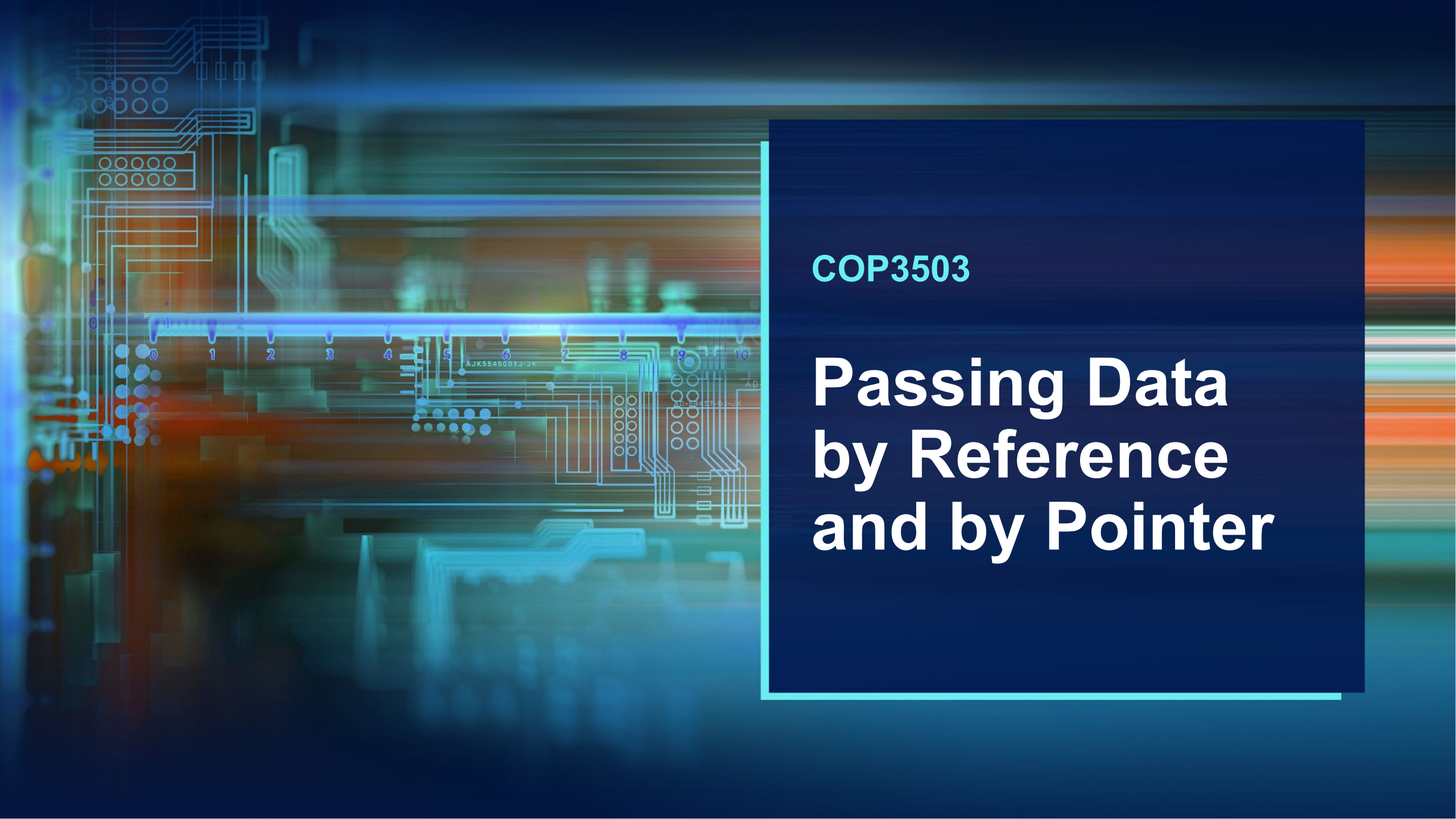
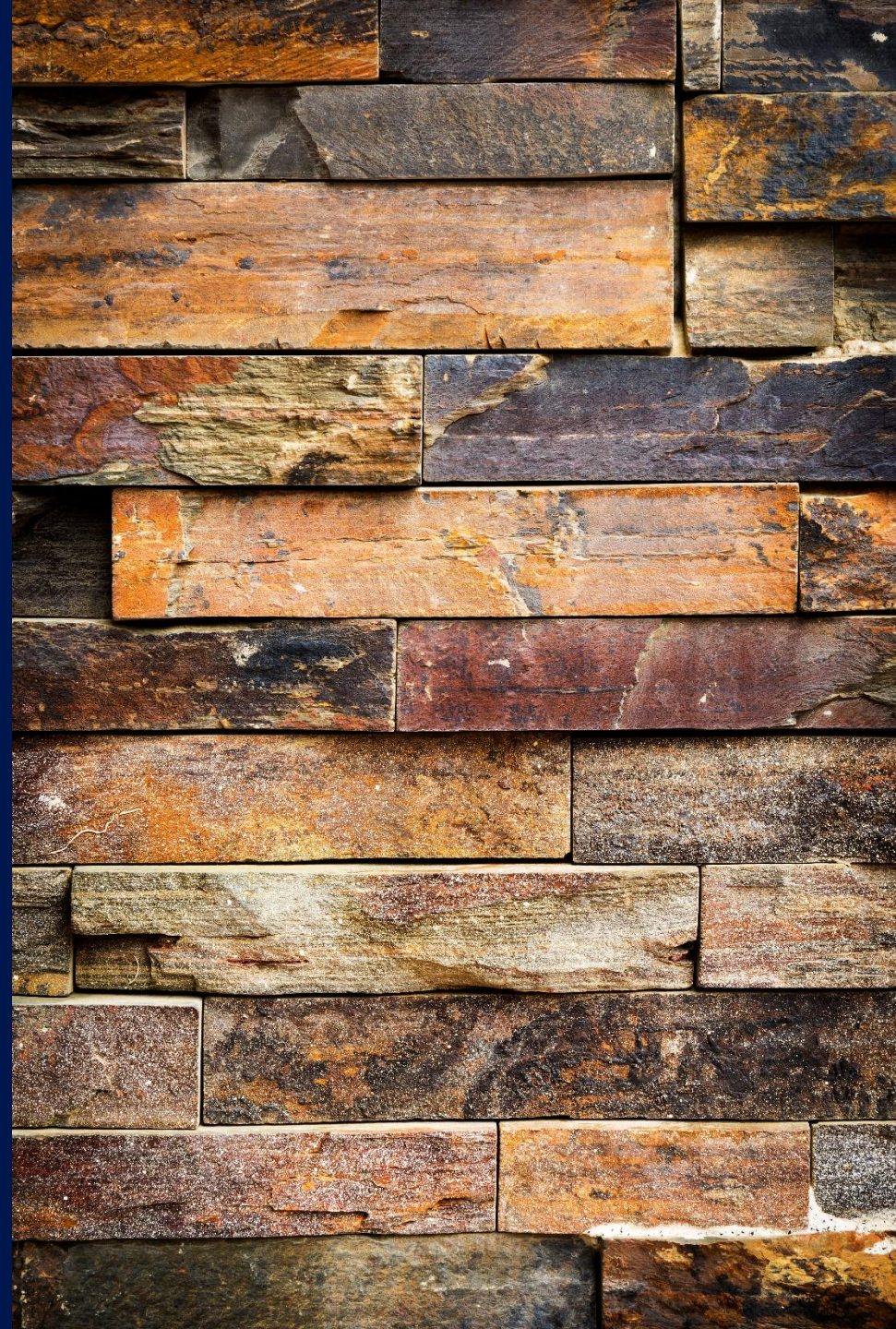


COP3503

Passing Data by Reference and by Pointer

The Word “Stack” in Different Situations

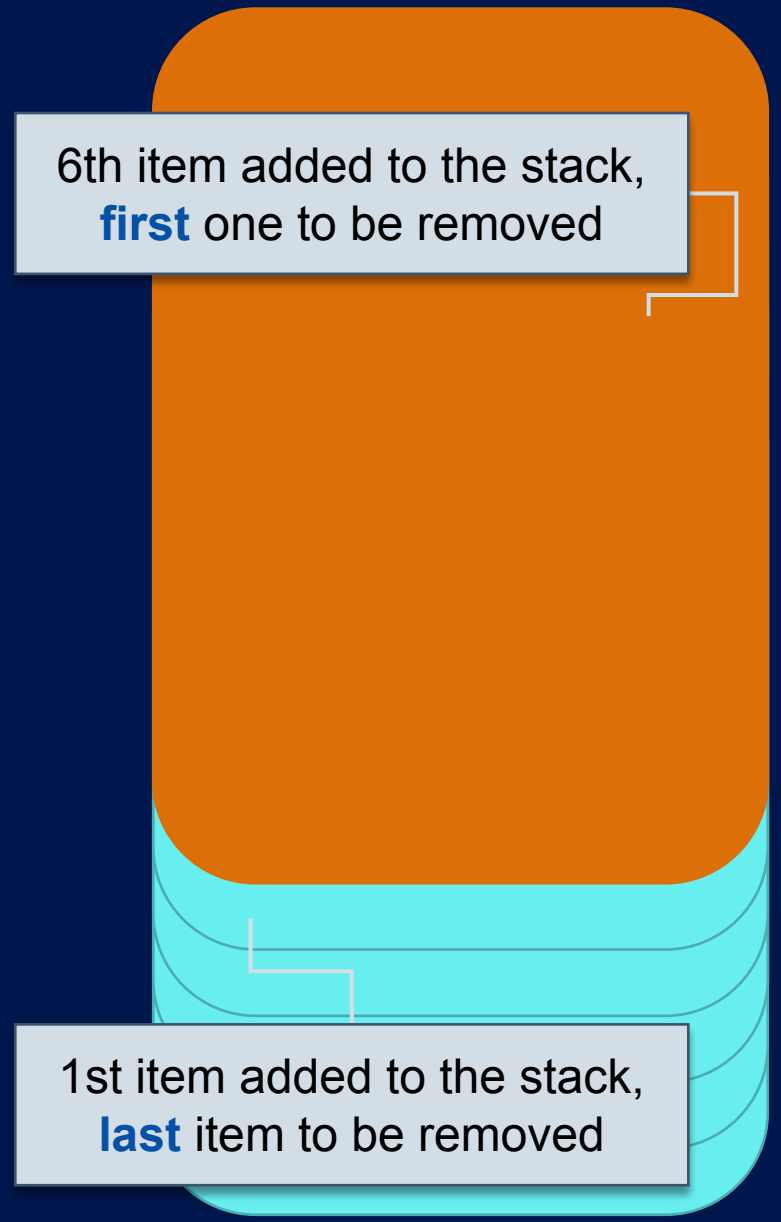
- A location in memory tracking stack frames
  - The Stack, part of the amazing duo, Stack and Heap
- A list of functions currently executing in a program
  - The call stack, which is stored on The Stack
- A last-in, first-out data structure
  - `class Stack {};` (More on this later)



