COP3503

Binary File I/O in C++

Binary File I/O in C++

- **Classes**: The same as text-based input
  - `ofstream` Output files
  - `ifstream` Input files
  - `fstream` Input **or** output files (or both)
- The same functions for opening and closing files:

```
myFile.open() // Use the ios_base::binary flag
myFile.is_open()
myFile.close()
```

Reading and Writing Differences

Text-based Operations

Writing: Use the insertion operator **<<**

```
outFile << "Hello world!" << endl;
outFile << 10 << endl;
outFile << someValue;
```

Reading: getline(), the extraction operator **>>**

```
string helloWorld;
int value;

getline(inFile, helloWorld);
inFile >> value;
```

Binary Operations

Writing: Use the **`write()`** function

```
outFile.write(/*address/*, /*byte count*/);
```

Reading: Use the **`read()`** function

```
inFile.read(/*address/*, /*byte count*/);
```

Writing with `write()`

(**`streamsize`** == long long)

```
write(const char* s, std::streamsize n);
```

A pointer to the address from which the write function starts copying

How many bytes to copy

This pointer **must** be a **`char*`**, regardless what you're writing to a file.

Why must it be a **`char*`**? Because this operation deals with copying bytes, one at a time.

If you aren't writing **`char`** data? You'll have to use a **typecast** to convert your pointer.

In C++, the **`char`** data type is defined as always 1 byte.

write() Examples

```
write(const char* s, std::streamsize n);
```

```
char letter = 'X';
const char* hero = "Batman";
int value = 256;

ofstream file("MyData.bin", ios_base::binary);

file.write(&letter, 1);
file.write(hero, 6 + 1);
file.write(reinterpret_cast<char*>(&value), sizeof(value));
```

Write 1 byte, starting from the address of value.

Write 7 bytes (the +1 is for that null-terminator character), starting at name. Name is an array of char, and we can use arrays like pointers.

What the heck is this?!

Calculate the size of the variable.

+ `reinterpret_cast` is a C++ style **typecast**.

C++ Typecasts

- There are **four** different types of C++ casts:

Four Types of C++ Casts
static_cast
reinterpret_cast
const_cast
dynamic_cast

```
// Same syntax for all four:
cast_type<targetType>(sourceData)
```

```
float value = 6.5f;
cout << (int)value; // C-style cast
cout << static_cast<int>(value); // C++ cast

char* ptr = (char*)&value; // C-style cast
ptr = reinterpret_cast<char*>(&value); // C++ cast
```

The overall goal for any cast is the same—convert one data type to another.

There are different goals for conversions, and these signify the desired outcome.

If used **correctly**, it might look like there's little to no difference between C and C++ casts.

Two very important questions students ask:

1. Why bother using C++ casts?
2. C-Style casts look easier; can I just use those?

C-Style Typecasts

```cpp
// Cast either value to a float, for a float result
int x = 10, y = 4;
float result = (float)x / y; // 10/4 == 2 with integer division
```

```cpp
float value = 6.5f;
cout << (int)value; // "Cast away" the fractional part, print 6
```

```cpp
char letter = 'A';     // Numeric value of 'A' == 65
cout << letter;        // Prints 'A', or (char)65
cout << (int)letter;   // Prints 65, or (int)'A'
```

These will work in C++ (you've probably used some before!).

```cpp
// Cast a float* to a char*, C-style
char* ptr = (char*)&value;
outFile.write((char*)&value, 4);
```

However… there are C++ specific typecasts that, and it's recommended you use those instead.

Your Cast Questions Answered!

- Some C++ casts perform **compile-time** checks.
- If a cast would cause problems, the compiler reports it (before problematic code executes!).

```
char smallValue = 20;
// Problem: *ptr SHOULD BE a 4-byte value... but only 1 byte is there
int* ptr = (int*)&smallValue;

// Generates a compiler error (int* and char* types differ)
int* ptr = static_cast<int*>(&smallValue);
```

- A `reinterpret_cast` will work in this case.

C++ casts make your intent clear.

You're not likely to accidentally write `reinterpret_cast<>()`.

```
// This says "I know these two types aren't exactly the
// same... I know the risks, I want to convert it anyway!"
int* ptr = reinterpret_cast<int*>(&smallValue);
```

Can you just "get away with" C-style casts?

Yes, and it could work just fine for you.

You can find lots of debate over the two in programming communities.

