COP3503

Binary File I/O

Binary vs Text I/O Operations

- Text file I/O deal with conversions
  - Convert the data into text representations
  - Data gets processed into/out of a stream.

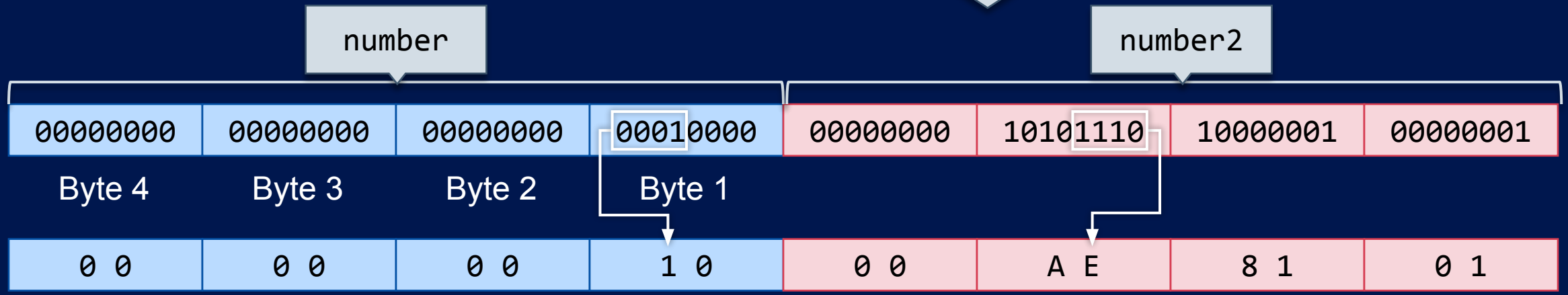
- Binary file I/O is a direct transfer of data from memory to a file, or vice-versa.
  - No conversion, just copying bytes
  - Faster transfer to files
  - Smaller file sizes (usually)

Binary Representation of Values

```
int number = 16;
int number2 = 98734;
```

Integers are commonly (but not always!) 4 byte data types (Each data type is made of some number of bytes).

Each **byte** is made of 8 **bits** (and each bit can be either 1 or 0).



Alternate representation: pairs of 4-bit values (still 8 bits)

Decimal versus hexadecimal versus binary: You don't need to know all of these conversions (for now)

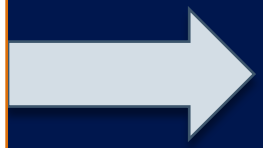
For binary file I/O, just know that everything is made of bytes—that's what we read and write from a file.

Text-based File I/O Involves Conversions

```
// Text-based output
ofstream file("someFile.txt");

int number = 16;
int number2 = 98734;
file << number << '\n';
file << number2;
```

These operations convert numbers to text representations and write those characters to the file.



Contents of "someFile.txt"

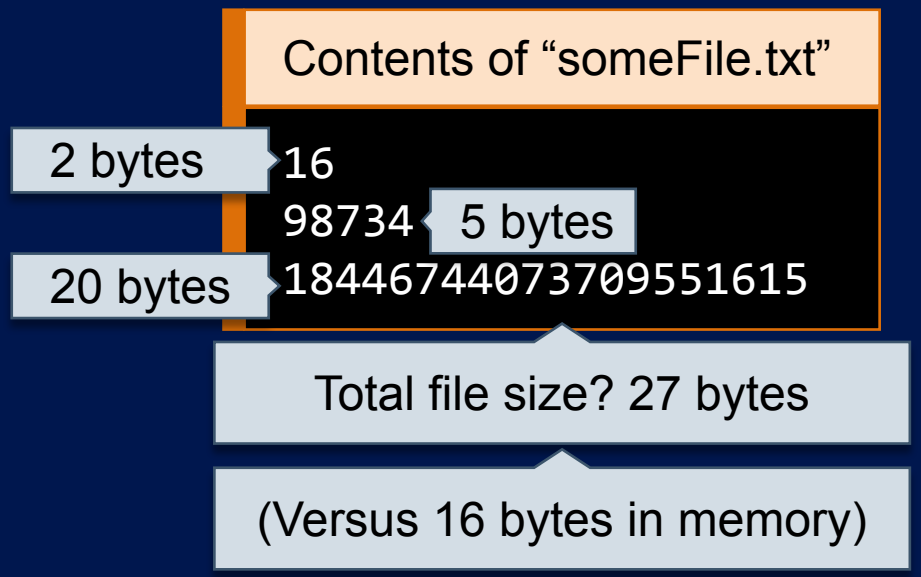
```
16
98734
```

This is not the number 16.
It's two characters: '1' and '6'.

