COP3503

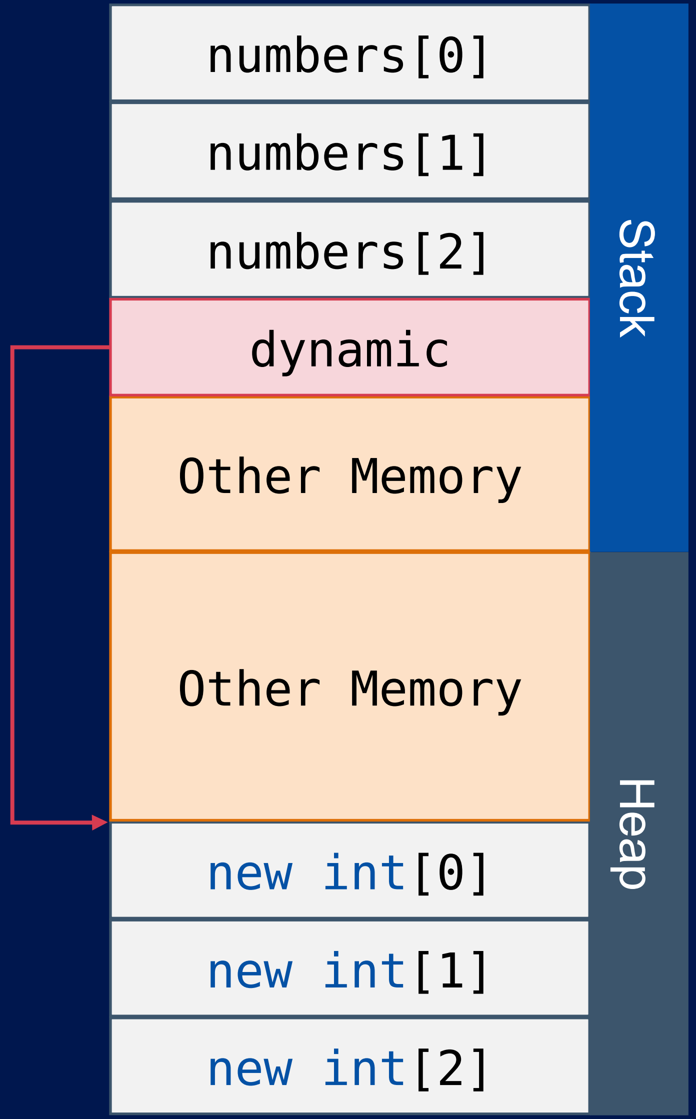
Dynamic Memory

Dynamic Memory

What is it?

- Memory that is allocated on the **heap** (the **free-store** in C++)
- Memory **allocated**, or reserved, using the **new** operator
- Memory that must be **deallocated** with **delete** when no longer needed

```
// 3 numbers in stack memory, "automatic storage"
int numbers[3];

// 1 pointer in stack memory, automatic storage
// 3 integers in HEAP memory, "dynamic storage"
int* dynamic = new int[3];
```



Question: Why go through all this trouble?

Answer: Because sometimes we have to

Compilers Need Constant Information

- Your compiler needs to know the size of data at compile time.
- This information ensures everything lines up in memory.
- Non-constant data throws off this process.

```
float hardCodeOK[10];   // Okay, 10 will always be 10

const int input = 5;
float constOK[input];   // Okay, input is constant

int count = 50;
cin >> count;
count += 200;
count -= 15;
float variableBad[count]; // Not okay, count COULD change
```

Stack Frame
4 * 10 bytes
4 * 5 bytes
4 * ??? bytes

Unknown array size, unknown stack frame size!

Side note: Some compilers **do** allow this, with non-standard extensions. Not supported everywhere, best to avoid using it.

Pointers, new, and the Heap to the Rescue!

```
int input;
cin >> input;

float variableBad[count];

// Dynamically allocate an array
float* scores = new float[input];
```

The size of a heap allocation isn't needed at **compile time**.

Your compiler sees this and says "That's not on the stack? Okay, we'll worry about the size of that later."