What About const_cast and dynamic_cast?

- const_cast allows you to “cast away” the const-ness of a pointer.
  - It’s a bit of a niche operation, one you may not use all that often.
  - In this course, we won’t need it, so we’ll just move on past it.

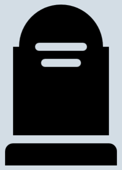
```
void Foo(const Example* ptr)
{
    // Bypass const protection – use sparingly, with great care!
    Example* notConst = const_cast<Example*>(ptr);
    notConst->FunctionThatChangesTheObject();
}
```

- dynamic_cast is used when we’re working with inheritance and polymorphism.
  - We aren’t there yet, so we’ll ignore this until then.
  - Even then, you may not use this one much either.

Typecasting: Converting to char* for write()

```
int someNumber = 10;
file.write(reinterpret_cast<char*>(&someNumber), 4); // &someNumber == int*

short speed = 200;
file.write(reinterpret_cast<char*>(&speed), 2);      // &speed == short*

double z = 0.0;
file.write(reinterpret_cast<char*>(&z), 8);          // &z == double*

int arrayOfInts[3] = { 2, 4, 6 };
file.write(reinterpret_cast<char*>(arrayOfInts), 12);// arrayOfInts == int*

char bling = '$';
file.write(&bling, 1); // &bling is already a char*, casting is unnecessary

int SomeFunction();
file.write(reinterpret_cast<char*>(&SomeFunction()), 4); // Can't do this!

// Get the value from the function, THEN write.
int value = SomeFunction();
file.write(reinterpret_cast<char*>(&value), 4);
```

Return values don't "live" anywhere in memory. They get copied into variables that "catch" the return… or the value "dies".



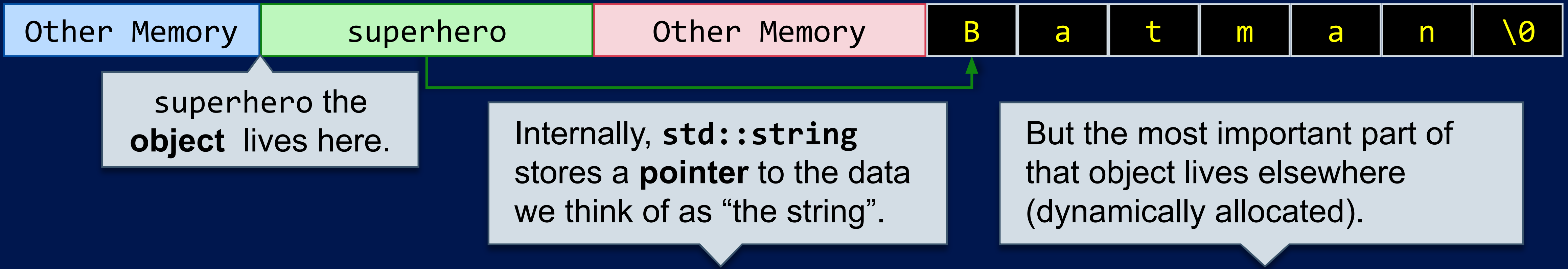
Sorry value.
You will be missed.

What About Writing Complex Objects?

```
SomeObject obj("Bob", 115, 1.732f);
file.write(reinterpret_cast<char*>(&obj), sizeof(obj)); // &obj == SomeObject*
```

- You should (almost) never write an instance of a class as a single operation.
- Why not? Objects can be made of pointers, dynamic memory, maybe other complex objects!
- The memory “footprint” of any given object may not be contiguous.

```
string superhero = "Batman";
```



What Happens if We Try to Write the Object?

```
string superhero = "Batman";
```

Other Memory	superhero	Other Memory	B	a	t	m	a	n	\0

If we write from the address of superhero…

We get data we already don't want, plus "extra bytes" containing unknown values!

This is the data that we want to write to a file.

So, in general:

⚠ **DO NOT** ATTEMPT TO WRITE COMPLEX OBJECTS AS A SINGLE BLOCK OF MEMORY.