This is not the number 98734.
This is five characters we perceive as that number.

Memory Compared to Text-Based Files

```
int number = 16;                    // 4 bytes in memory
int number2 = 98734;                // 4 bytes in memory
unsigned long long bigNumber;       // 8 bytes in memory
bigNumber = 18446744073709551615;
```



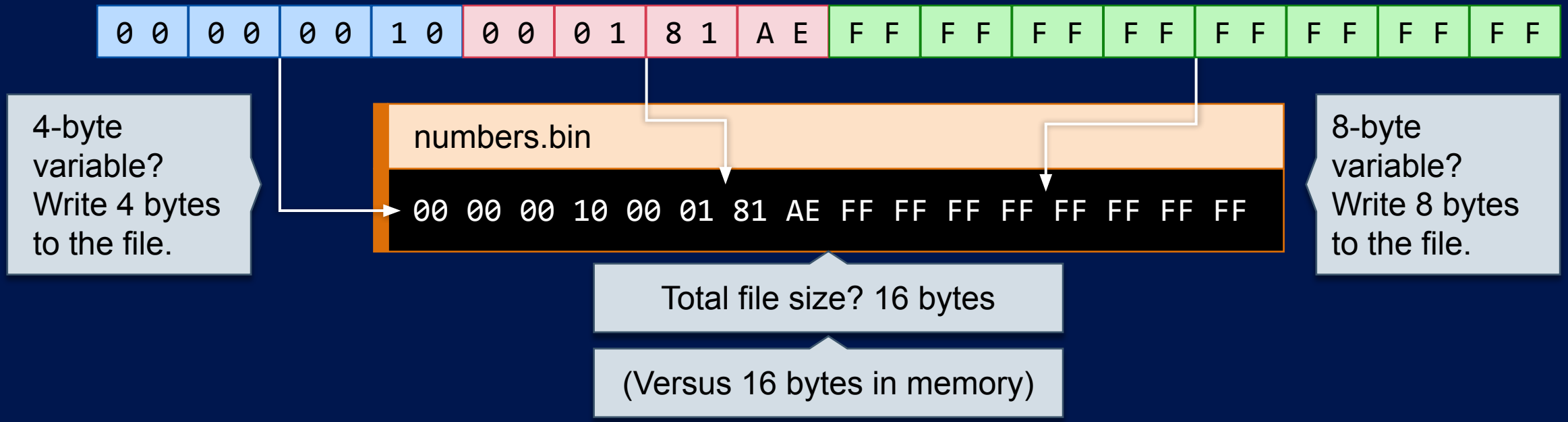
Characters are at least 1 byte per character.

There are many ways of encoding a file, each with different character sizes—we won’t cover those here.

Memory Compared to Binary Files

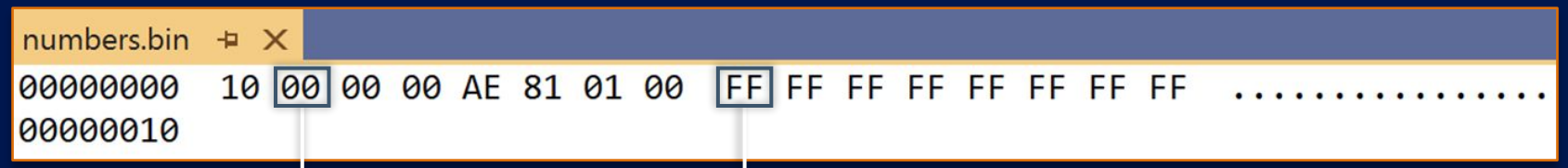
- In binary I/O, bytes are copied directly from memory to the file—we might call these the "raw bytes".

```
int number = 16;
int number2 = 98734;
unsigned long long bigNumber = 18446744073709551615;
```



What Does a Binary File Look Like?