After this, the variable can be used like a normal array:

```
for (int i = 0; i < count; i++)
        cin >> scores[i];
```

Stack Frame
4 * 1 bytes (input)
4 * 1 bytes (scores)

Anatomy of an Allocation

```
float* scores           = new float[someVariable];          // Variable amount? okay
char* letters     = new char[10];                            // Hard-coding is okay
int* numbers            = new int[10000000];                 // Too big for the stack!
SomeObject* obj   = new SomeObject;                       // Just one? Okay too!
```

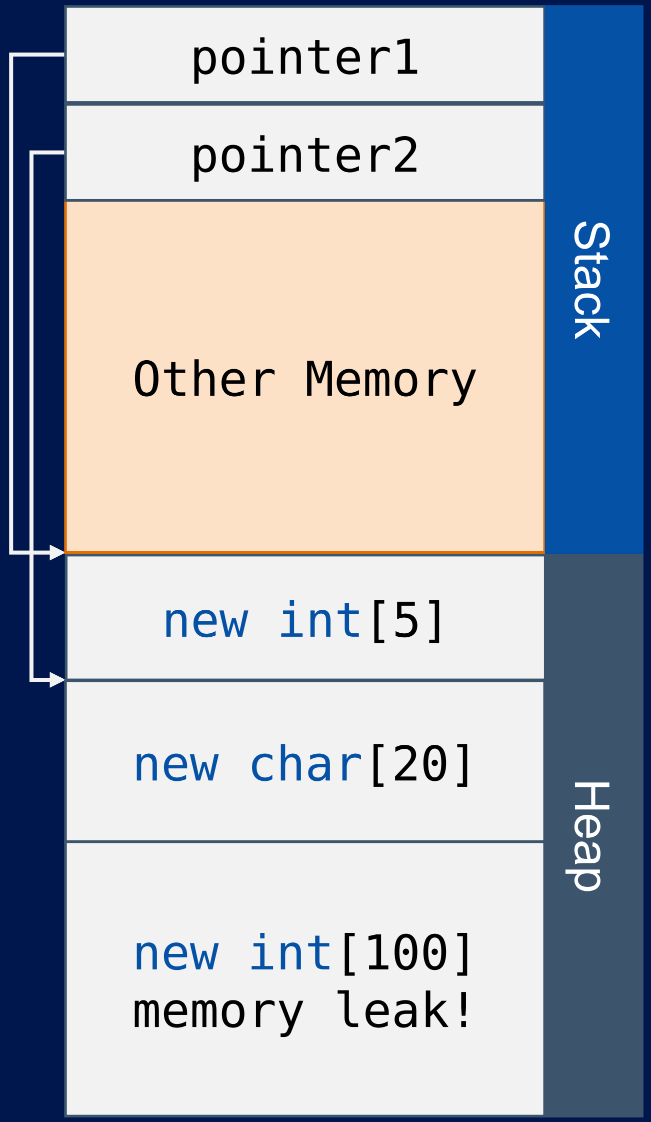
Structure of any memory allocation:

```
DataType* singleObject = new DataType; // OR.
DataType* moreThanOne  = new DataType[quantit
```

new memory has to be **caught** and stored by something, otherwise, it causes **memory leaks**.

Catching New Memory

```
int Sum(int value1, int value2)
{
        return value1 + value2;
}

Sum(10, 50);   // Waste of effort
10 + 50;                // Ditto

// Catch or use the results
int sum = Sum(10, 50);
cout << sum << endl << Sum(20, 7);

int* pointer1 = new int[5];
char* pointer2 = new char[20];

new int[100]; // Allocate 100 integers, and... forget about them!
```



Viewing the Address of Something

- Dynamic allocations **don't "fall out of scope"**.
- Once allocated with **`new`**, that memory is "reserved" until you **deallocate** it.
- Deallocating memory "frees" it, making it available for some future reservation.
- Dynamic memory is deallocated with the **`delete`** operator.
- We **`deallocate memory addresses`**, not variables (though pointer variables store the addresses!).

```
SomeObject* obj = new SomeObject;
delete obj;
```

We want to deallocate the memory reserved by this call to **`new`**.

This pointer stores that address.

We are **not** deleting the `obj` variable.

Deleting Memory

Two ways to do it

For single objects:

```
SomeObject* obj = new SomeObject; // Allocate memory
delete obj;                          // Deallocate
int* intPointer = new int;           // Allocate
delete intPointer;                   // Deallocate
```

For arrays of objects:

```
double* someData = new double[100]; // Allocate an array
delete[] someData;                       // Deallocate an array
```

What Are `new` and `delete`, Really?