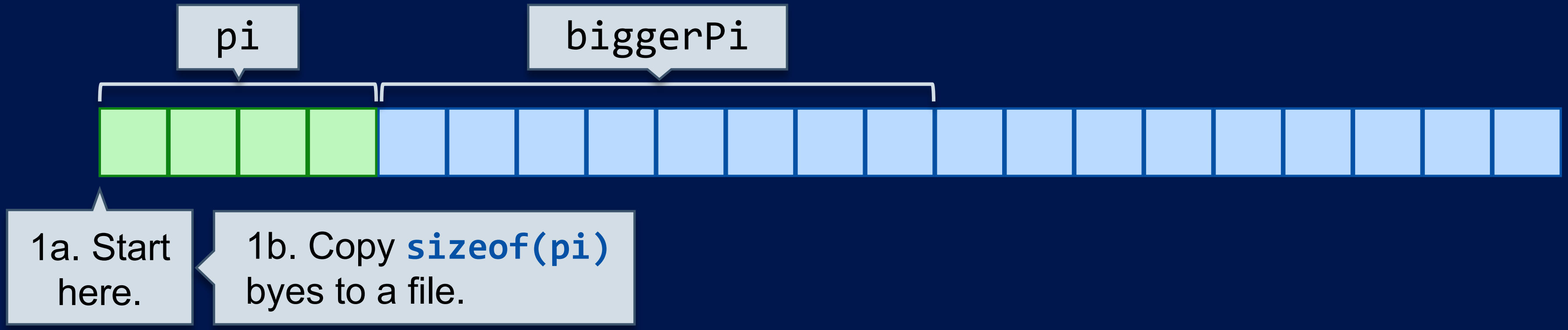
Instead, use multiple write operations for the relevant member data.

Writing Some Binary Data with o/fstream

Seemingly “simple” data.
It can be deceptive how much.

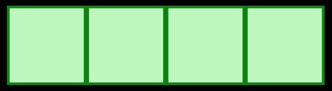
```
float pi = 3.14;
double biggerPi = 3.14159;
int notAPi = 3;
string pie = "Apple";

fstream file("BinaryFile.bin", ios_base::out | ios_base::binary);
if (file.is_open())
{
    file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
    file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
    file.write(reinterpret_cast<char*>(&notAPi), sizeof(notaPi));

    // Write a string as a 2 step process
    // 1. Write the length of the string
    unsigned int length = pie.size() + 1; // +1 for null-terminator '\0'
file.write(reinterpret_cast<char*>(&length), sizeof(length));

    // 2. Write the string data itself
    // c_str() returns the "real" string data as a char*
    file.write(pie.c_str(), length);
}
```

So, What Actually Happens?

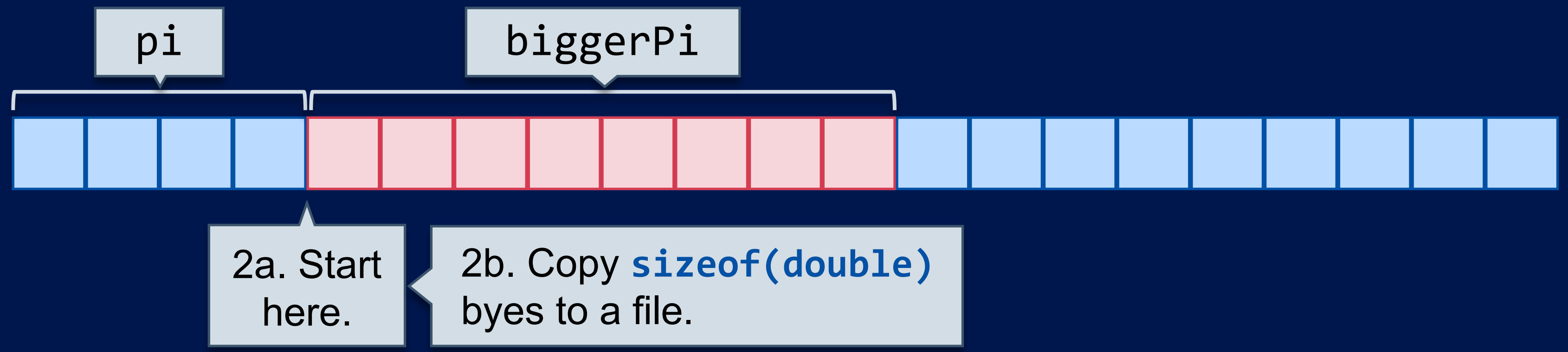



```
file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
```