- In Visual Studio



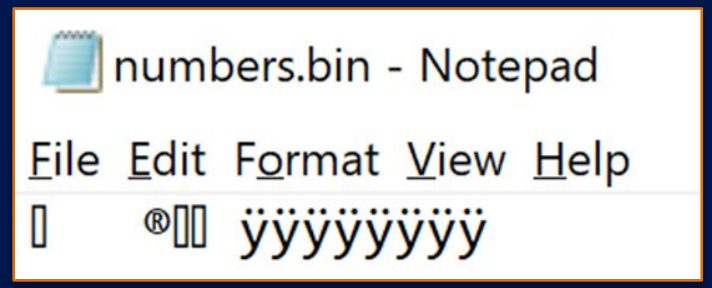
A value of `0` is the `NUL` character.

A value of `FF` (255) is `ÿ`.

- In Notepad++

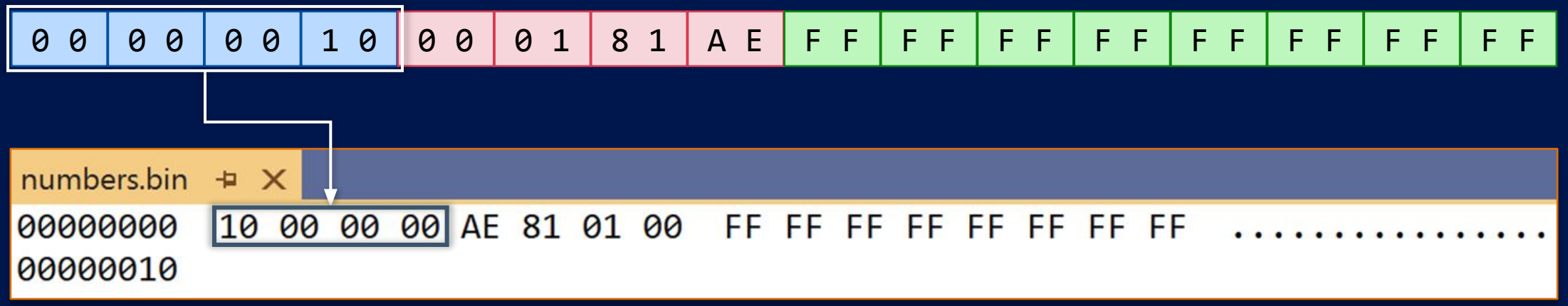
A text editor displays bytes as characters.

- In Notepad (on Windows)



In summary:
Binary data isn't meant to be read by humans. Write code to read that data.

Big-Endian and Little-Endian



- These are both correct, but from different contexts – one is **Big-endian**, and the other is **Little-endian**.
- Computers generally agree on what a byte is—it's 8 bits. `00000000` or `0 0`
- They don't agree on how to order multiple bytes that are grouped together (i.e., a multi-byte value).

Big-Endian and Little-Endian Example

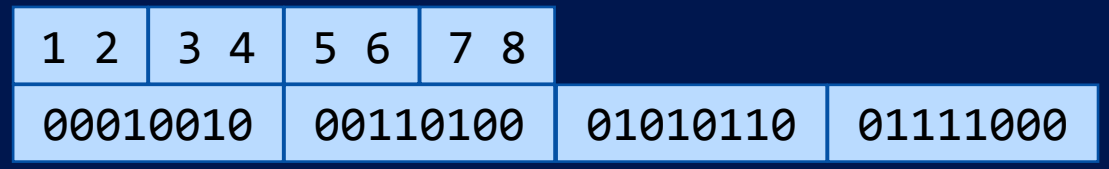
- In any numbering system, we can have “big” and “small” place values:
  - **Numeric value**: 1285
  - The 1 is “bigger” than a 2, because it’s in a larger place—the 1000s place.
  - The 5 is the “smallest” or “littlest” value in the number.

