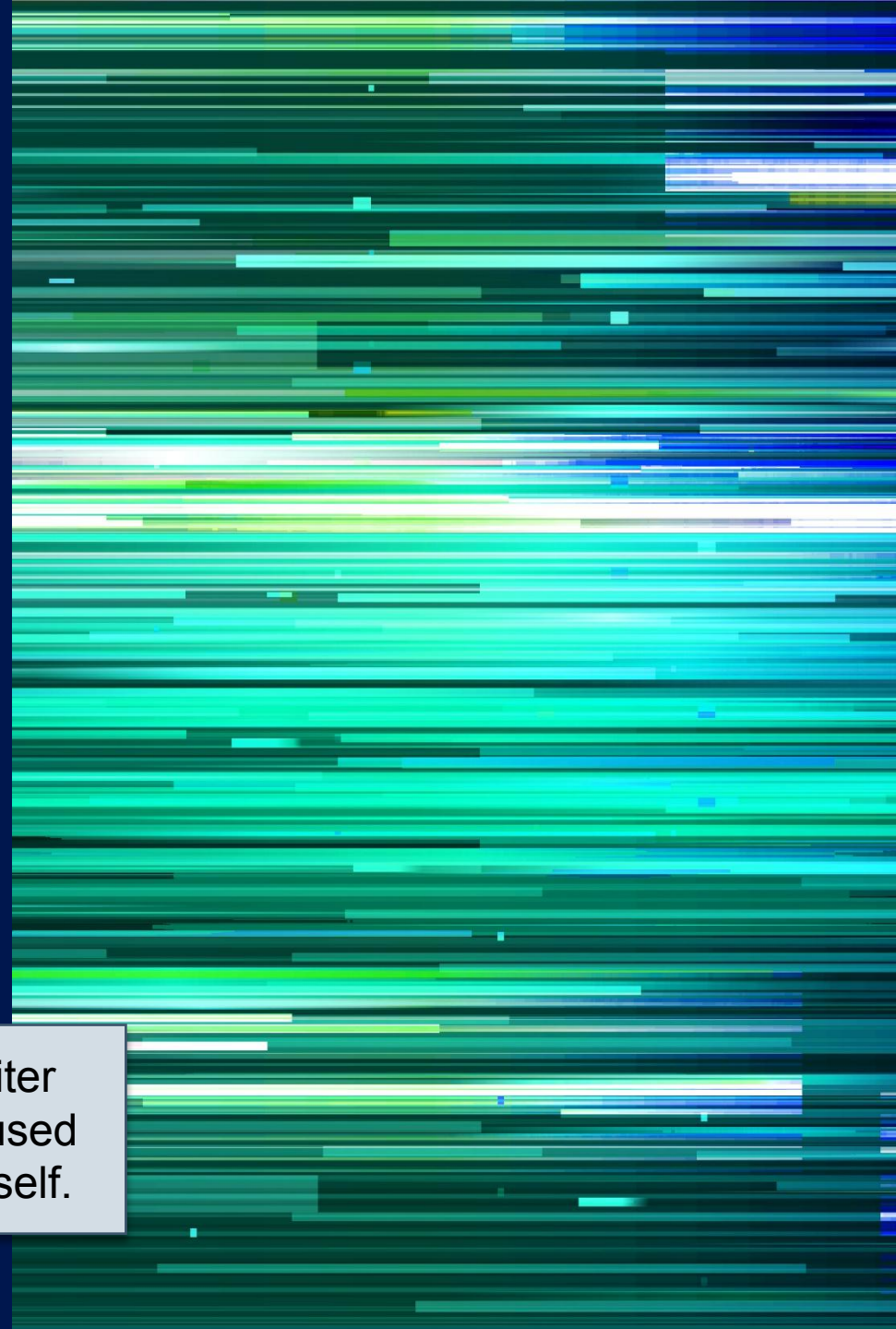
```
Ford|Mustang|14231|28147.39
Dodge|Neon|23429|12212.41
```

One line, one object…
but in multiple pieces

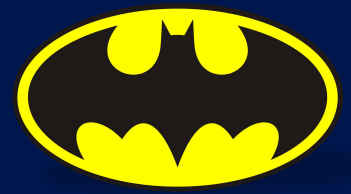
# Delimiters

- A character used to indicate some **stopping point** or **boundary** between parts of data
- Used to break data into separate **tokens**
- Can be anything you want—
  some may be better choices than others
- Comma delimiters
  - Spaces
  - Commas
  - Pipe symbol / Vertical bar:

This is a good delimiter because it is rarely used as part of the data itself.