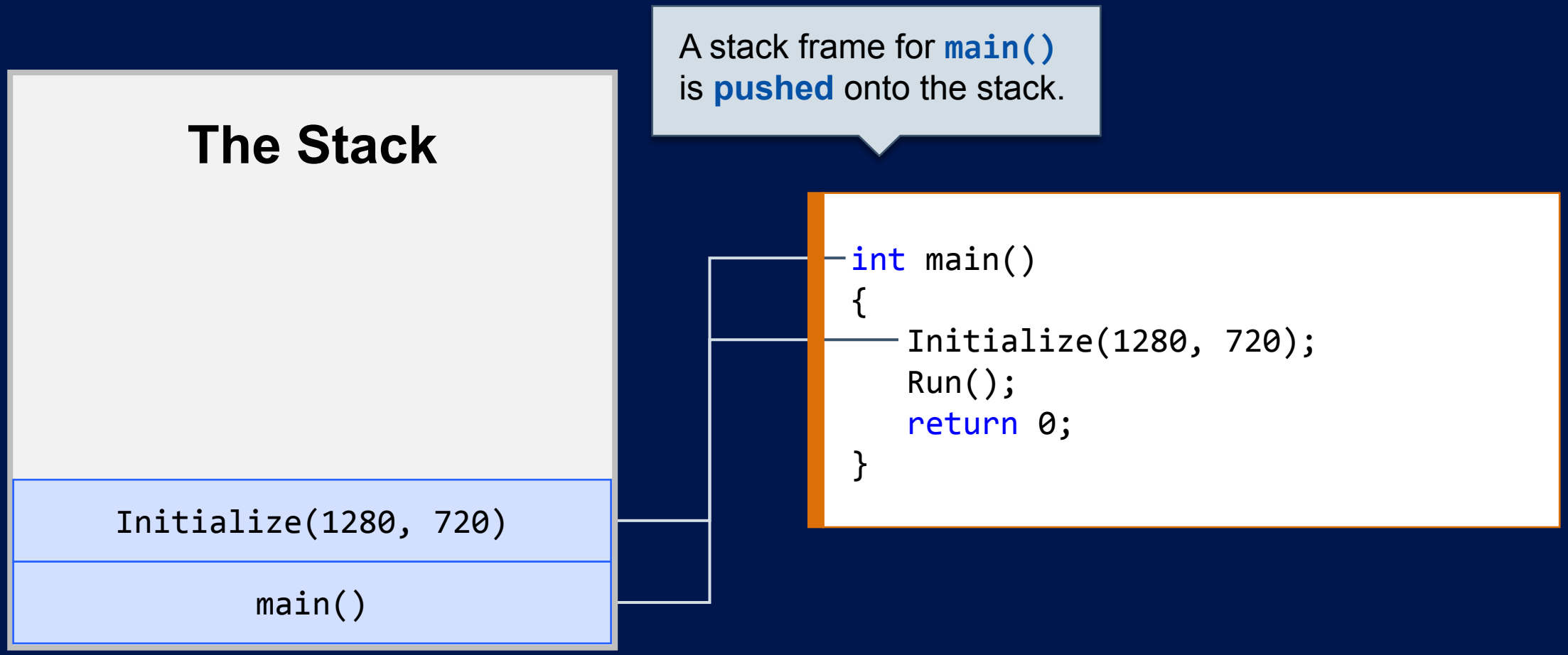
The Stack

Summary

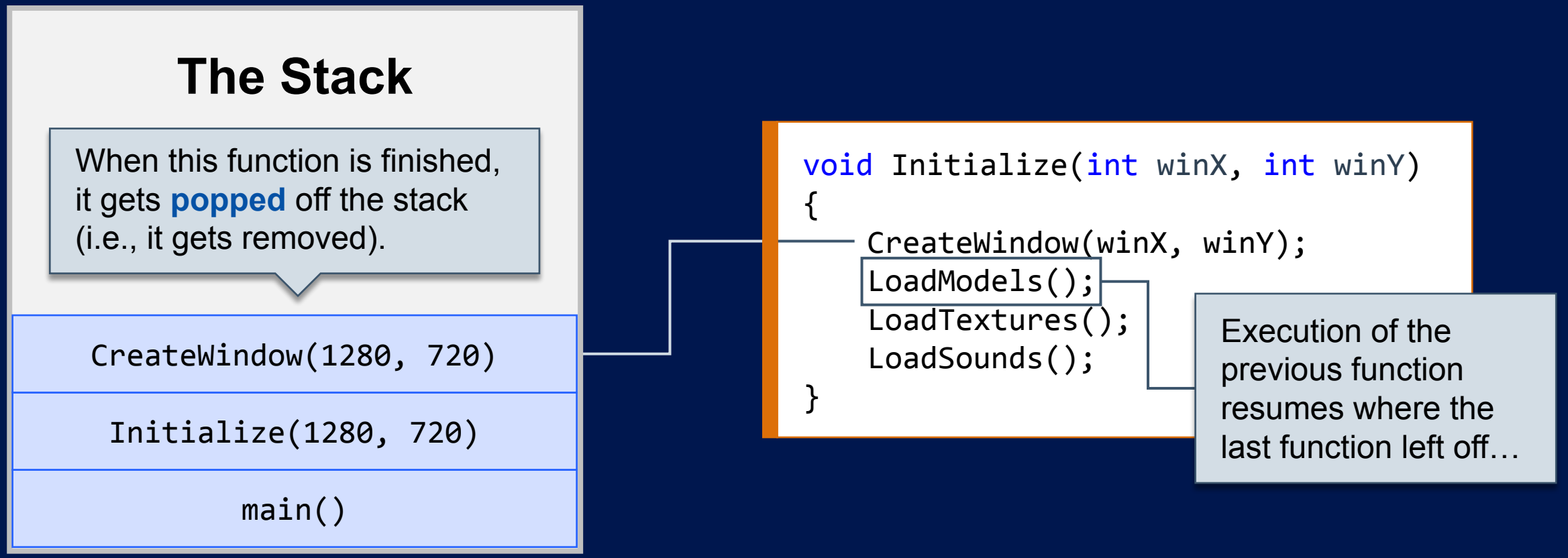
- The **stack** is a region of memory that tracks **stack frames**.
- A **stack frame** is a chunk of memory that contains the information necessary to execute a function:
  - Memory for variables created in that function
  - Memory for passed-in parameters
  - Memory address of previous stack frame
- The ordering of a stack is **last-in, first-out**.
  - Similar to a stack of books, or a deck of cards



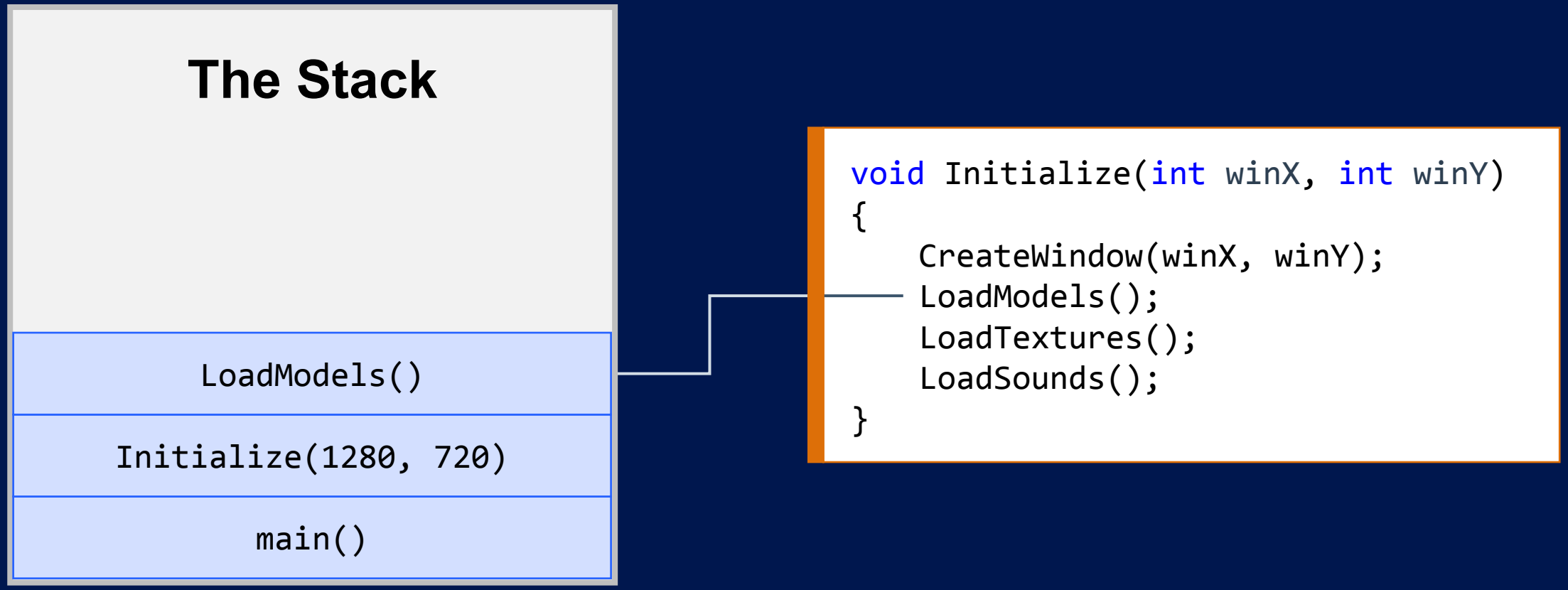
Call Stack Example



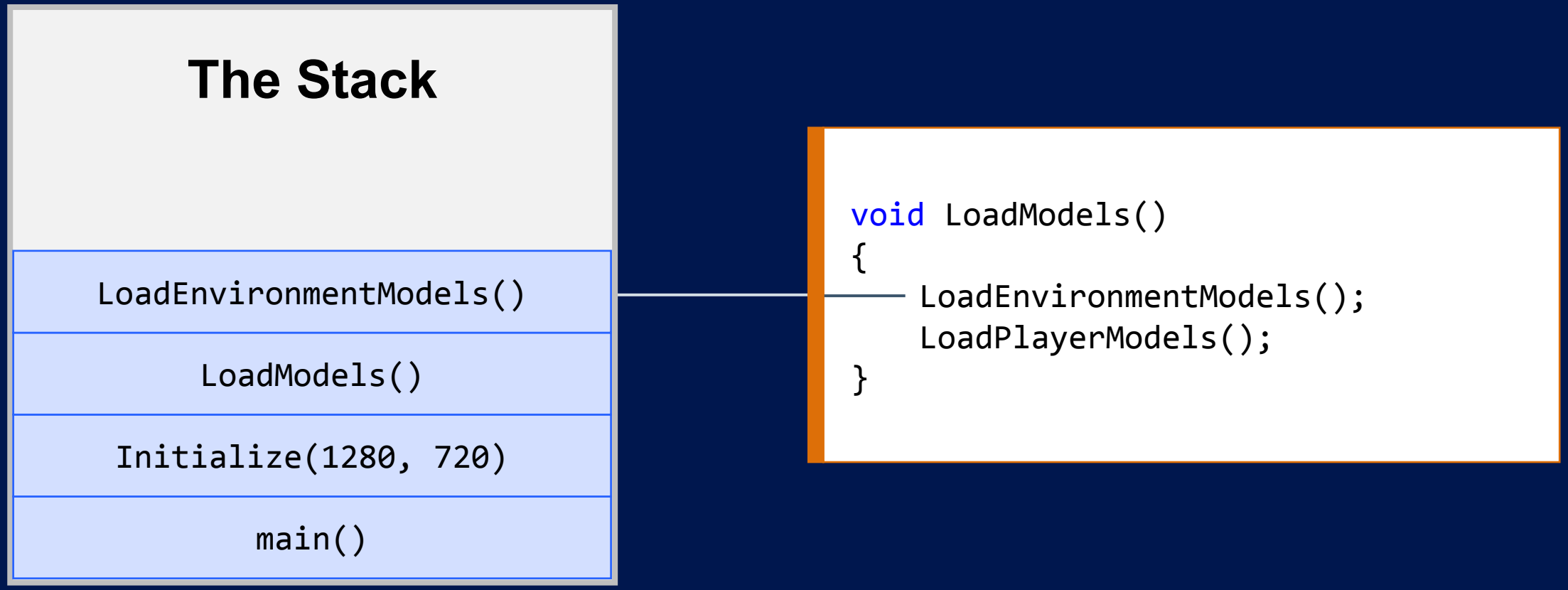
Initialize() Executes...