Big-endian: “Big” end written first
1285
How humans write numbers

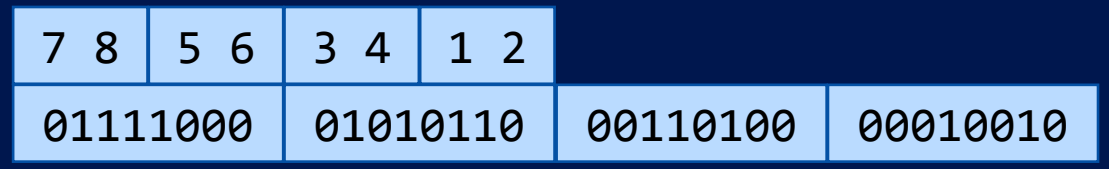
Little-endian: “Little” end written first
5821
How many (but not all) computer architectures store numbers

```
int value = 305419896; // Hexadecimal value: 0x12345678
```

- Big-endian



- Little-endian



This won’t affect you directly in this course.

Our binary operations will copy bytes in the proper order, whatever your platform is.

Binary Files Are Not Formatted Like Text Files

- Since they aren't for human eyes, binary files don't have a visible format.
  - No delimiters, newline characters, etc.
- The data is all on one "line" — one stream of bytes.
- Data is extracted according to byte counts that you specify.

Text_File_1

```
41,18467,6334,26500
```

Text_File_2

```
41|18467|6334|26500
```

Text_File_3

```
41
18467
6334
26500
```

BinaryFile

```
2900000023480000BE18000084670000
```

How do we read this kind of data?

There is **some** order to this data,
even if we don't know it at first.

To read it properly,
we have to determine that order.

How Do You Determine a File's Structure?

- There are a easy ways to learn a binary file's structure (i.e. the order of its bytes):
  1. You know it because you made it
  2. You read about it through some documentation.

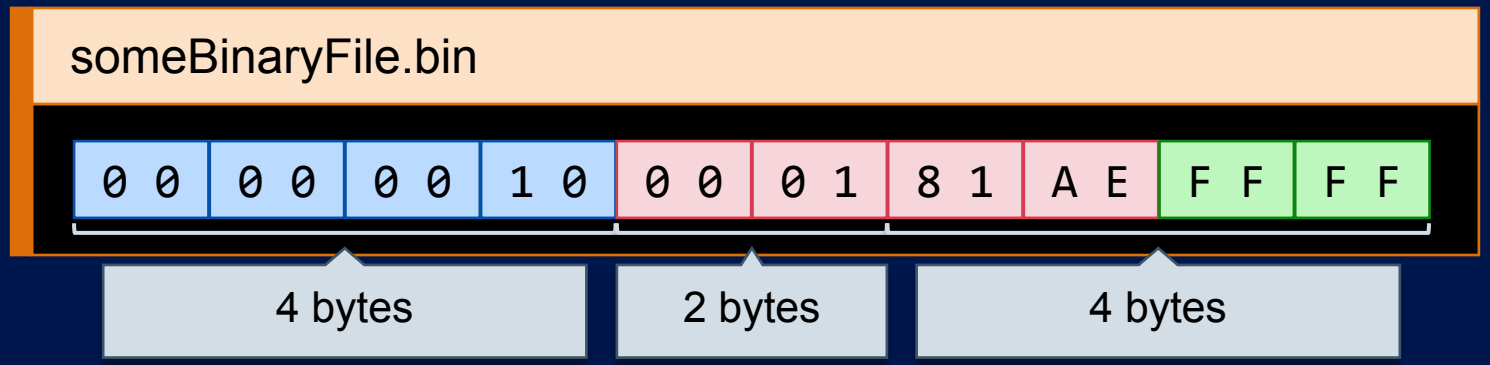
Many file formats have published info about their structure.

- If those options aren't available, the not-so-easy way:
  - Reverse-engineering (aka educated guessing!)

Not recommended – guesswork is not an effective way to program!

Some Formats Are Static

- Some imaginary file with a 4-byte integer, a 2-byte short, and a 4-byte float.