- Behind the scenes, **`new`/`delete`** are operators, special types of functions (more on these later).

What you write	What is being called
`new int`	`operator new()`
`new int[10]`	**`operator new[]()`**

What you write	What is being called
`delete somePointer;`	`operator delete()`
`delete[] somePointer;`	**`operator delete[]()`**

Simple way to remember:

Use `[]` with **`new`**?
Use `[]` with **`delete`**.

What if you mix and match?

```
new -> delete[]
new[] -> delete
```

Undefined behavior!
Might work, might not,
might throw an exception.

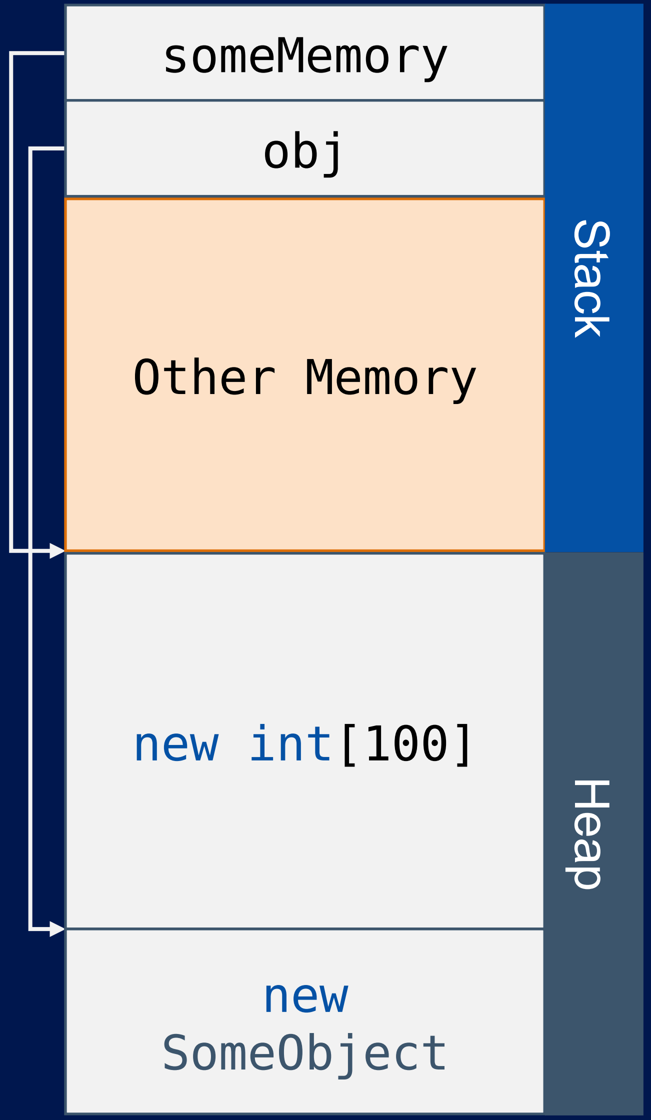
Can We Delete By Using `nullptr`?

```
somePointer = nullptr; // Same as delete?
```

- **Short answer**: No!
- **Long answer**: Noooooooo!
- Setting a pointer to **`nullptr`** just erases the treasure map, not the treasure!