# Delimiters Can Be Anything You Want



```
Batman,Robin,Batgirl,Joker,Two-Face,Killer Croc
```

**Commas** and **pipes** are pretty common.

```
Batman|Robin|Batgirl|Joker|Two-Face|Killer Croc
```

```
Batman$Robin$Batgirl$Joker$Two-Face$Killer Croc
```

```
Batman%Robin%Batgirl%Joker%Two-Face%Killer Croc
```

**$** and **%** work just fine, though it may be a little tough to read.

```
BatmanQRobinQBatgirlQJokerQTwo-FaceQKiller Croc
```

**Q** (and other letters) might be a bad idea.

```
Batman Robin Batgirl Joker Two-Face Killer Croc
```

**Spaces** aren't always a good idea.

Now Batman has to fight "Killer" **and** "Croc".

```
oker Two-Face "Killer Croc"
```

You could use double-quotes for these cases—but you would need specific code to handle it.

# Comma-Separated Values (CSV) File

- A CSV file is a very common text-based format to store information.
- The information is separated by using a comma as a **delimiter**.
- CSV is often used as a “generic” format for spreadsheets.
- Stores only the “real” data, no formatting or program-specific data

Some data in a text editor, as a .CSV file

```
1  Ford,Mustang,14231,28147.39
2  Dodge,Neon,23429,12212.41
```

**One line in a file**: one row in a spreadsheet
**Each token (data separated by a delimiter)**: a column

Data in Excel

| | A | B | C | D |
|---|---|---|---|---|
| 1 | Ford | Mustang | 14231 | 28147.39 |
| 2 | Dodge | Neon | 23429 | 12212.41 |

Many spreadsheet programs will read .CSV data and use commas to split data into cells.

# How to Read Multiple Values From One Line?

The data

```
1 Ford,Mustang,14231,28147.39
2 Dodge,Neon,23429,12212.41
```

**First**:
Read a line from the file, using `getline()`.

```
ifstream someFile("Vehicles.csv");
string singleLine;
getline(someFile, singleLine);
```

```
singleLine == "Ford,Mustang,14231,28147.39"
```

**Second**:
Break the string into multiple parts (**tokenize**, break into **tokens**) based on the delimiter.

How do we find those delimiters?

**The Hard Way**:
Lots of calls to the `find_first/last_of()` and `substr()` functions of `std::string`.

**The Easier Way**:
Use a class called `istringstream` and the `getline()` function.

# Using istringstream

```
#include <sstream>
using std::istringstream;
```

```
ifstream someFile("Vehicles.csv");
string singleLine;
getline(someFile, singleLine);

// Convert a string to a "stream" of data
// Essentially turn a string into a file we can "read" with getline()
istringstream stream(singleLine);

string token;
// Read the stream until you reach a comma
getline(stream, token, ',');
```

The `getline()` function can be passed a third, optional parameter – a delimiter to search for.

Data gets read up to the delimiter, then removed from the stream (and so is the delimiter).

We can read the contents in a loop, or one at a time.

Reading tokens individually may be necessary to handle different data types.

| Stream Contents Initially |
| --- |
| `"Ford,Mustang,14231,28147.39"` |

| After getline() with ‘,’ as a delimiter |
| --- |
| `"Mustang,14231,28147.39"` |

# Using `istringstream` With a Loop

```
istringstream stream(singleLine);

string token;
// Keep tokenizing (based on commas)
until
// there is no more data
while (getline(stream, token, ','))
{
    cout << "Token: " << token << endl;
}
```

- This gets us tokens as **`std::strings`**
- What if we needed them as numbers?
  We would have to convert to another format.

**Stream Contents**

`"Ford,Mustang,14231,28147.39"`

After **`getline()`** with ',' as a delimiter

`"Mustang,14231,28147.39"`

After calling **`getline()`** again…

`stream == "14231,28147.39"`

One more call…

`stream == "28147.39"`

After the fourth (and last) time…

`stream == ""`

**Output**

```
Ford
Mustang
14231
28147.39
```

```
getline(stream, token, ','); // Get the token
int miles = stoi(token); // Convert to a format we need
```

The **`while()`** condition fails after that—no more data!