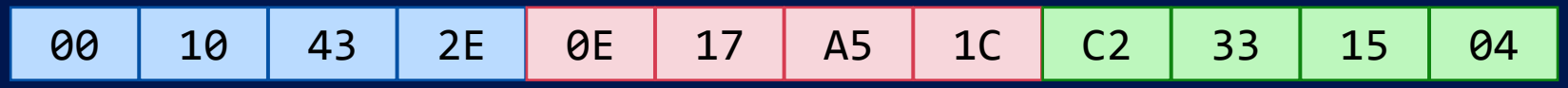
- Process of reading this file:
  1. Open file.
  2. Read 4 bytes for the integer.
  3. Read 2 bytes for the short.
  4. Read 4 bytes for the float.

It's easier to read a file that is fixed like this.

Most files will in the amounts of data they contain.

What About Dynamic Files?

- **someFile.data**: A file with 3 floats



- **otherFile.data**: A file with 4 floats

- **giantFile.data**: A file with 30,000 floats

- What process would we need to work in all situations?

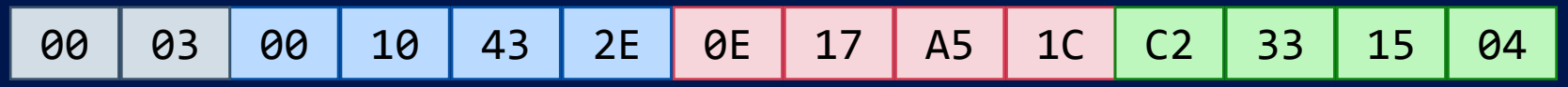
```
float myData[???];

for (int i = 0; i < ???; i++)
    myData[i] = Read4BytesFromFile(); // #notarealfunction
```

How to fix these unknowns?

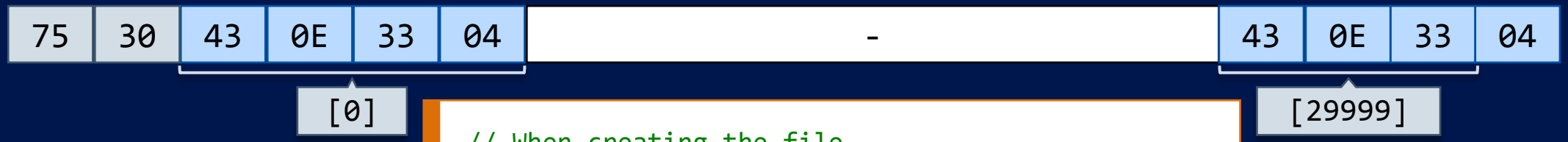
Solution: Data, to Describe the Other Data

- Some sort of count data needs to be added to the file (arbitrarily a 2-byte short, in this example).
- **someFile.data**: A file with 3 floats, preceded by a 2-byte count



- **otherFile.data**: A file with 4 floats, preceded by a 2-byte count

- **giantFile.data**: A file with 30,000 floats



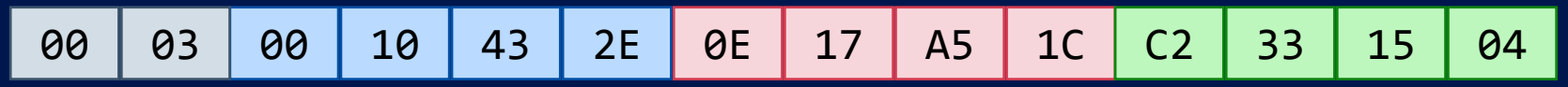

```
// When creating the file…
void WriteDataToFile(float* theData, short count)
{
     Write count
     for (count)
       write theData[i]
}
```

This 2-step process works for any type:

1. Write the AMOUNT of data
2. Write each element of the data (which may have sub-steps for complex objects)

Reading Data With Counters

- **someFile.data**: A file with 3 floats, preceded by a 2-byte count



- **otherFile.data**: A file with 4 floats, preceded by a 2-byte count

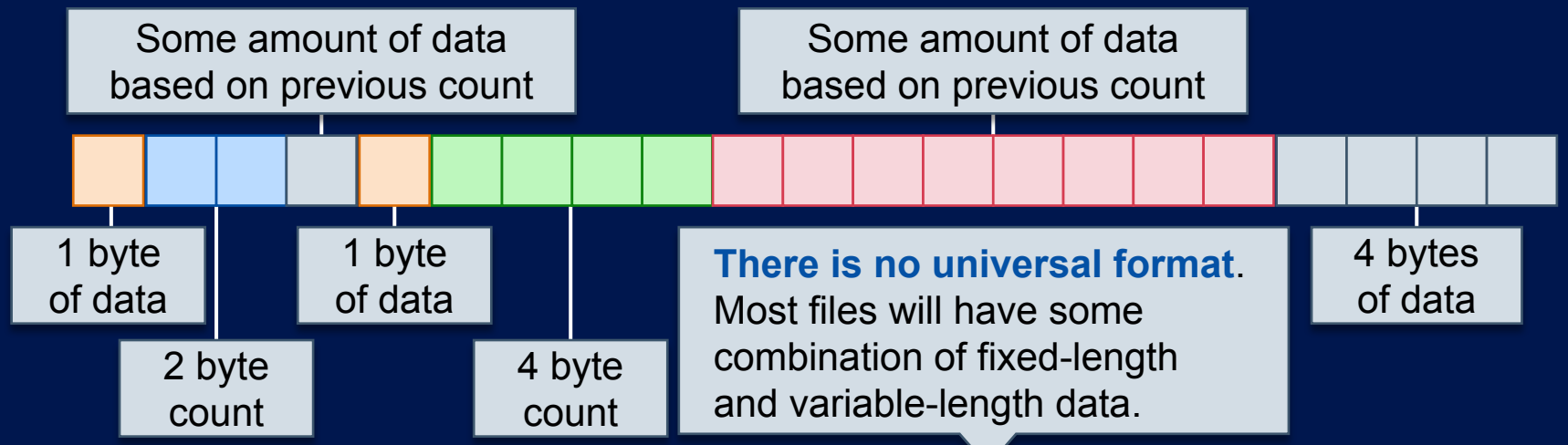

```
// When reading the file…
// We know it's a short because… we made it this way
short counter = Read2BytesFomFile(); // #alsonotarealfunction

// float myData[counter]; Variable length arrays are not valid C++
float *myData = new float[counter]; // #newtotherescue

for (int i = 0; i < counter; i++)
    myData[i] = Read4BytesFromFile(); // #stillnotarealfunction
```

We could arbitrarily decide to use an **int**, or an **unsigned long long**, instead of a **short**.

Files With Lots of Mixed Data (Hypothetical Example)