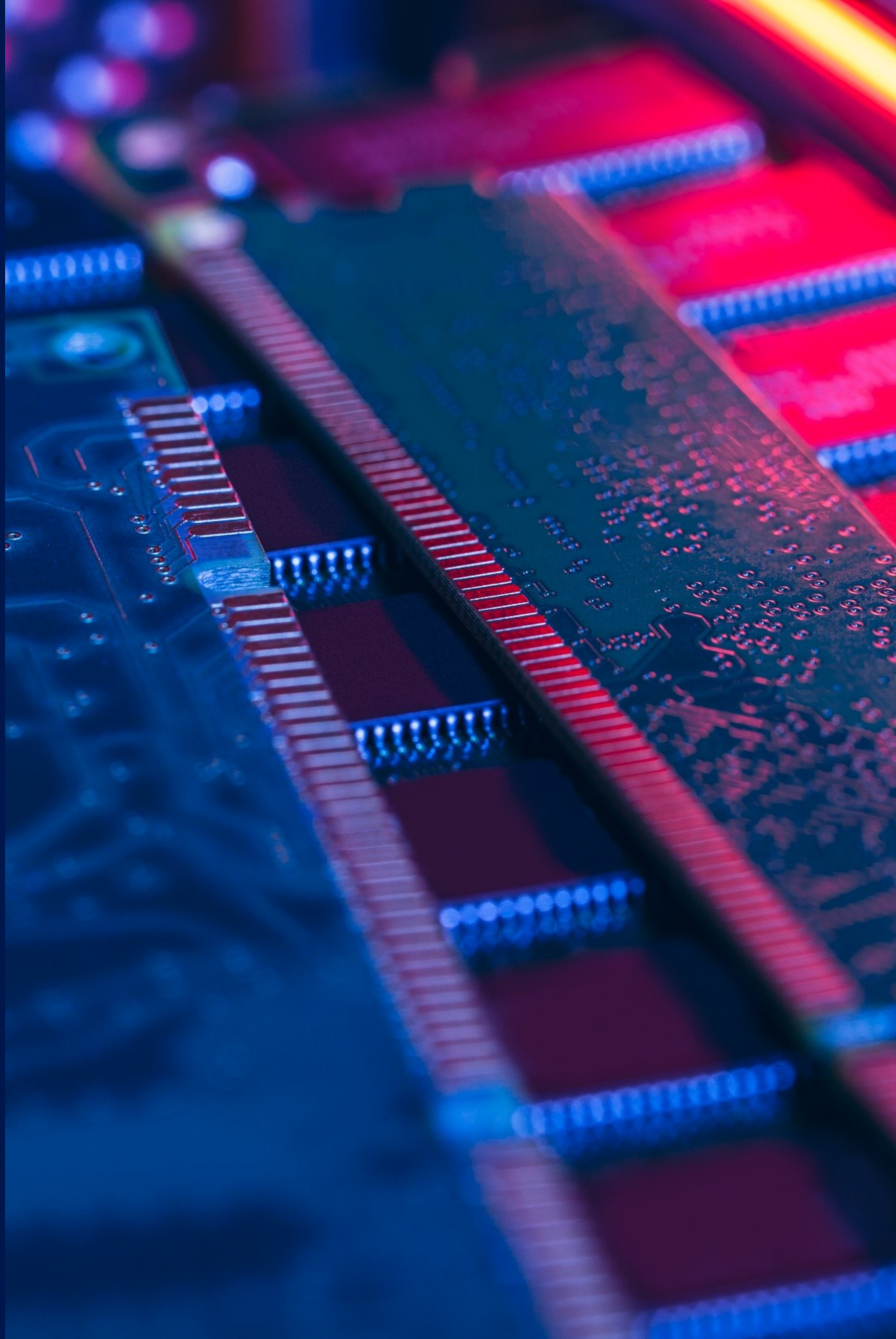
```
int* someMemory = new int[100];
SomeObject* obj = new SomeObject;
// Do some stuff with the variables

// All done!
someMemory = nullptr;
obj = nullptr;
```

Memory leaks! Only **`delete`** (or **`delete[]`**) will deallocate memory!



When Do You Delete Memory?

- **In short**:
  When your program no longer needs something
- Yeah but… when is **that**?
  (Answer: depends on the program!)
- `main()` is just one function.
  - Programs may have a lot of processes, all with different goals.
  - You have to decide it's best to create or delete something.
- Classes have **destructors** which are called when objects fall out of scope (or when **`delete`** is used to deallocate dynamic objects).
- Destructors are the perfect place for classes to delete memory (it's what they were made for!).



When to Delete

Examples

```
int main()
{
        float* someData = nullptr;
        int count;
        cin >> count;
        someData = new float[count];

        // Do some stuff with the
array

        // All done,
        delete[] some
        return 0;
}
```

Reusing variables is fine
Just be sure to **delete**
before reassigning a
pointer to something **new**.

```
int main()
{
        float* someData = nullptr;
        int count;
        cin >> count;
        someData = new float[count];

        // Do some stuff with the
array
        // All done, clean it up
        delete[] someData;

        // Repeat the process
        cin >> count;
        someData = new float[count];

        // Do some stuff with the
array
        // All done, clean it up
        delete[] someData;
        return 0;
}
```

Using delete on the Wrong Pointer

Calling `delete` (or `delete[]`) on dynamic allocation	Result
`int* someData = new int[100];` `delete[] someData; // Good!`	**Success! No issues**

Calling `delete` on non-allocated memory	Result

`float someData[10];`
`delete[] someData;     // Bad!`
`SomeObject x;`
`delete x;              // Bad!`
 	
`int* numbers = new int[100];`
`delete[] numbers;      // Once is okay`
`delete[] numbers;      // Twice is bad!`
	Undefined behavior – program will probably throw an exception.
	