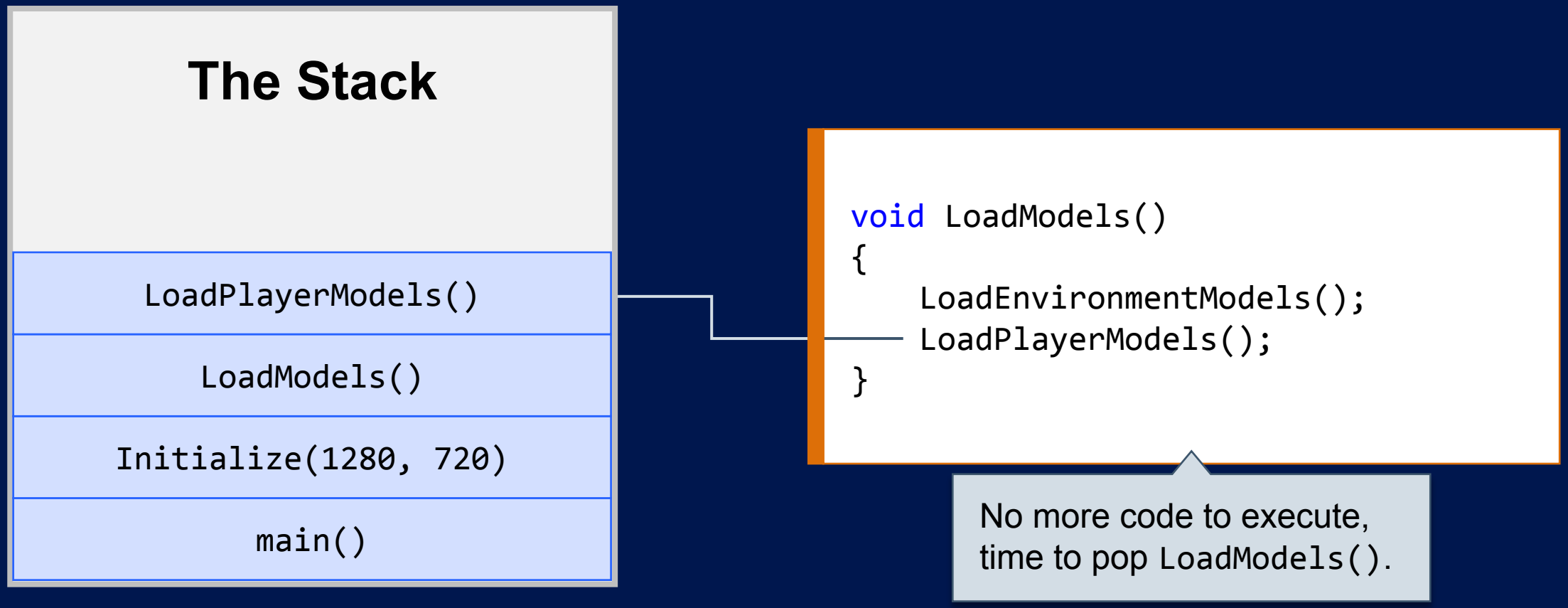
Continuing On…



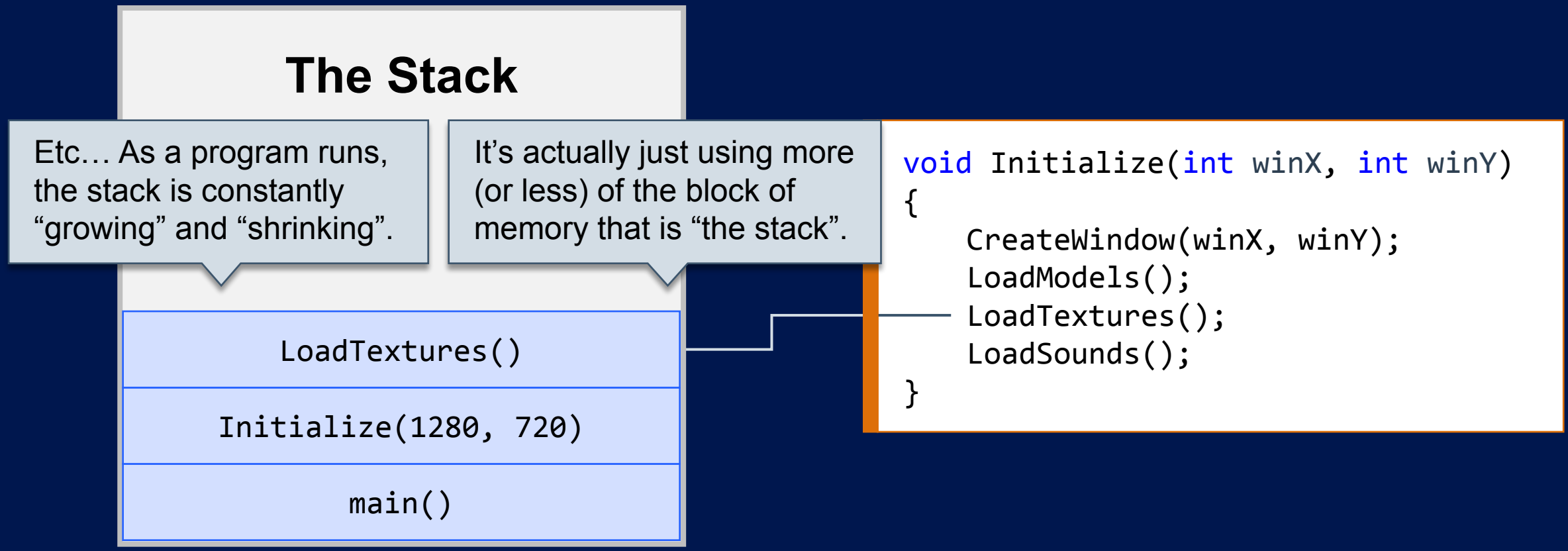
Loadmodels() Is Put on the Stack and Executes…



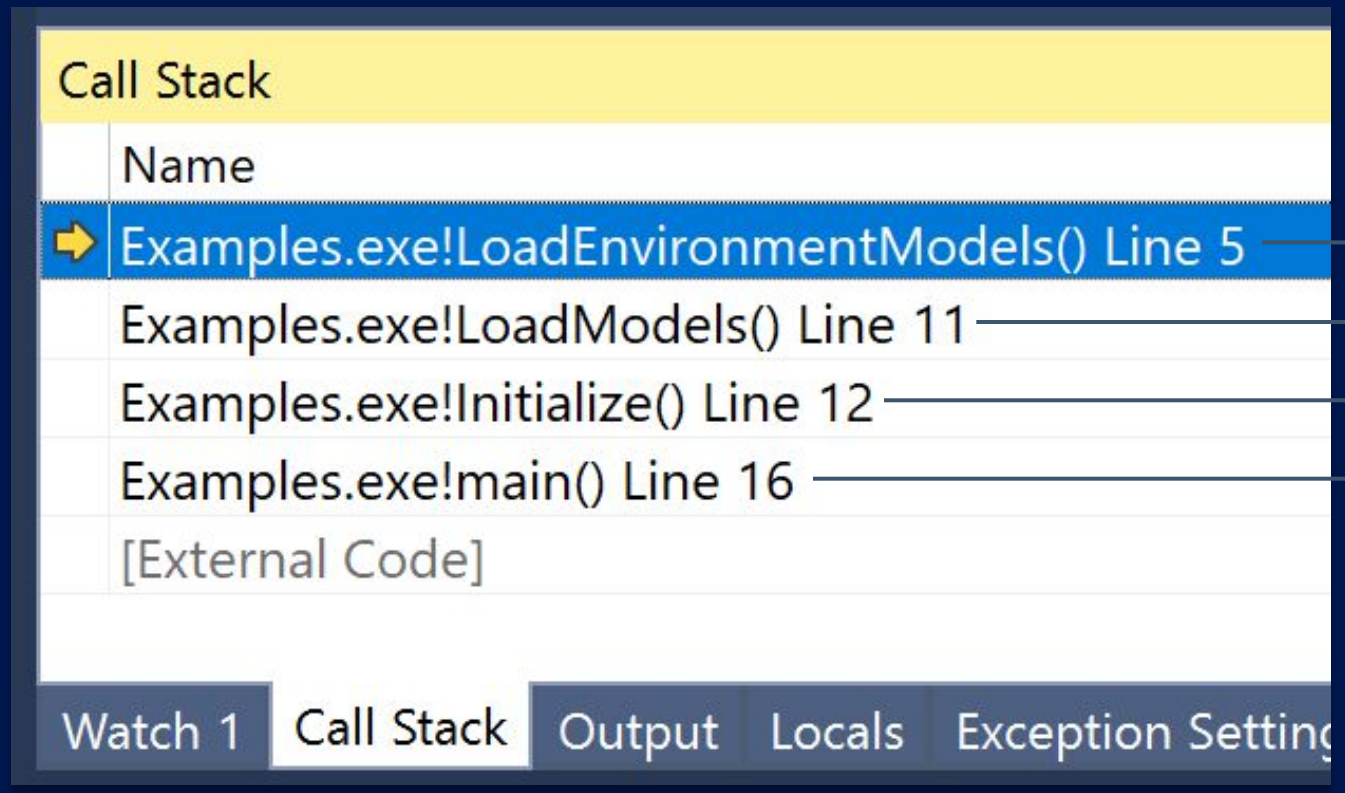
Loadmodels() Continues...