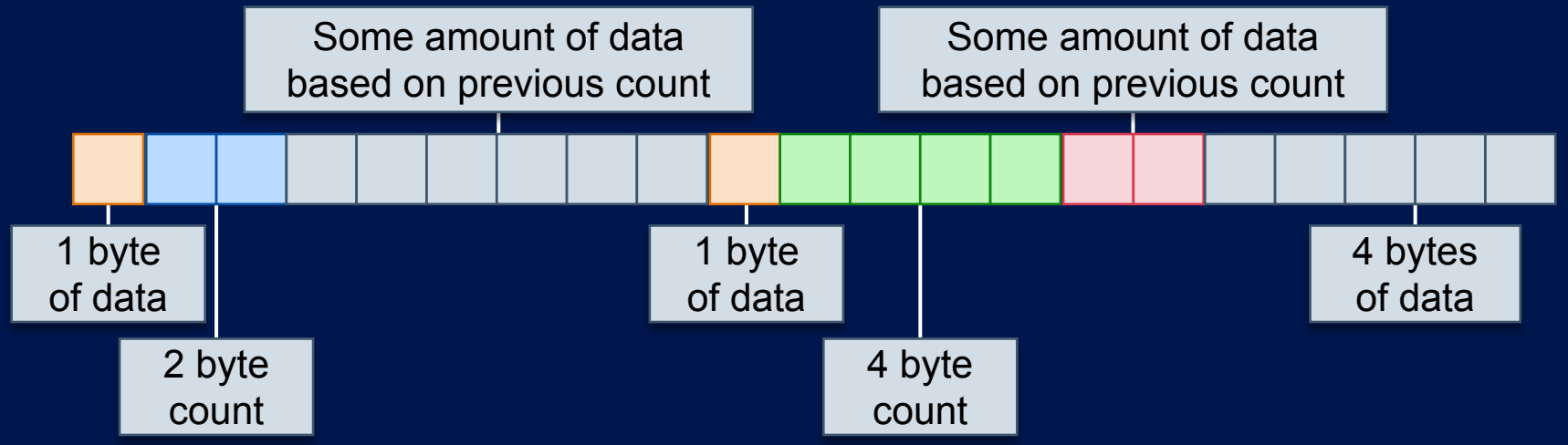


File Format Description	
1 byte	Some data
2 bytes	Count
X bytes	Data based on Count
1 byte	Data
4 bytes	Count
X bytes	Data based on Count
4 bytes	Data

Each piece of data would have a description of its purpose, how to use it, etc.

The size of data is critical. Without those byte counts it's impossible to work with binary.

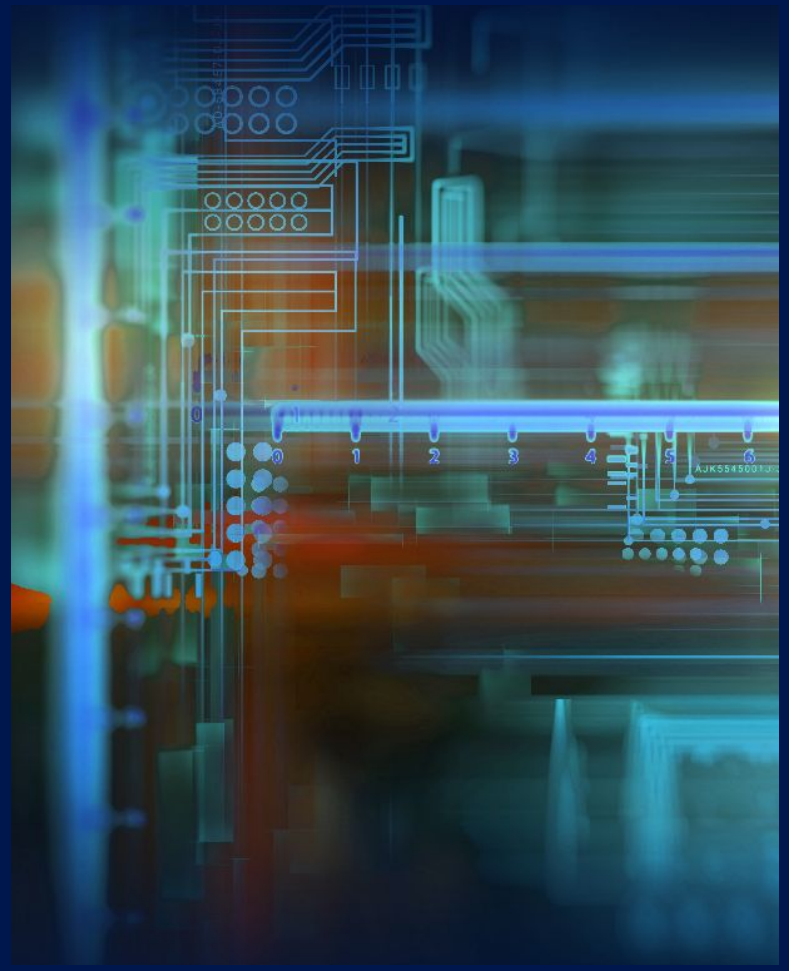
- What about the same **format**, but different **amounts** of data?



Recap

- Binary data is computer-readable, but not human-readable.
- Data stored in these files is typically (but not always) smaller.
- Text-based file I/O converts data to characters, binary file I/O copies bytes directly.
- Reading or writing binary data requires we know the byte counts of our data

Conclusion



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

Thank you for watching.