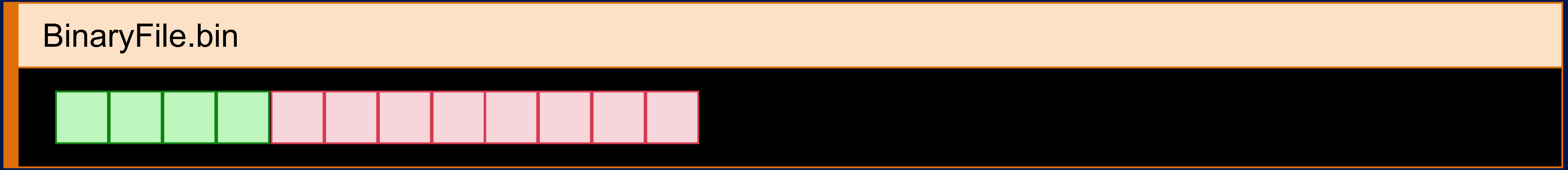
BinaryFile.bin



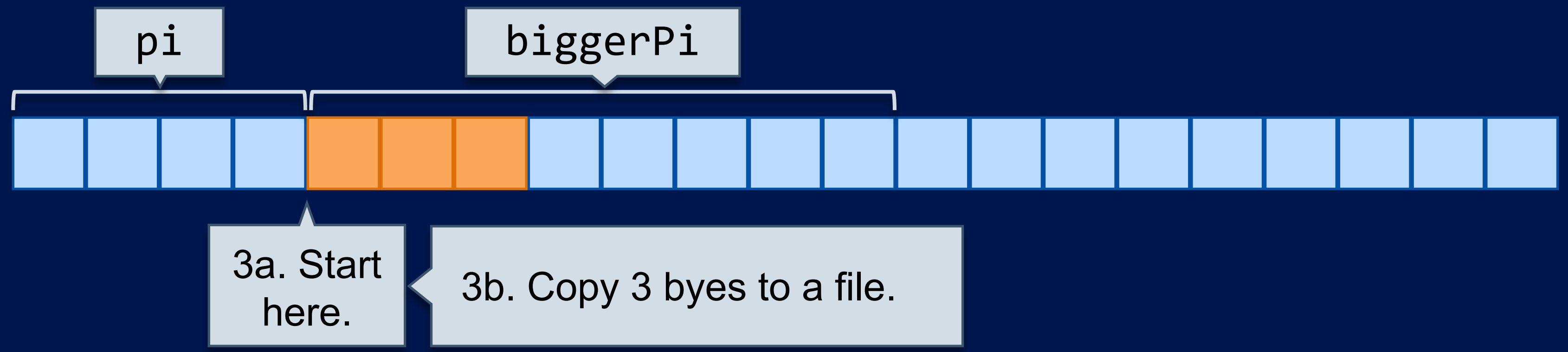
So, What Actually Happens?




```
file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
```

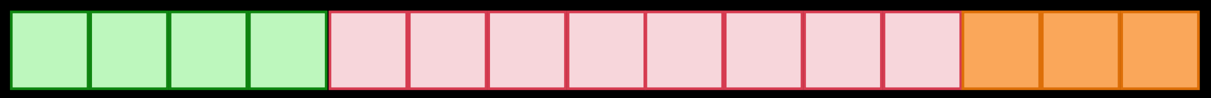


So, What Actually Happens?

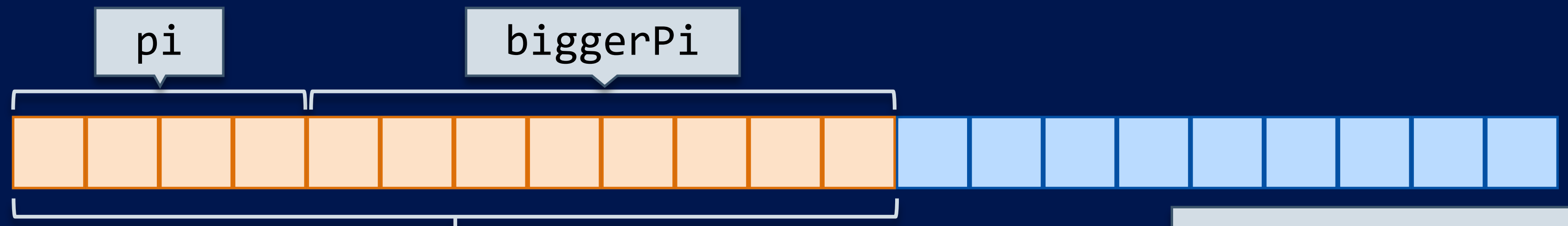



```
file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
file.write(reinterpret_cast<char*>(&biggerPi), 3);    // What will happen here?
```