	Can't deallocate something never allocated
	
	Can't deallocate something twice

Calling `delete` on a null pointer (`nullptr`)	Result

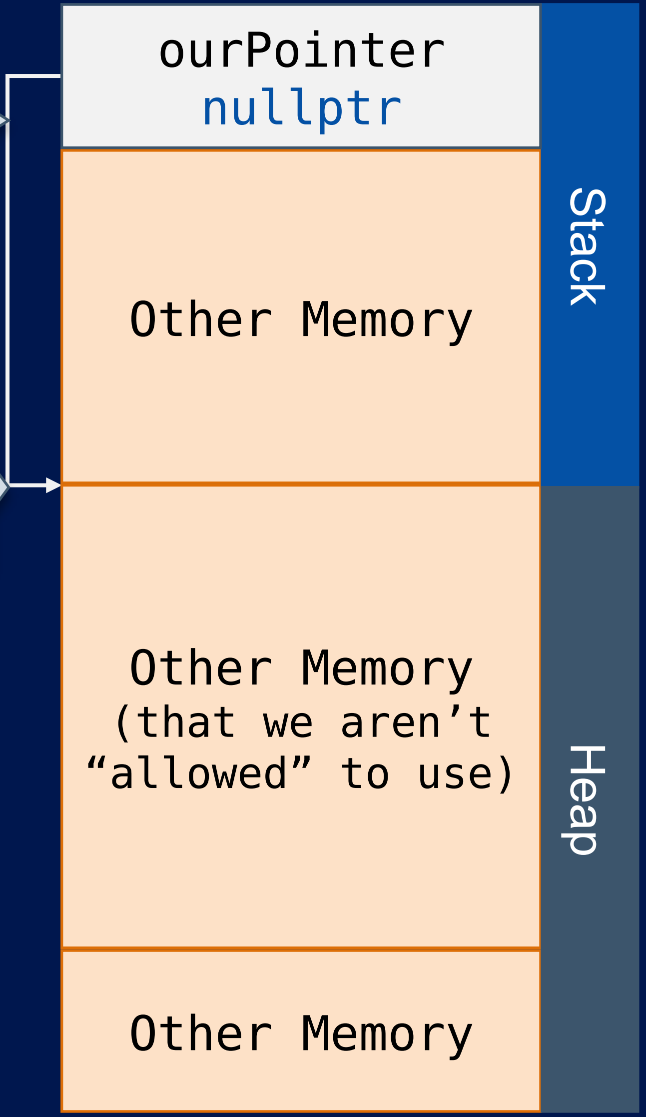
`SomeObject* pointer = nullptr;`
`delete pointer;   // Okay`
`delete[] pointer; // Okay`
	The C++ standard says `delete` on a null pointer is okay.

Setting Pointers to `nullptr` After `delete`

The pointer still points to the same location after **`delete`** is called.