# Converting Tokens to Usable Values

```
1  Ford,Mustang,14231,28147.39
2  Dodge,Neon,23429,12212.41
```

If you know the format is:
string, string, int, double…

```
istringstream stream(singleLine); // Create a stream

string make;
string model;
int miles;
double price;

getline(stream, make, ','); // Read into "make" directly
getline(stream, model, ','); // Read into "model" directly

string token; // Store results of getline()
getline(stream, token, ',');
miles = stoi(token); // Convert a string to an integer

getline(stream, token, ',');
price = stod(token); // Convert a string to a double

// Do something with the values
```

All of this is just for the first line of the file.

Need more than one line? Use a loop!

# All Together Now!

While we successfully get a line from the file…

Break that line down into various pieces, according to the file format.

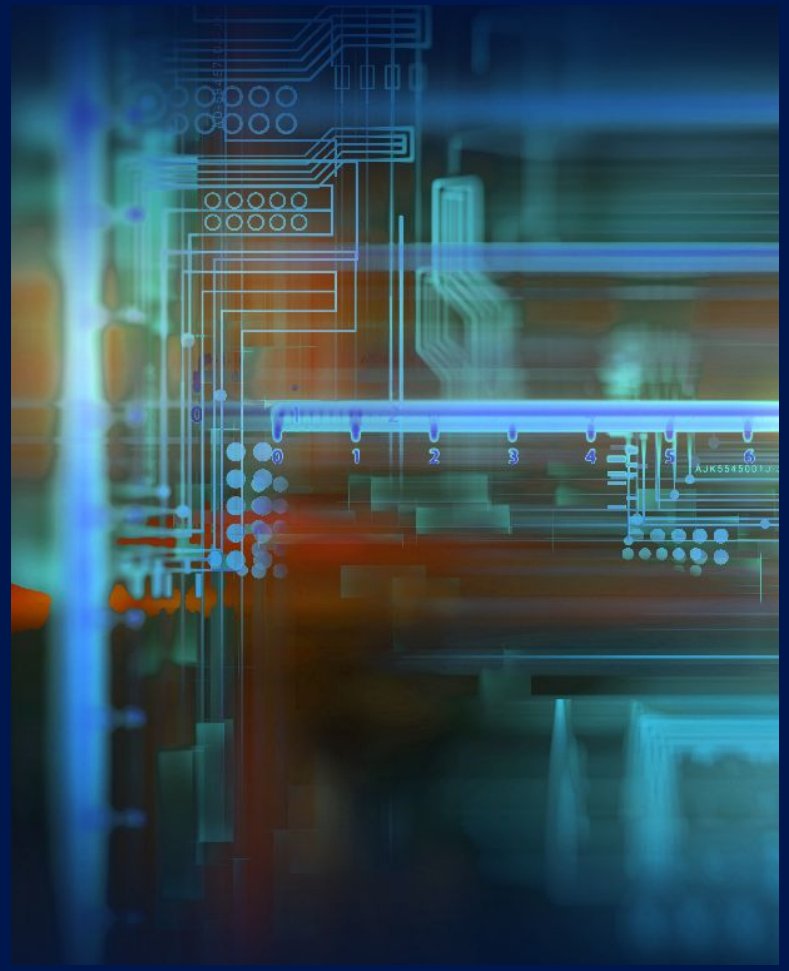
Use the data you've extracted, however you see fit.

```
ifstream someFile("Vehicles.csv");
string singleLine;
vector<Vehicle> vehicles;
while (getline(someFile, singleLine))
{
    string make;
    string model;
    int miles;
    double price;

    istringstream stream(singleLine);
    getline(stream, make, ',');
    getline(stream, model, ',');

    string token;
    getline(stream, token, ',');
    miles = stoi(token);
    getline(stream, token, ',');
    price = stod(token);

    Vehicle car(make, model, miles, price);
    vehicles.push_back(car);
}
```

# Recap

- C++, like any language, has (relatively) easy-to-use tools to work with files.
  - These tools copy data into your program or copy data out of it.
- You use these tools to open a file, work with it, and (optionally) close it.
- **All files have a structure**, and each must be handled according to that:
  - We can create our own formats if needed.
  - We may have to learn and work with existing formats (such as CSV).
- Files are often split up by **delimiters**, characters that separate **tokens** of data.
  - We have to **tokenize** the data into smaller pieces before we use it.

# Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

# Thank you for watching.