Loadmodels() Is Finished, Move Down the LIne



The Call Stack in an IDE



The current function…

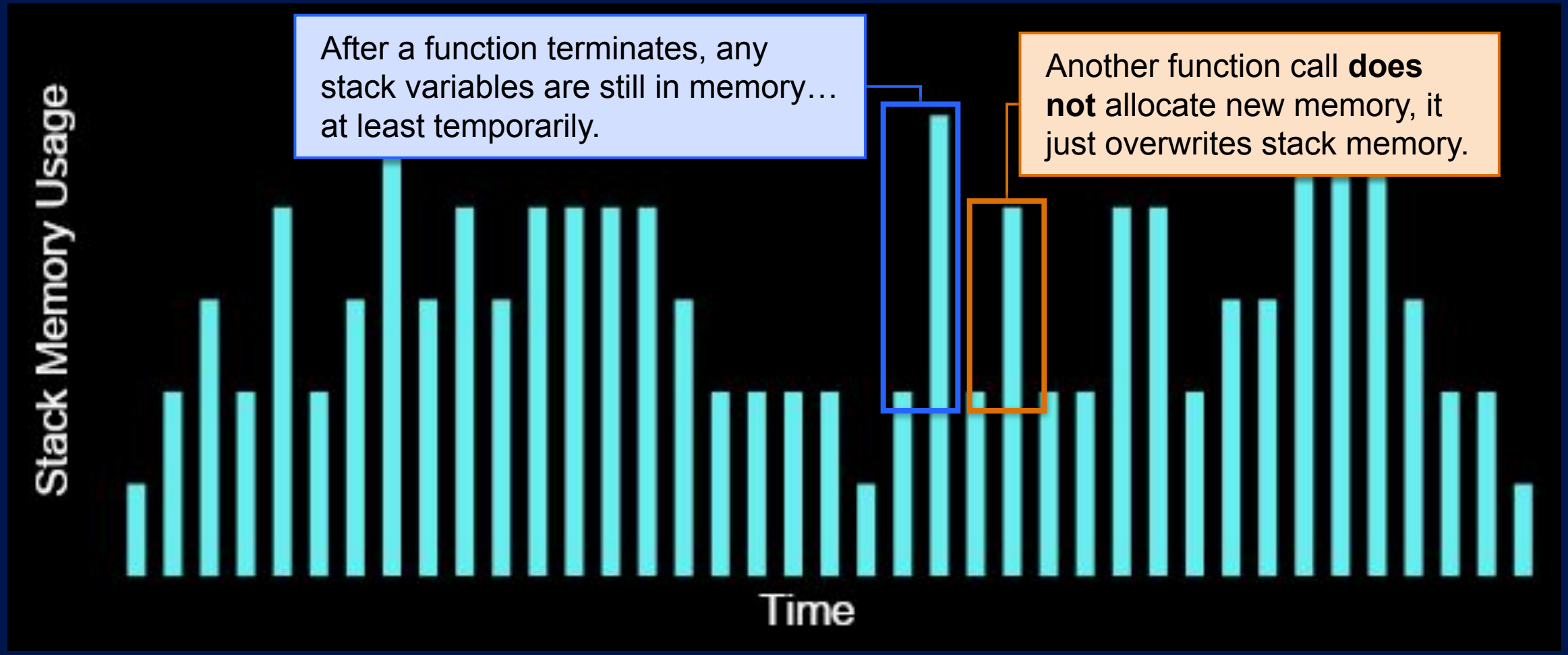
…which was called by this function…

…which was called by this function…

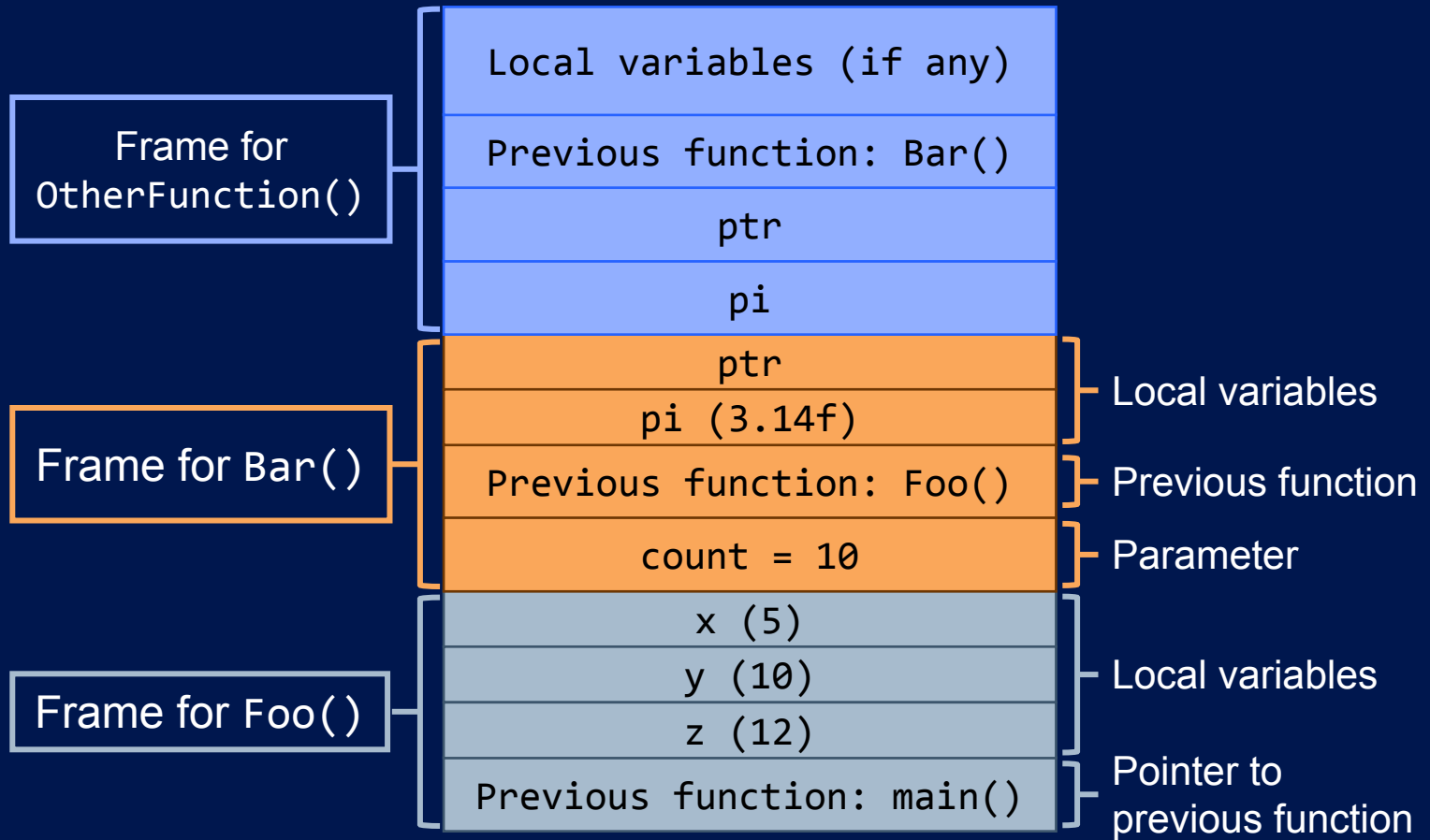
…which was called by main().

The call stack provides information about the order of operations

Stack Memory Is “Churned” Over Time

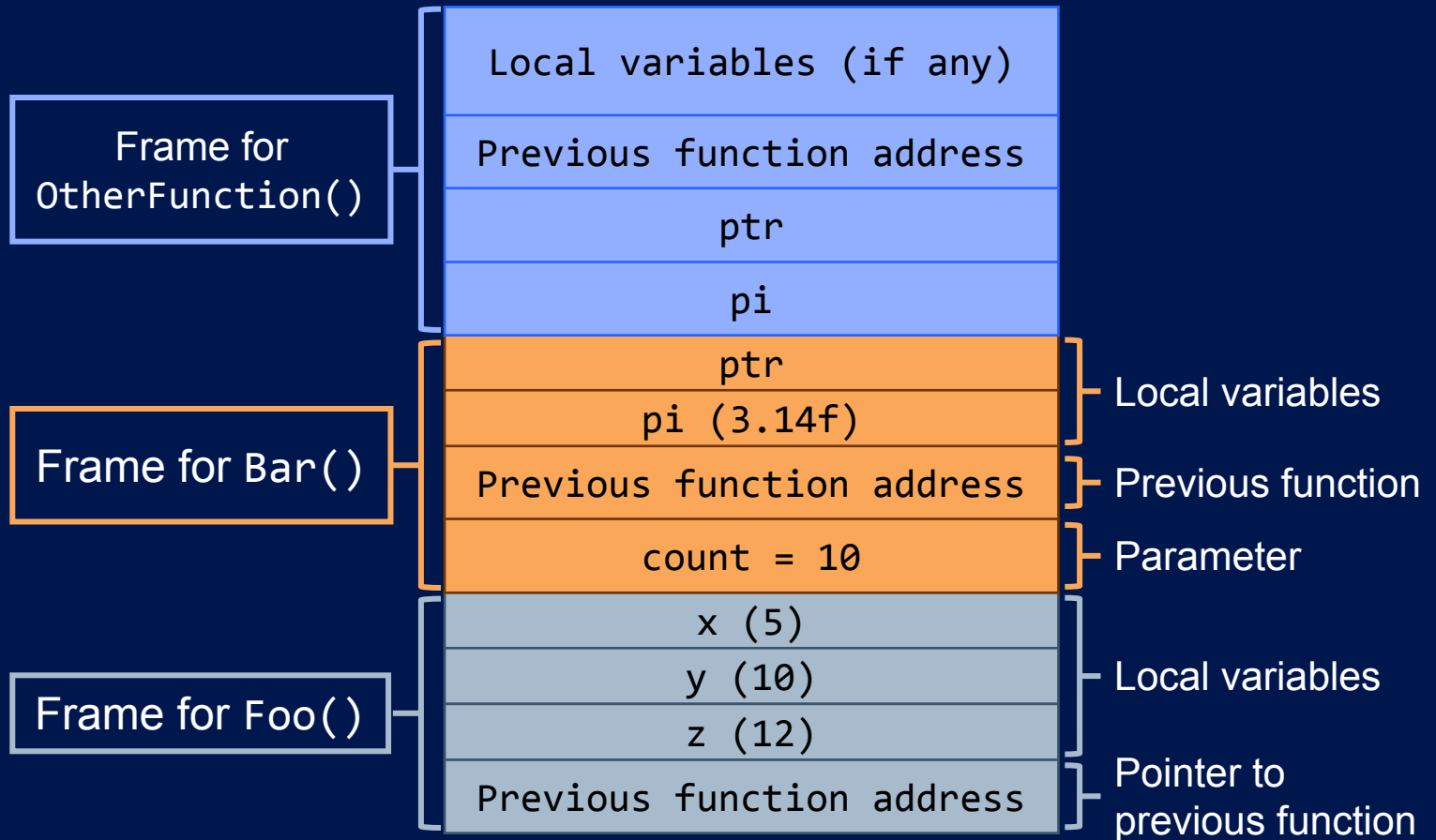


Stack Frame Visualized




```
void Bar(int count)
{
    int* ptr = new int[count];
    double pi = 3.14f;
    OtherFunction(pi, ptr);
}
void Foo()
{
    int x = 5;
    int y = 10;
    int z = 12;
    Bar(10);
}
int main()
{
    Foo();
}
```