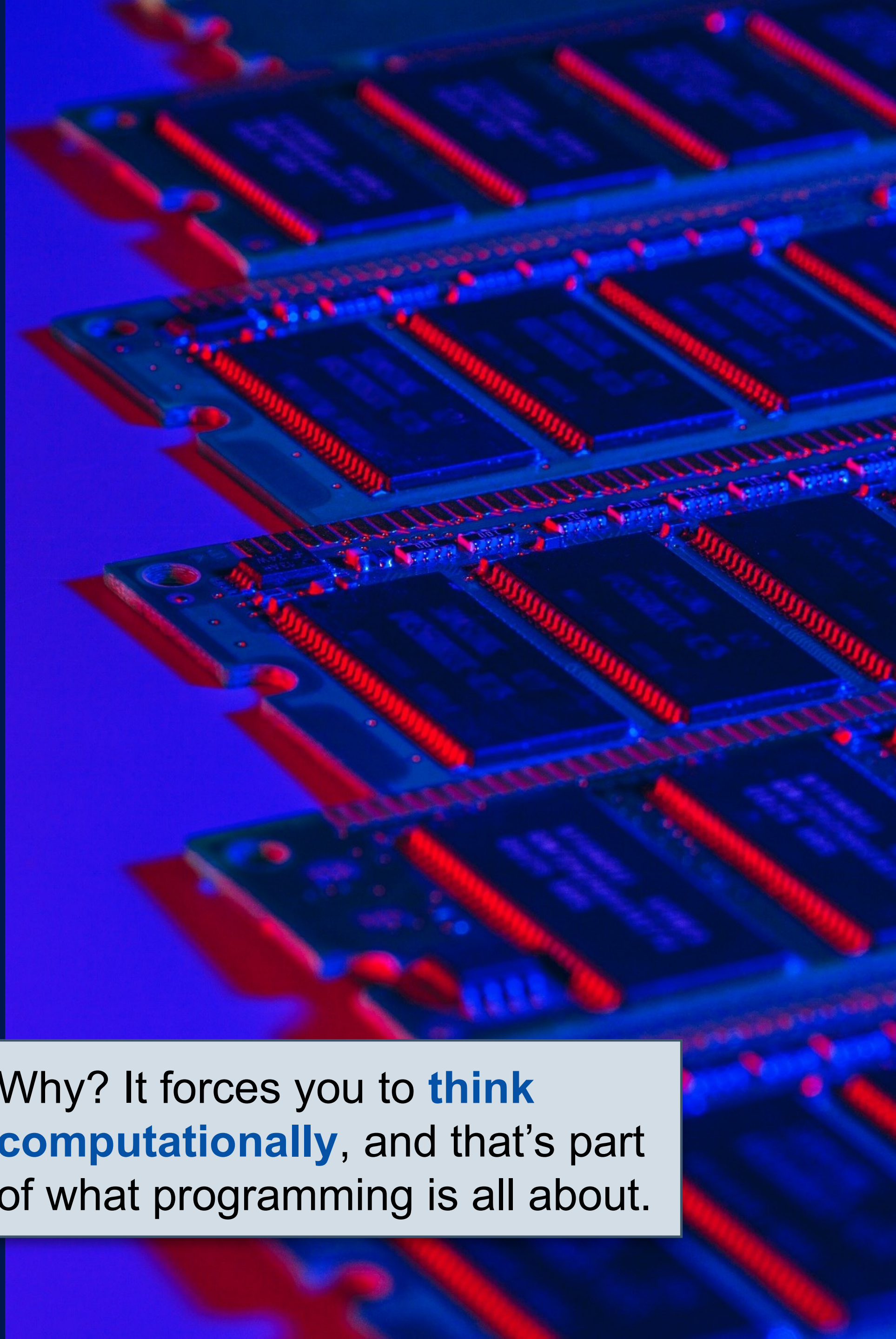
```
int* ourPointer = new int[100];
delete[] ourPointer;  // Deallocate, no problem!
ourPointer = nullptr; // Indicate "This shouldn't be used."


// At some point later in our code...
if (ourPointer == nullptr)
{
       cout << "No data allocated, reallocating..." << endl;
       int count;
       cin >> count;
       ourPointer = new int[count];
}
```

The pointer is now **`invalid`**.



You Must Delete Dynamic Memory

- Some languages have **memory management** systems.
  - Variables that are no longer used or referenced get flagged as **garbage**.
  - Periodically, the **garbage collector** frees that memory.
  - These systems are done for the programmer automatically.
- C++ does not have a system like this – you have to clean up after yourself.
  - Some newer features mimic this, to an extent.
- It can be a massive headache but learning to do so can make you a very effective programmer.
  - Even in situations where dynamic memory isn't a thing.



Why? It forces you to **think computationally**, and that's part of what programming is all about.

Many Built-In Classes Use Dynamic Memory

The vector class

```
vector<int> numbers;
numbers.push_back(5); // Just one

vector<int> words;
moreNumbers.push_back(10);
moreNumbers.push_back(20);
moreNumbers.push_back(-14); // Store 3 numbers
```

How does the class store this data internally? What is the class declaration like?

```
class MyVector
{
        // Some call it wasteful, I call it "prepa
        int data[1000000000];
};
```

1. This is an array 4GB in size!
2. This won't fit on the stack!
3. Do you **need** that many integers? (In most cases: no)

Yes, that is 1 billion elements…

No, this is not a good idea…

Surely there must be a better way? (There is!)