BinaryFile.bin



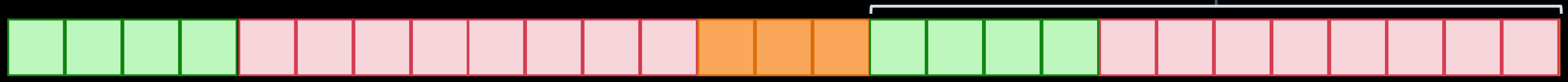
So, What Actually Happens?



From the indicated address, copy the indicated number of bytes.

```
file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
file.write(reinterpret_cast<char*>(&biggerPi), 3);    // What will happen here?
file.write(reinterpret_cast<char*>(&pi), 12);         // What about here? What about 20? 50?)
```

BinaryFile.bin



So, What Actually Happens?

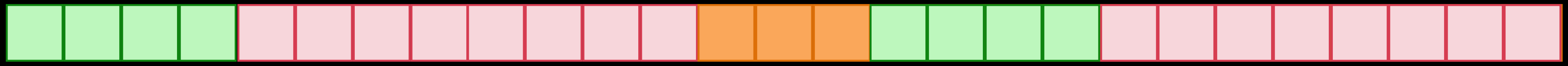
Copying “past” a variable **can** be done very easily…

For it to work, your memory must be contiguous, with no “gaps” between variables.

If you aren’t **certain**, then one variable at a time is best, it will always work.

```
file.write(reinterpret_cast<char*>(&pi), sizeof(pi));
file.write(reinterpret_cast<char*>(&biggerPi), sizeof(biggerPi));
file.write(reinterpret_cast<char*>(&biggerPi), 3);    // What will happen here?
file.write(reinterpret_cast<char*>(&pi), 12);         // What about here? What about 20? 50?)
```

BinaryFile.bin



What About Reading the Wrong Amount?

00000000	00000001	00000000	00000100

A value of 1, stored in 2 bytes.

A value of 4, stored in 2 bytes.

```
int value;
someFile.read(reinterpret_cast<char*>(&value), 4);
```

value

	65536 place	256 place	1s place
00000000	00000001	00000000	00000100

A value of 65,540, stored in 4 bytes

If we don't take the time to calculate all of this ourselves, we might think:

"My computer is doing weird stuff."

"My program is acting funny."

"How does 1 + 4 = 65,540?"

"Unexpected results" is what happened here.

A seemingly "simple" code issue might take considerable time to break down, understand, and fix.

sizeof()

Pointers are always the same size, regardless of data type.

That size, however, could be 4, 8, maybe something else, depending on the platform. Generally speaking: `sizeof(any pointer) == sizeof(int)`

- Returns the size, in bytes, of the specified variable or type

```
int x = 50;
double pie = 3.14;
sizeof(char);              //  1
sizeof(unsigned char);     //  1
sizeof(short);             //  2
sizeof(unsigned int);      //  4
sizeof(x);                 //  4
sizeof(float);             //  4
sizeof(pie);               //  8
sizeof(long long);         //  8
```

```
int *x = new int[500];
char *aString;
sizeof(char *);            // 4
sizeof(aString);           // 4
sizeof(short *);           // 4
sizeof(unsigned int *);    // 4
sizeof(x);                 // 4
sizeof(float *);           // 4
sizeof(double *);          // 4
sizeof(long long *);       // 4
```

C++ Sizes Are...Unusual

- Many common data types (such as int, short) have loosely defined sizes.
- We might often talk of int as 4 bytes in size (and commonly it is…).
- It's officially defined as being **at least** 16 bits (2 bytes) in size—that's not very specific.
- A short is often treated as 2 bytes, but it's also defined as **at least** 2 bytes.

https://en.cppreference.com/w/cpp/language/types

- This can make some code problematic, as it may not always give the same results.

```
sizeof(int)
```

- What will this **always** be? 2, 4, 8?

```
int x = 10;
// 4 bytes is PROBABLY okay... but it might not be
write(reinterpret_cast char*>(&x), 4);
```

To avoid some of this ambiguity, C++ has data types with exact sizes.