Stack Frame Visualized

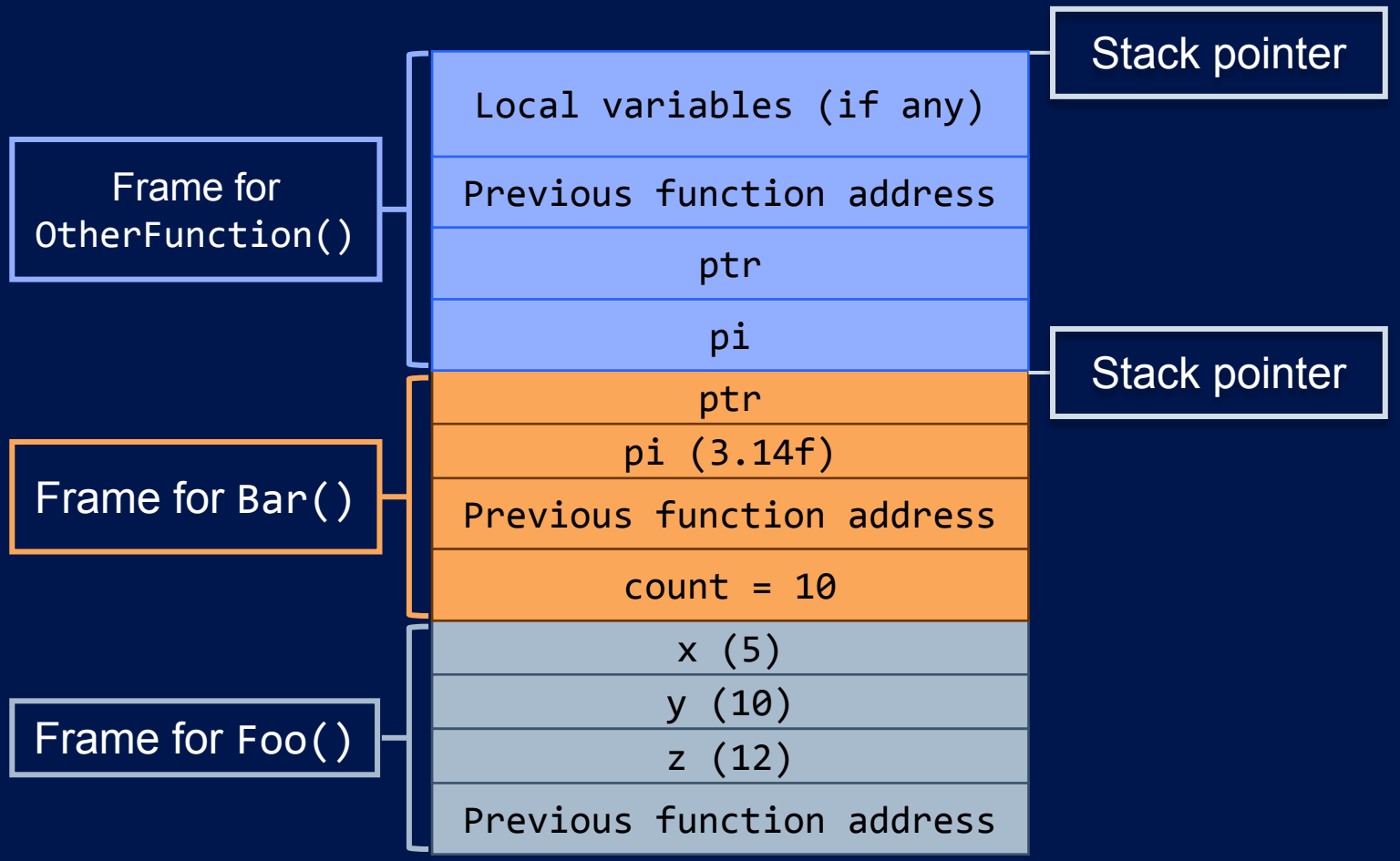


Size of a stack frame:
The size of any variables declared, plus pointers to previous functions, plus the size of parameters

The size is determined by the compiler during the build process.

At runtime, these stack frames get pushed or popped.

Stack Frame Visualized

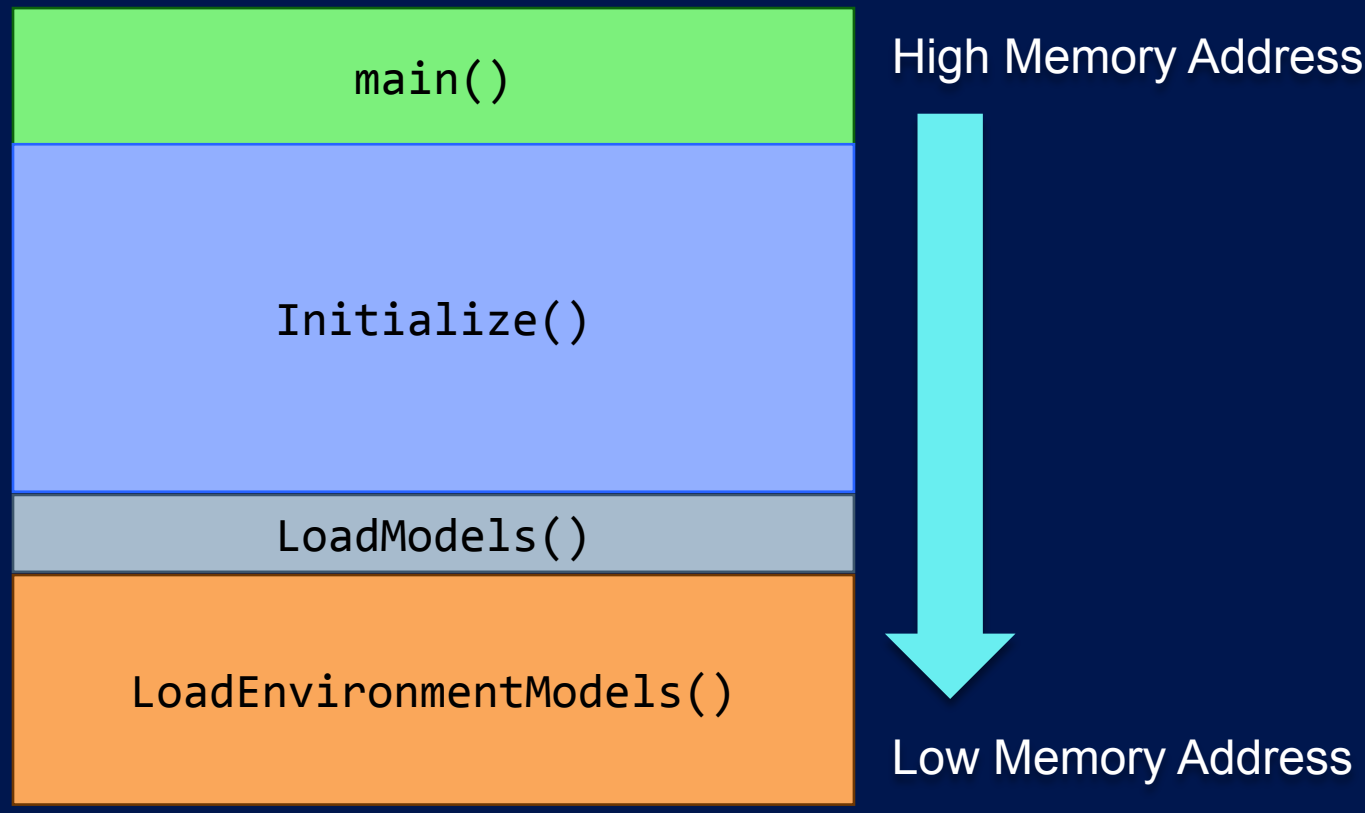


A **stack pointer** points to the current stack frame.

The pointer moves to different addresses as functions are called (or as they end).

Moving this pointer is very fast, and takes almost no time at all

Stack Memory Allocations May Be Reversed



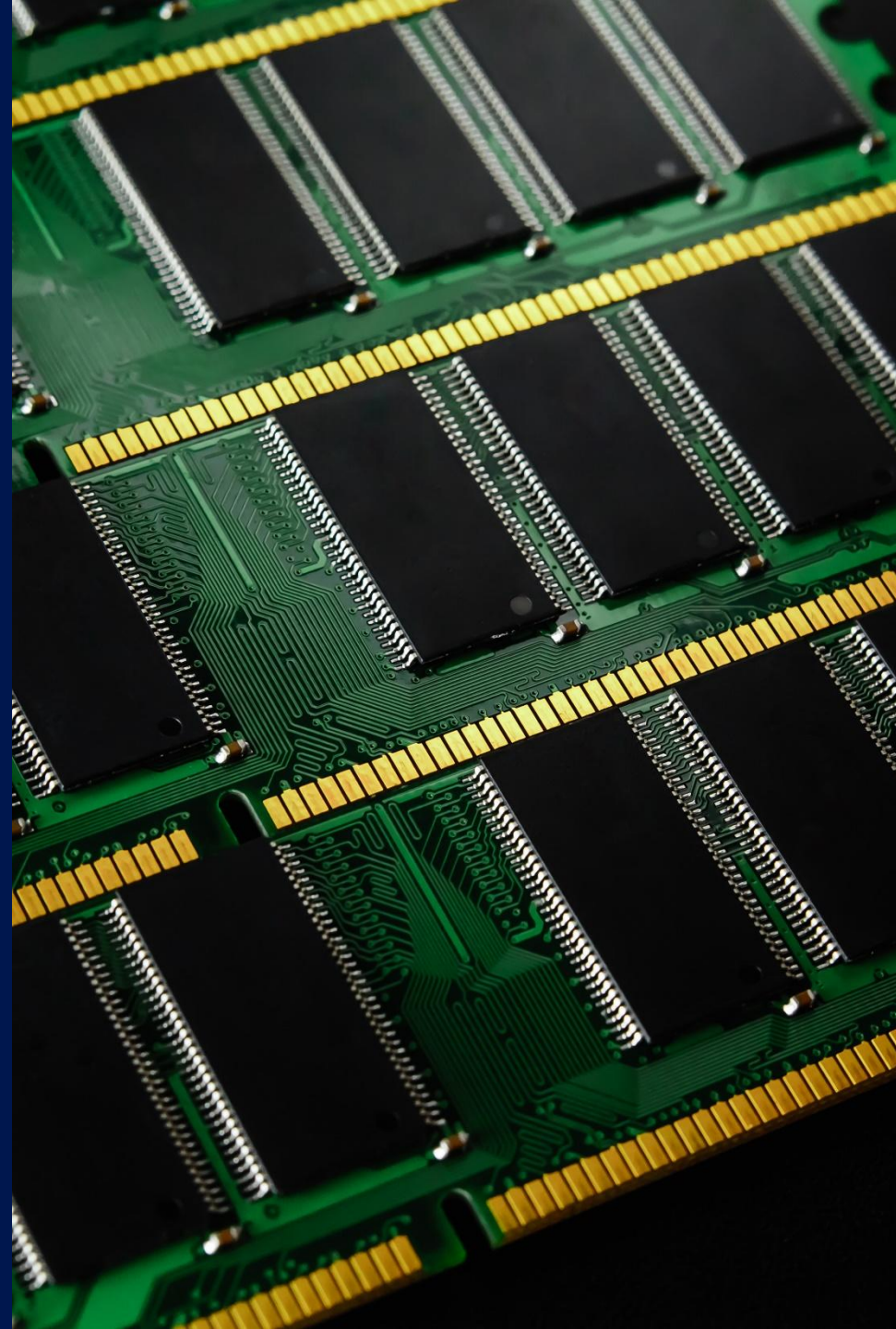
We may think of a stack as growing “up” (i.e., stacking books).