Alternatives to Enormous Arrays

```
class MyVector
{
        // Wasteful
        int data[1000000000];
};
```

```
class MyVector
{
        // May not be enough
        int data[10];
};
```

What you really want:

```
class MyVector
{
        // Perfect! But, how to accomplish this?
        int data[AsManyAsYouNeedRightNow];
};
```

With dynamic memory allocation!

A Better Approach

```cpp
class MyVector
{
        int* data;
        unsigned int numberOfEle
        unsigned int capacity;
public:
        MyVector(int initialSize);
        ~MyVector() { delete[] data; } // Destructor
};

// Constructor
MyVector::MyVector(int initialsize)
{
        data = new int[initialsize]; // Create just as much data as we need
        numberOfElements = 0;         // We aren't using any of it just yet
        capacity = initialsize;               // How many elements COULD we use?
}

MyVector numbers(10);                         // Capacity of 10
MyVector moreNumbers(3);                      // Capacity of 3
MyVector whoaLotsOfNumbers(50000);    // Capacity of 50,000
```

Pointers, the secret sauce

Variables for tracking the capacity and amount of data (because pointers don't have any of this).

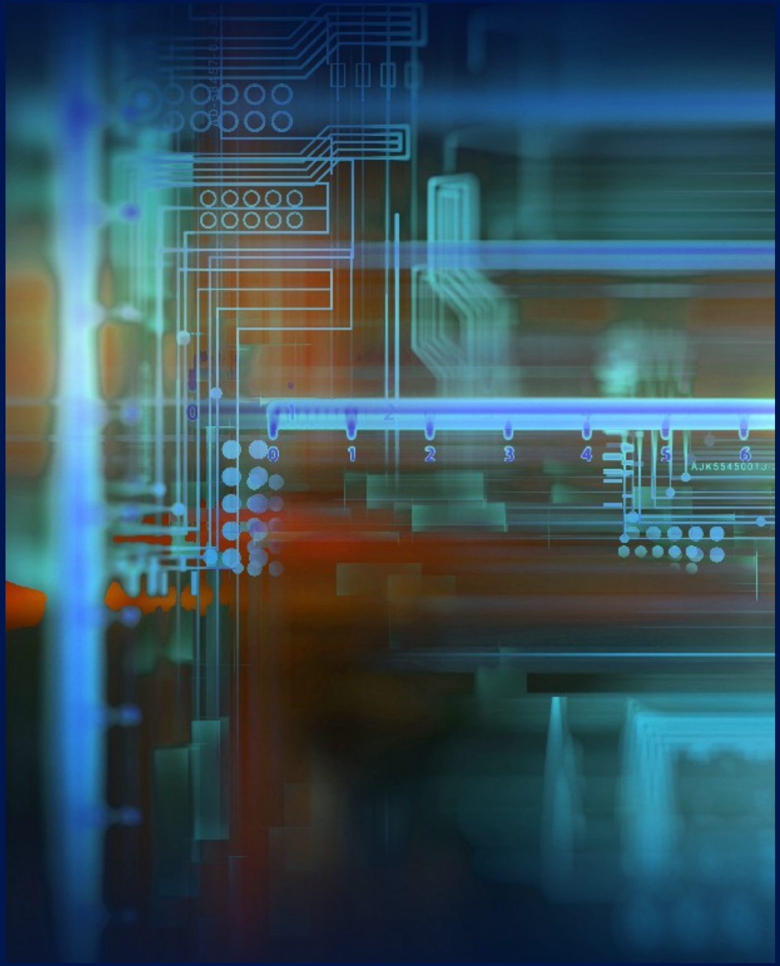
This is a rough approximation at the moment.

There is still a lot more to cover about all of this: templates, destructors, deep-copying, the Big Three… we'll get there soon enough!

Recap

- Dynamic memory allows your program to **allocate resources on the heap**, instead of the stack.
  - We might need more memory than can fit on the stack.
  - We might need information that isn’t known at compile-time.
- We **allocate**, or reserve, memory with **`new`**.
  - We have to “catch” this with a pointer.
- We **deallocate**, or free memory with **`delete`**.
  - We call `delete` on a pointer to the allocation.
- If memory isn’t deallocated properly, we get **memory leaks**.

Conclusion



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

Thank you for watching.