C++ Data Types With Exact Sizes

Data Type	Size in Bits (8 bits == 1 byte)
`char, unsigned char`	**Exactly** 8 bits
`int8_t, uint8_t`	**Exactly** 8 bits
`int16_t, uint16_t`	**Exactly** 16 bits
`int32_t, uint32_t`	**Exactly** 32 bits
`int64_t, uint64_t`	**Exactly** 64 bits

Data Type	Size in Bits (8 bits == 1 byte)
`short`	**At least** 16 bits (2 bytes)
`int`	**At least** 16 bits (2 bytes)
`unsigned int`	**At least** 16 bits (2 bytes)
`long long`	**At least** 64 bits (8 bytes)

If you ever write code (for a job, commercial software, etc.) that might have to run on more than one platform, use specific sizes.

In this course, using int or short will be fine—they should always be 2 or 4 bytes, respectively.

Just be aware that there are a lot more details that go beyond material we cover in this course.

For more information: https://en.cppreference.com/w/cpp/types/integer

sizeof() and Classes

- Can you use **sizeof()** on class objects?
- Yes! Will it you give the number you really need? Probably not!
- Classes are **containers**, often containing pointers, or using dynamic memory allocation.
- **sizeof()** will tell you how big the container is… not necessarily the size of the **contents**.

```
struct SomeData
{
    int x, y; // 8
    int* ptr; // 4
};
```

```
SomeData example;
example.ptr = new int[500]; // Allocate 2000 bytes

// Total memory used: 2000 + 12 bytes
cout << sizeof(example); // Prints 12
```

The **sizeof()** operator doesn't take dynamic allocations into account.

sizeof() on STL Containers

```
string hero = "Batman";
string zero = "";
string big = "Great beard of Zeus, what a long string! What will sizeof() return?"
vector<float> floatVec;
vector<float> floatVec2(500);
```

```
cout << "sizeof hero: " << sizeof(hero) << endl;
cout << "sizeof zero: " << sizeof(zero) << endl;
cout << "sizeof big: " << sizeof(big) << endl;
cout << "sizeof string: " << sizeof(string) << endl;
cout << "sizeof vector o' floats: " << sizeof(floatVec) << endl;
cout << "sizeof vector o‘ 500 floats: " << sizeof(floatVec) << endl;
```

C:\WINDOWS\system32\cmd.exe

```
sizeof hero: 28
sizeof zero: 28
sizeof big: 28
sizeof string: 28
sizeof vector o' floats: 16
sizeof vector o' 500 floats: 16
Press any key to continue . . .
```

+ **sizeof()** is not the same as **size()**…

```
unsigned int total = floatVec2.size() * sizeof(float);
cout << "Size of data in vector: " << total << endl;
```

C:\WINDOWS\system32\cmd.exe

```
Size of data in vector: 2000
Press any key to continue . . .
```

In many cases, **sizeof()** with class objects isn't something you'll need to use.

Reading with `read()`

```
read(char *s, std::streamsize n);
```

A pointer to the address at which the read function starts copying

How many bytes to copy

This seems suspiciously familiar…

```
write(const char* s, std::streamsize n);
```

Reading Some Binary Data with `i/fstream`

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

```
ifstream file("BinaryFile.bin", ios_base::binary);
if (file.is_open())
{
    char byte;
    short count;
    file.read(&byte, 1); // No cast needed, &byte is a char*
    file.read(reinterpret_cast<char*>(&count), 2);

    int* data = new int[count];
    file.read(reinterpret_cast<char*>(data), count * sizeof(data[0]));

    // Okay to overwrite and reuse variable, if you're done with it
    file.read(&byte, 1);

    int nextValue;
    file.read(reinterpret_cast<char*>(&nextValue), 4);

    // And so on, and so on...
}
```

Hard-coding the size can be okay…
if you are **sure**. Like, really sure.

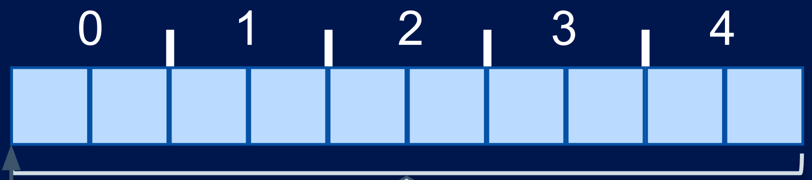
Calculate the total size of the array.