In practice it **may** be the opposite: the stack grows “down”.

Subsequent frames are added to lower addresses.

Stack Memory Is Limited

- Depends on the operating system, possibly compiler settings
- Typically small, only a few megabytes in size
  - You can still create a **lot** of data in a few megabytes
  - 1 megabyte can be ~250,000 integers, ~125,000 doubles.
- This can affect how you create objects in your code.
- Too much data on the stack? You can cause a **stack overflow** error.



Stack Overflow Error

- Too much memory allocated on the stack
- Every variable occupies **some** amount of memory.
- Every function call occupies some amount of memory.
- It all adds up!

Water

Pour a gallon of water into a glass that holds less than a gallon…

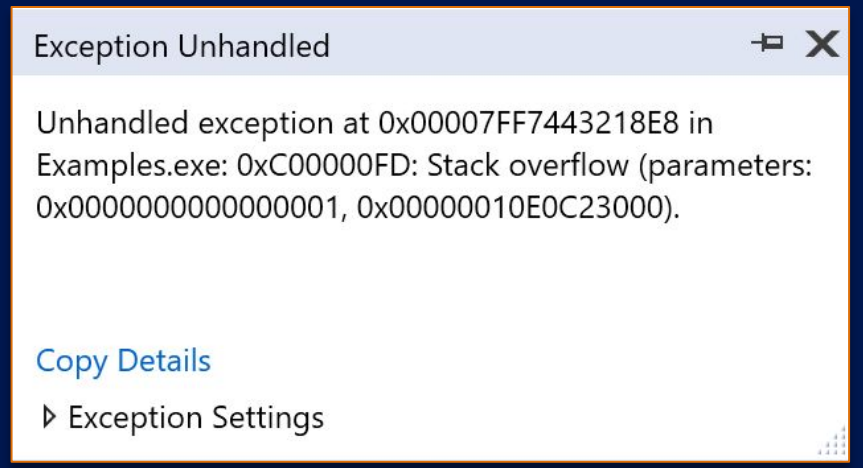
Stack Overflow

Having 1 large variable or many smaller variables doesn't matter: Try to use too much memory, and your program breaks.

```
int main()
{
    int array1[75000];
    int array2[75000];
    int array3[75000];
    int array4[75000];
    return 0;
}
```

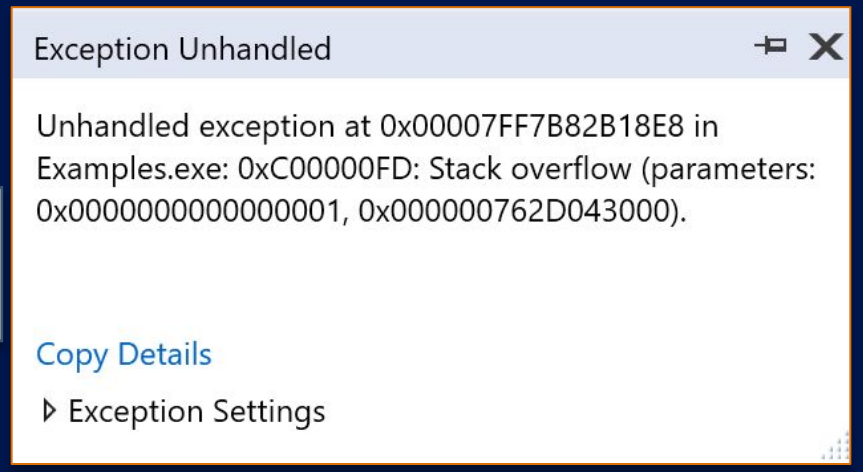
1 int is typically 4 bytes
4 * 75000 = 300,000 bytes, or 300 kB

300 kB * 4 = 1.2MB, larger than the stack in this example




```
int main()
{
    int bigArray[300000];
    return 0;
}
```

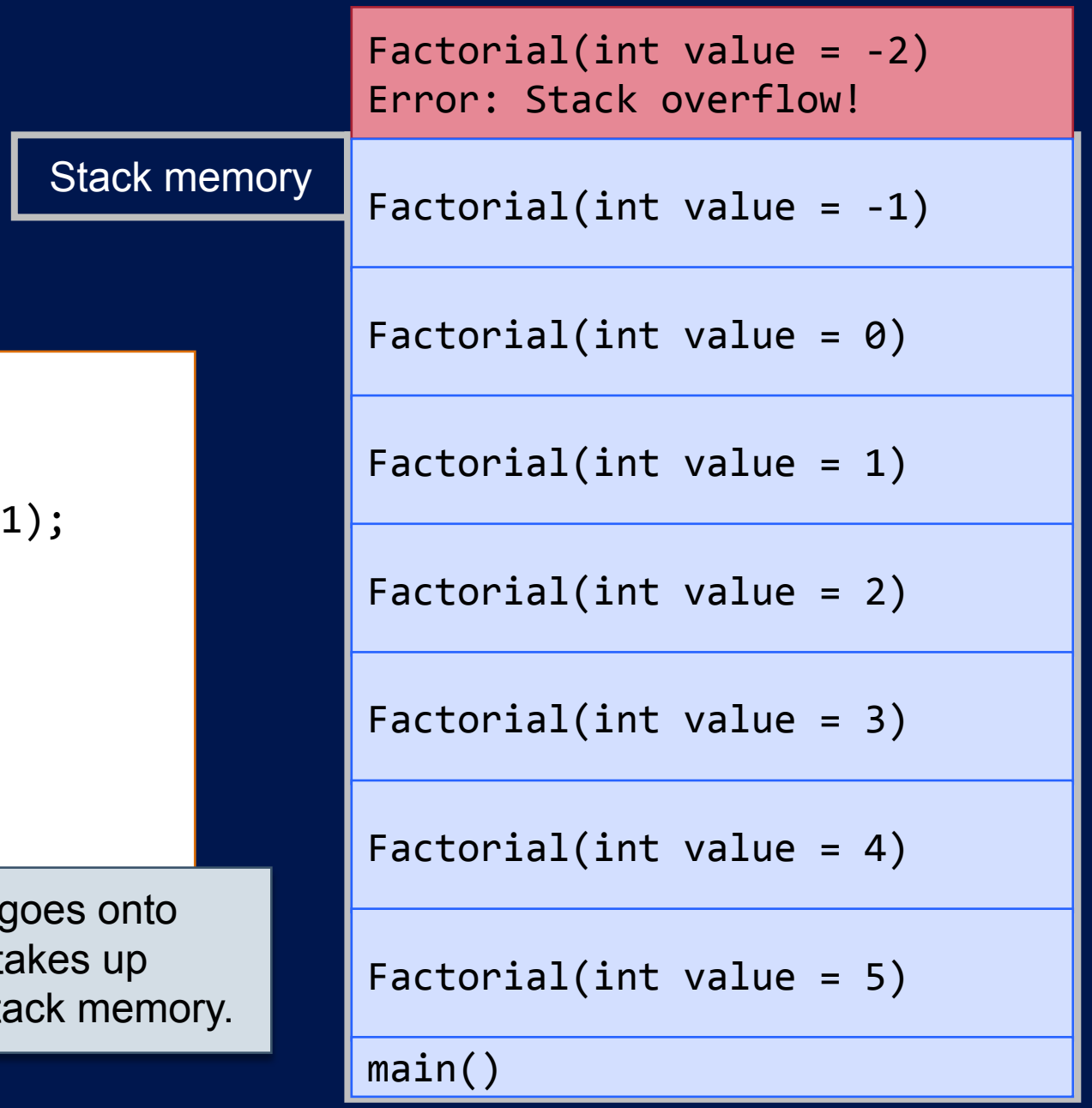
Same number of integers, same memory footprint



Stack Overflow From Recursion

```
int Factorial(int value)
{
    return value * Factorial(value - 1);
}

int main()
{
    cout << Factorial(5);
    return 0;
}
```

Each function call goes onto the call stack and takes up some amount of stack memory.



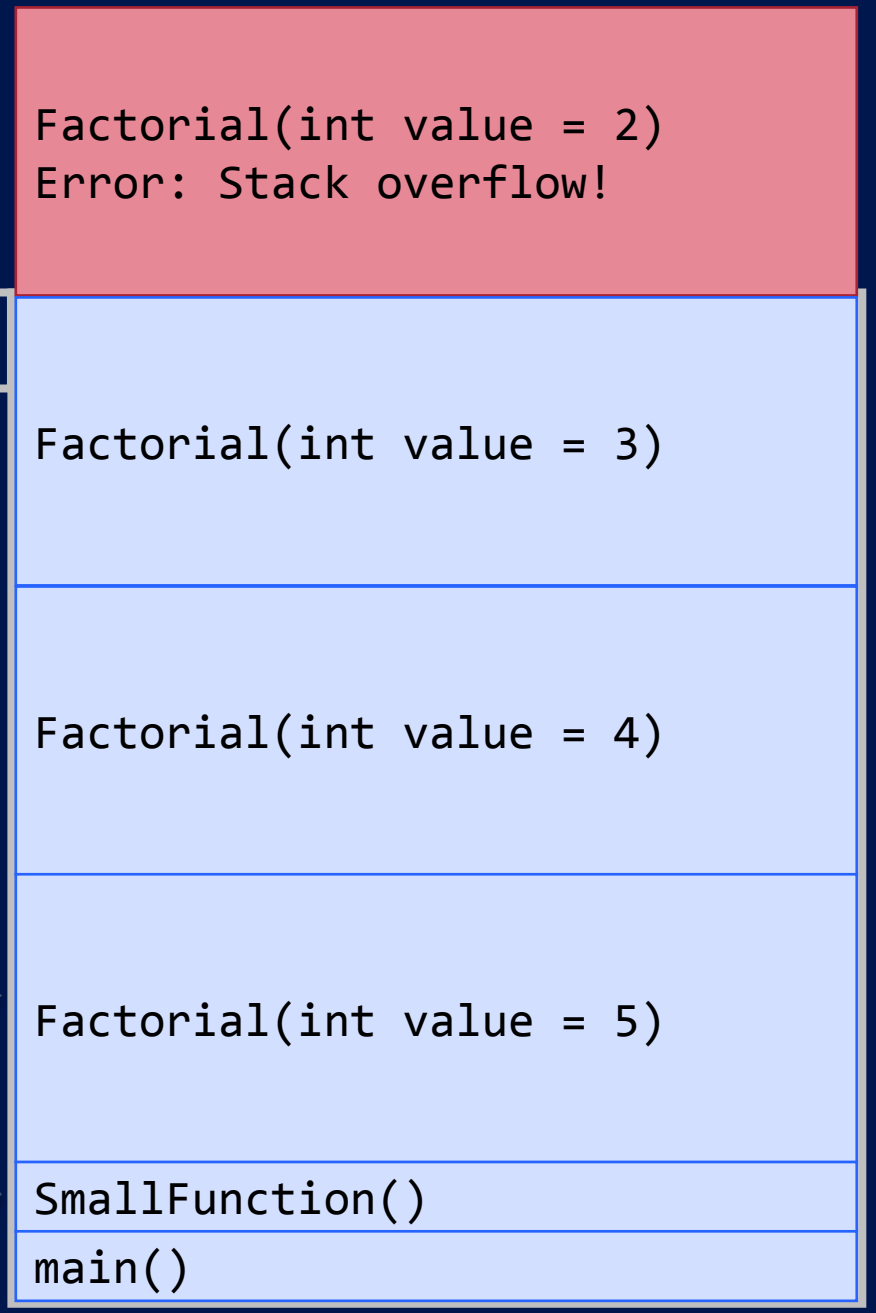
The Stack Is a Small Amount of Memory

```
int Factorial(int value)
{
    int bigArray[1000];
    return value * Factorial(value - 1);
}
void SmallFunction()
{
    Factorial(5);
}
int main()
{
    SmallFunction();
    return 0;
}
```

