# Lecture 2a: Primitives and Arrays on the Stack

CS 70: Data Structures and Program Development

Tuesday, January 28

# Focusing in on C++

# Declaring variables

```
int x = 3;
```

C++ variables have: a name, a type, a value, and a location in memory.

1. Who chooses these four?

# Declaring variables

```cpp
int x = 3;
```

C++ variables have: a name, a type, a value, and a location in memory.

1. Who chooses these four?
2. Which of these four can change while the program runs?

# Declaring variables

```
int x = 3;
```

C++ variables have: a name, a type, a value, and a location in memory.

1. Who chooses these four?
2. Which of these four can change while the program runs?
3. What does `x` look like in memory, while the program is running?

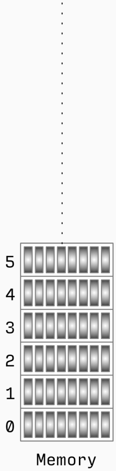
# Functions and Memory

Every running function in C++ needs a fixed, minimum amount of memory.

- Space for function arguments
- Space for local variables
- (Extra space added by the compiler) **Note: not in our model**
    - (Space for the "return address")
    - (Scratch space for temporary calculations)

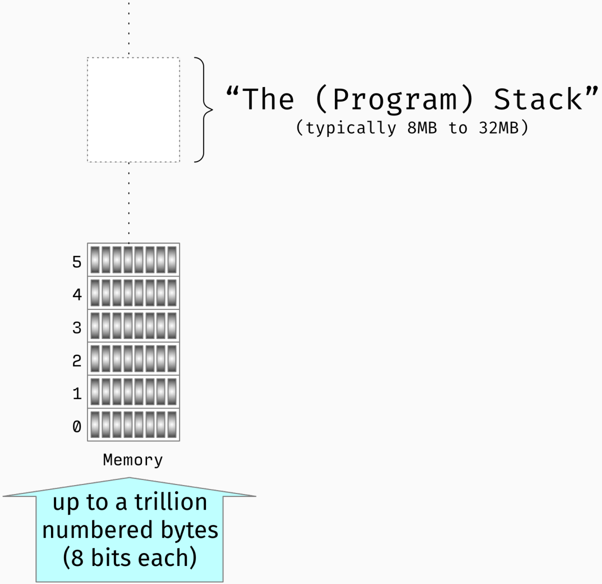
Where to put this data?

# Recall: Memory




up to a trillion numbered bytes (8 bits each)

# “The Stack”

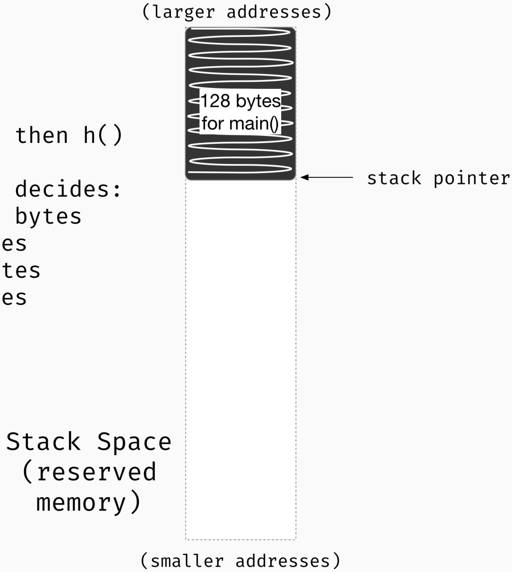

“The (Program) Stack”
(typically 8MB to 32MB)
5
4
3
2
1
0
Memory
up to a trillion
numbered bytes
(8 bits each)

# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```

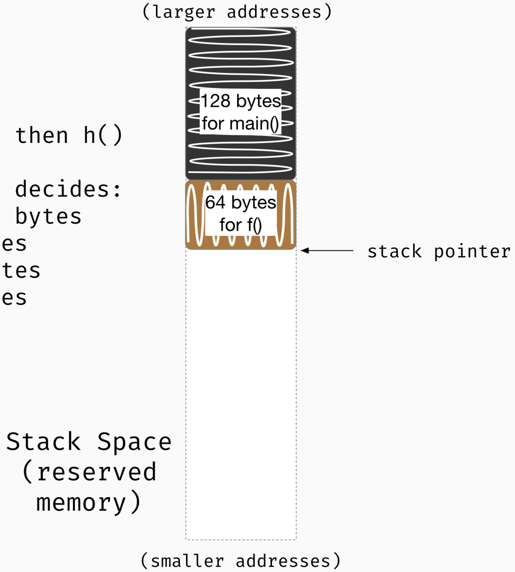
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