This is just reading the data into your program. What you do after that is up to you.

For example, the “data” variable will be a memory leak if you don’t delete it at some point…

Reading (or Writing) Collections of Data

```
short someData[5];
```



Start here…

Read/write all 10 bytes at once.

This is a very “computer-esque” way to work with data.

Computers don’t “care” about any variables, specific index locations, and so on. They just read and write bytes.

```
ifstream inputFile("important.data", ios_base::binary);

// Read each element one at a time
for (int i = 0; i < 5; i++)
    inputFile.read(reinterpret_cast<char*>(&someData[i]), 2);

// OR, read the entire array
inputFile.read(reinterpret_cast<char*>(someData), sizeof(someData[0]) * 5);
```

```
// Data goes out…
ofstream outputFile("output.data", ios_base::binary);
outputFile.write(reinterpret_cast<char*>(someData), sizeof(someData[0]) * 5);
```

Advanced: Reading an Entire File

(Not necessary in this course)

```
ifstream someFile("example.binaryFile", ios_base::binary);
someFile.seekg(0, ios_base::end); // Jump to the end of file with seekg()

unsigned int byteCount = someFile.tellg(); // How many bytes are in the file?
someFile.seekg(0, ios_base::beg); // Go back to the start of file

char* data = new char[byteCount];
someFile.read(data, byteCount); // Entire file, stored in memory!
```

What you do **after** you have all of this data… is up to you!
Store it?
Save a copy?
Compare to something else?
Etc…

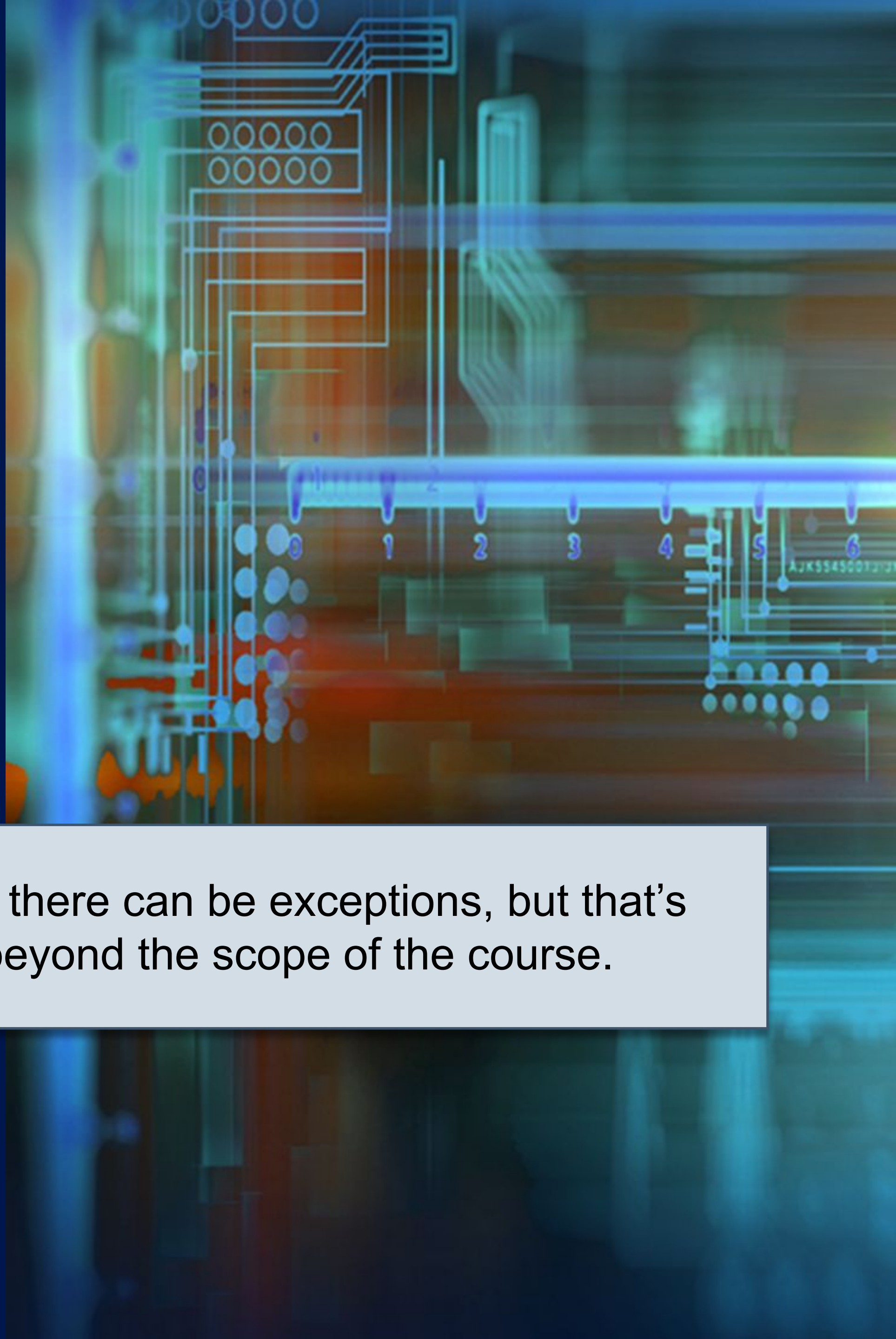
You may never do something like this (you won't have to in this course).