Functions with lots of data, on the other hand…

…use more space on the stack.

Stack frames are typically small; a call stack might store dozens, or hundreds of them.



What If You Need Large Amounts of Memory?

- Then, you **dynamically allocate** memory on the **heap**.

Automatic storage

```
void SomeFunction()
{
    int bigArray[300000];
    // Do some stuff
}
```

Dynamic storage

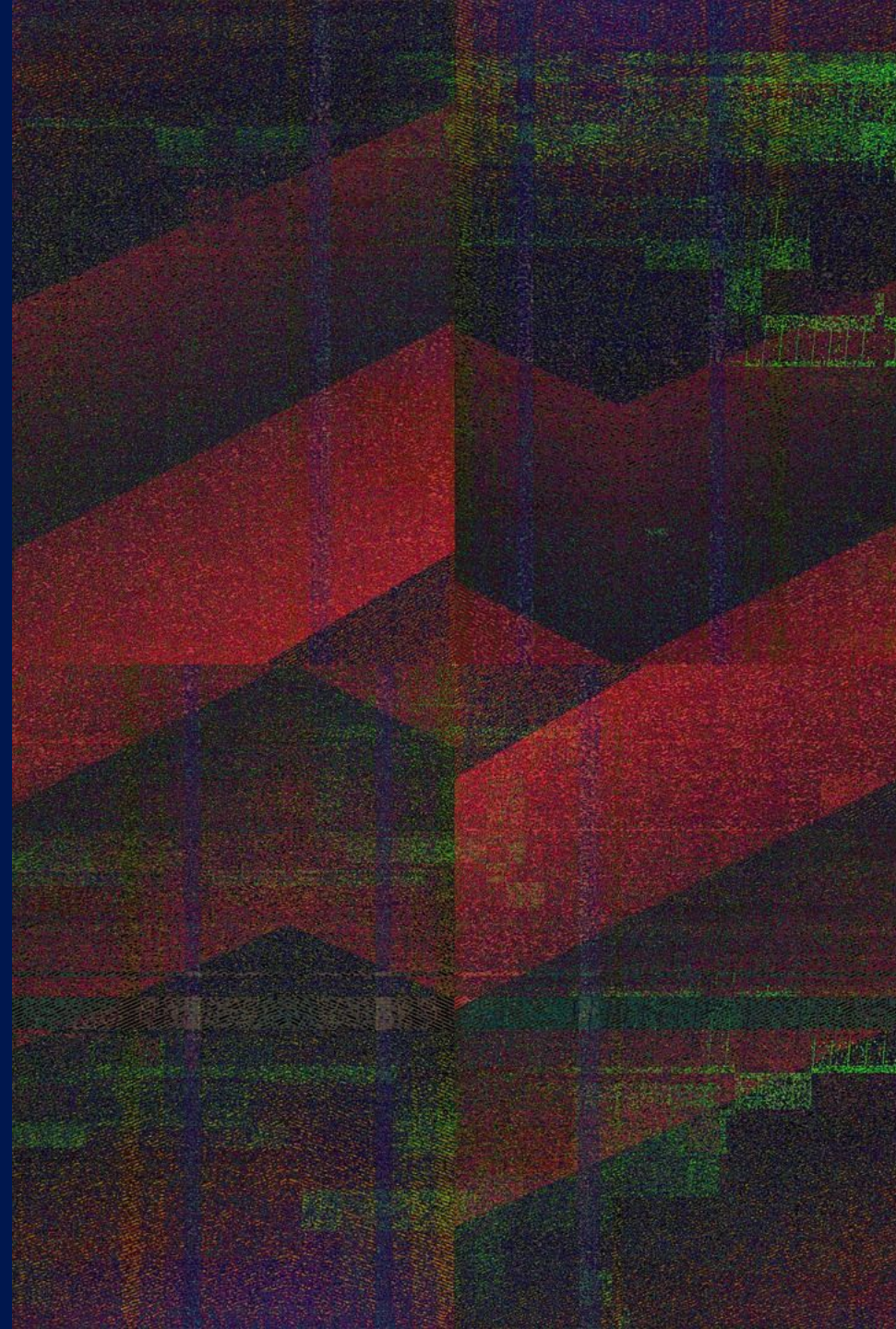
```
void SomeFunctio
{
    // 1. Allocate memory
    int* bigArray = new int[300000];

                    you need to do

                    te memory
                    ray;
}
```

This data is allocated on the heap instead of the stack.

bigArray is on the stack (all local variables are), but it **points** to memory on the heap.

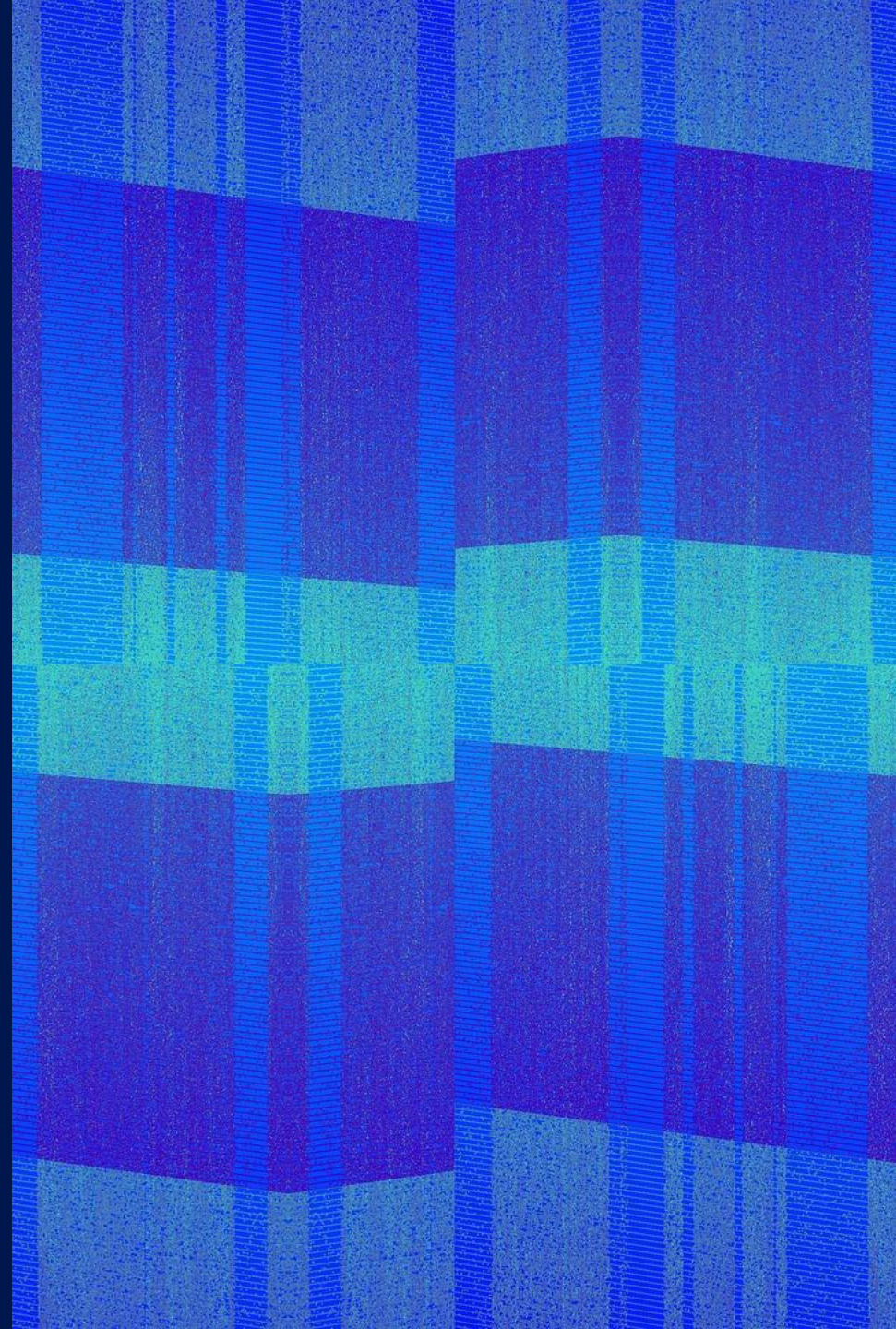
The Heap Is for Dynamic Memory

- The heap is for data we can't put on the stack:
  - Large amounts—no stack overflow!
  - Data we don't want to "fall out of scope"
  - Data whose size we don't know at compile time
- Heap memory is allocated, or reserved with `new`
  - It **must** be deallocated or freed with `delete`.
- Much slower than the stack
  - But it must be used in many cases.
  - Use the stack when you can, the heap when you must.