But… many languages have functionality like this, in the event that you do.

You could also do this for parts of a file—reading in large sections of a file all at once.

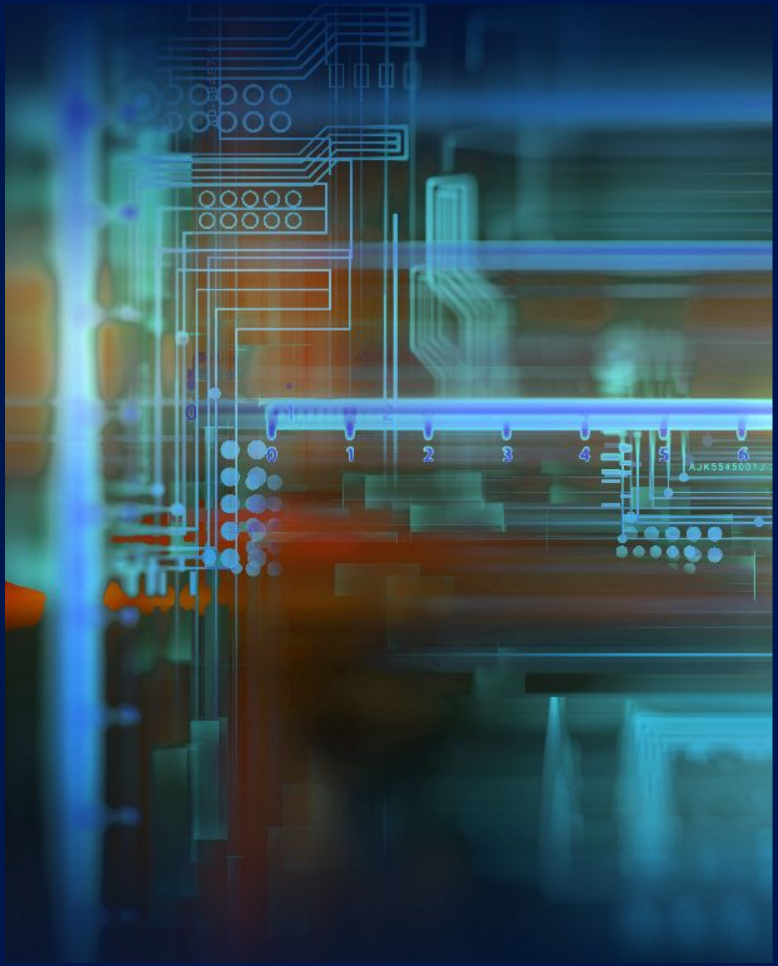
Recap

- Binary file I/O in C++ uses the same stream classes as text-based operations.
- We use the `read()` and `write()` functions.
  - Not the insertion operator `<<` or the extraction operator `>>`
- Copying bytes (to or from a file) deals with character pointers.
  - Typecasting is required for non-character types.
  - C-Style casts can work, but more formal C++ casts are recommended.
- Copying bytes requires exact byte counts—a single incorrect byte can ruin everything!
- The `sizeof()` operator can be used to determine the size of variables.
  - Generally not used for class objects, only primitives



* there can be exceptions, but that’s beyond the scope of the course.

Conclusion



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

Thank you for watching.