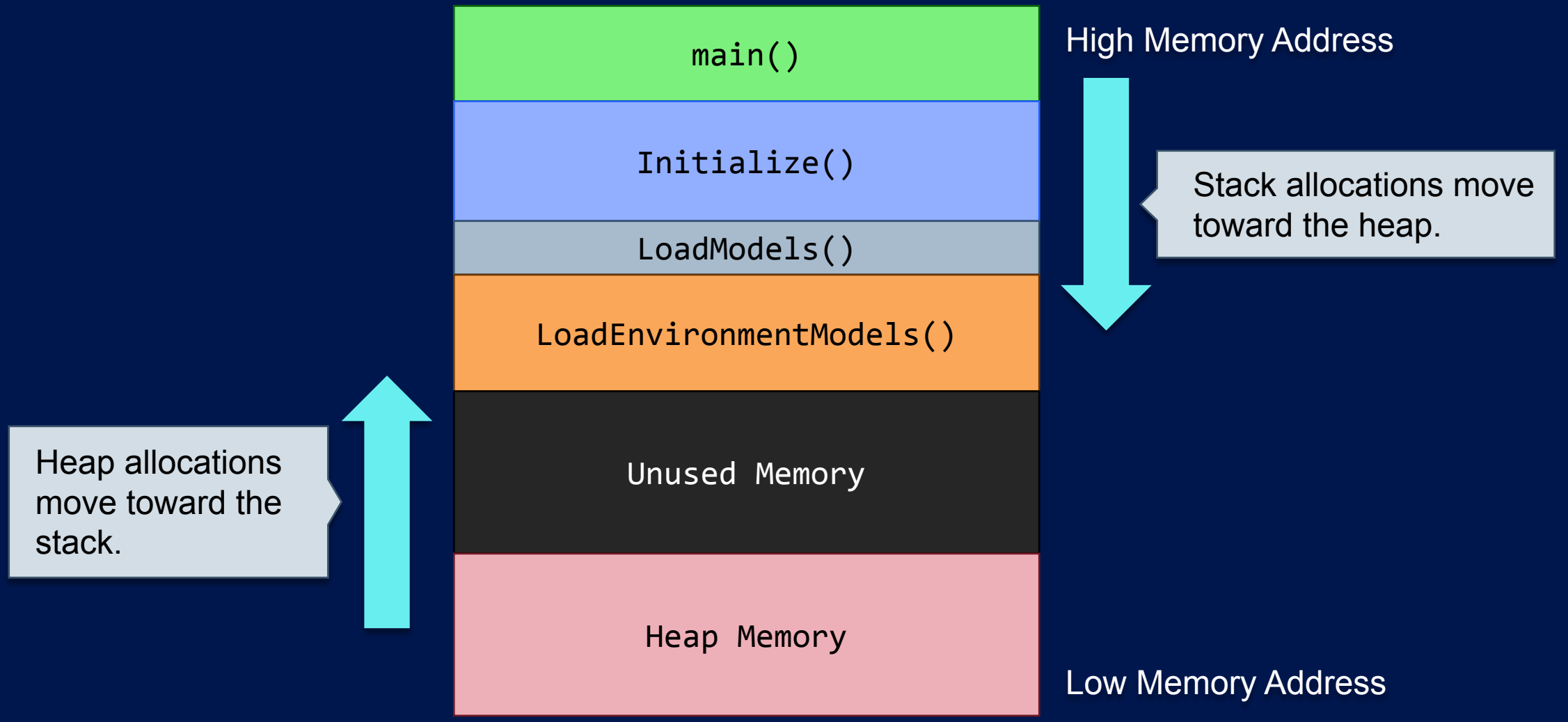


Heap and the “Free Store”

- Heap and Stack are general computing concepts.
- The C++ standard doesn’t mention “heap” and “stack” anywhere.
  - You may hear the heap called the **free store**.
- The C++ standard uses **automatic** and **dynamic** storage.
- Conceptually, the same thing:
  - **Stack**: Automatic storage
  - **Heap**: Dynamic storage



Heap and Stack Memory Grow Toward Each Other



Stack Variables vs. Heap Variables

- All local variables are on the stack (in automatic storage).
- All dynamically allocated memory is on the heap.

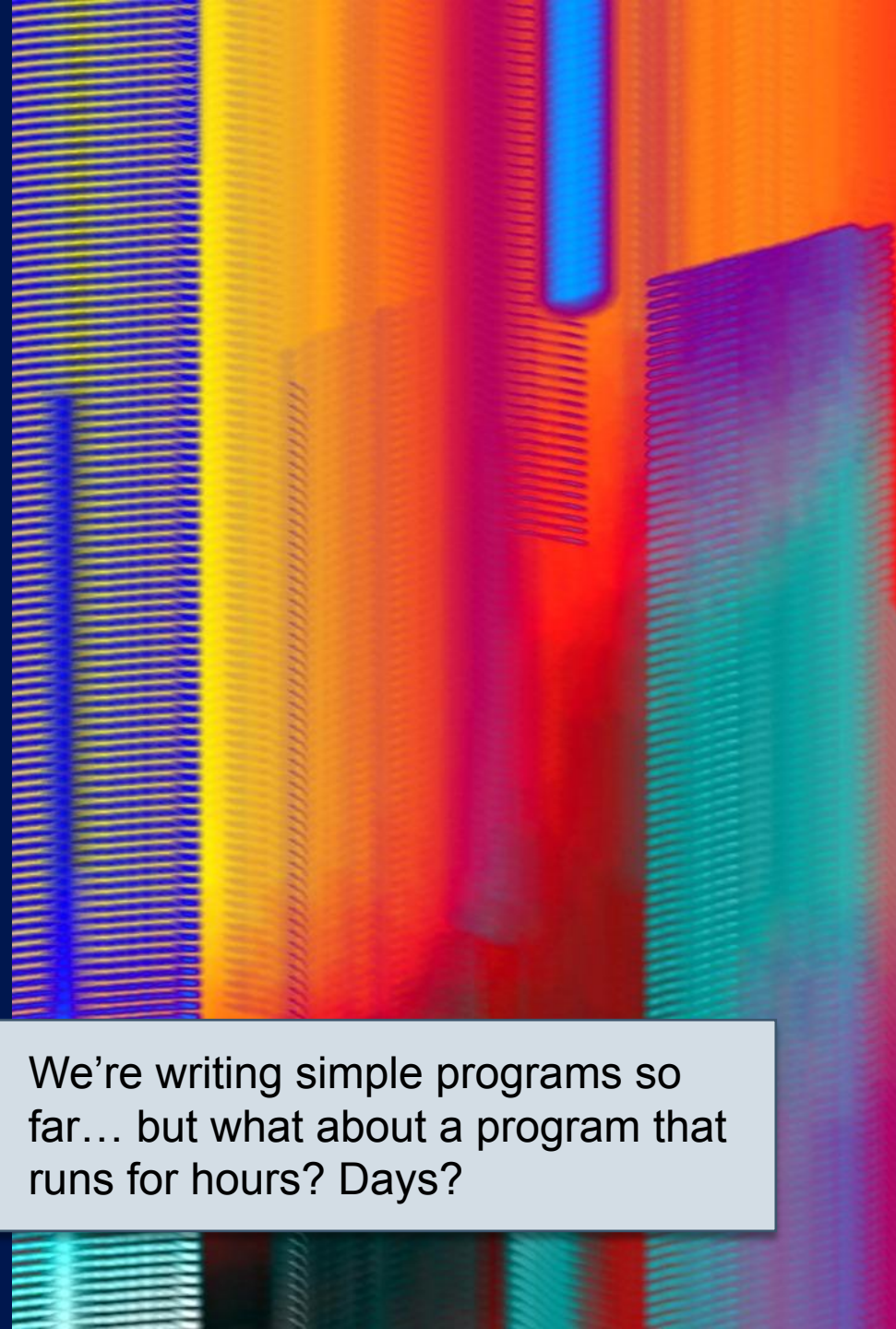
```
void Foo()
{
    int x;          // stack
    int* ptr = &x; // stack pointer, pointing to memory on the stack
    float* ptr2;   // stack
    double* someDoubles = new double[25]; // someDoubles is on the stack
}
```

Pointer, on the stack

Allocated memory, on the heap

When Does Heap Data “Disappear”?

- Stack-based variables get “deleted” when they go out of scope.
  - When their function comes off the call stack
  - The space they once occupied will later be overwritten.
- Stack-based memory is suitable for temporary data.
  - Create some variables, use them, then forget about them
- Heap data lives on… “forever”.
  - It never falls out of scope.
  - You must delete it.
  - Most operating systems will free up memory when a program ends.



We’re writing simple programs so far… but what about a program that runs for hours? Days?

Recap

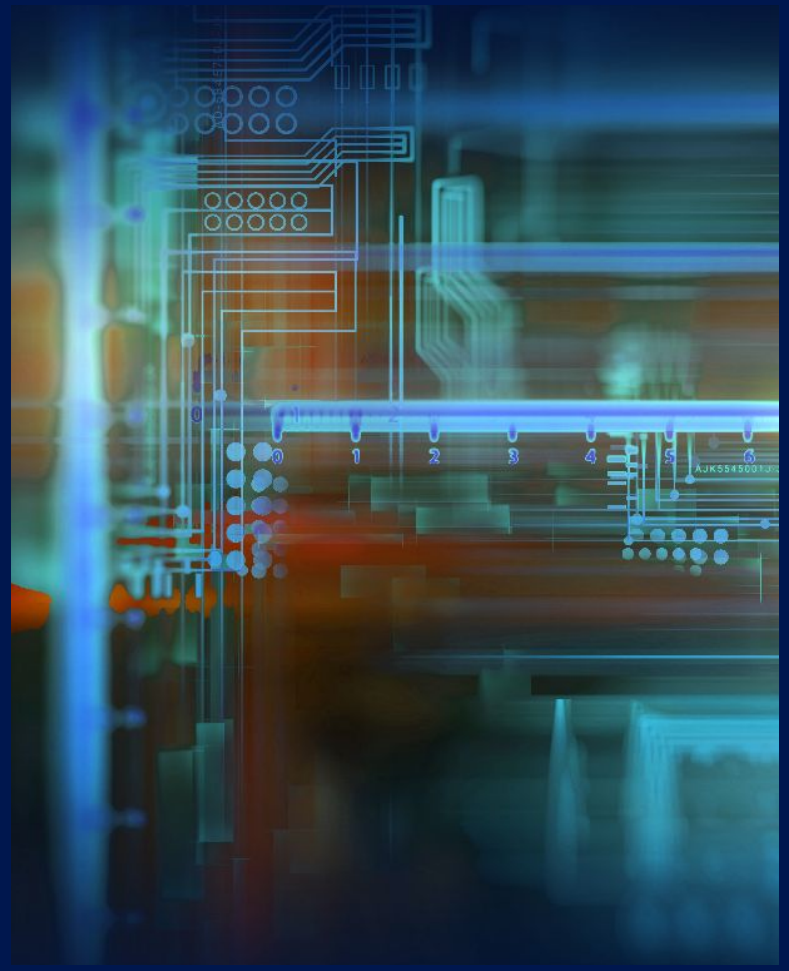
Stack

- Short-term storage
- **Automatic memory**:
  no new/delete required
- Small in size
- Functions and the local variables they use are placed on the stack
- Stack overflow errors if you try to use more than you have

Heap

- Long-term storage
  (or large data storage)
- **Dynamic memory**:
  The programmer must allocate and deallocate
- Large, essentially all the rest of your computer's memory
- Memory leaks can occur if not managed properly.

Conclusion



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

Thank you for watching.