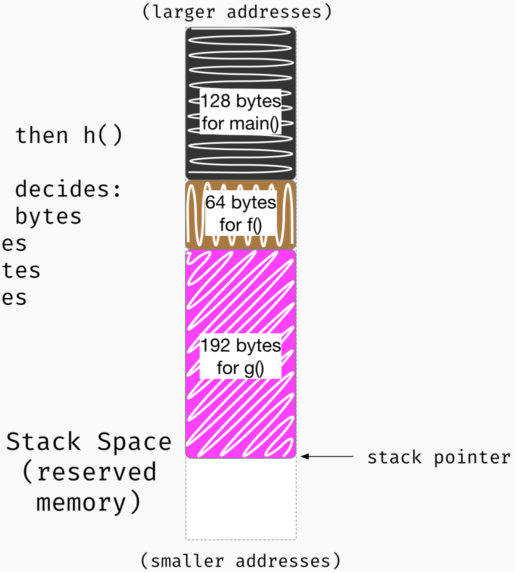
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



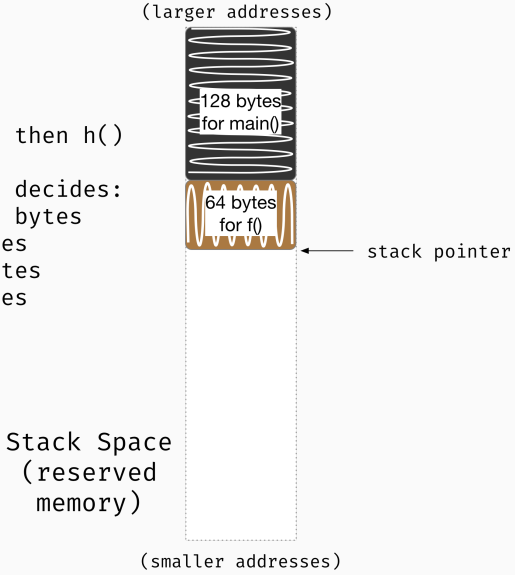
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



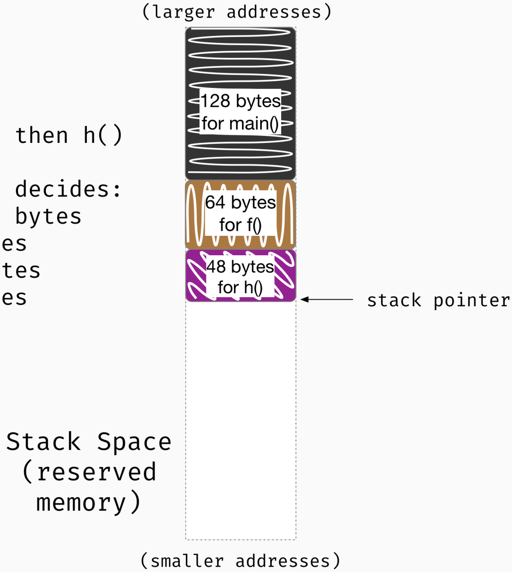
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



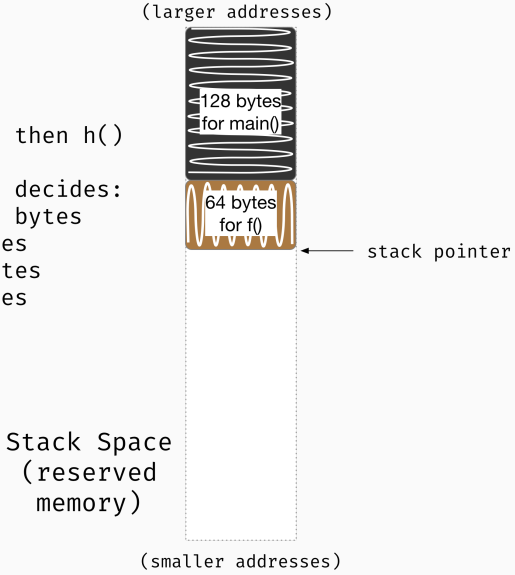
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



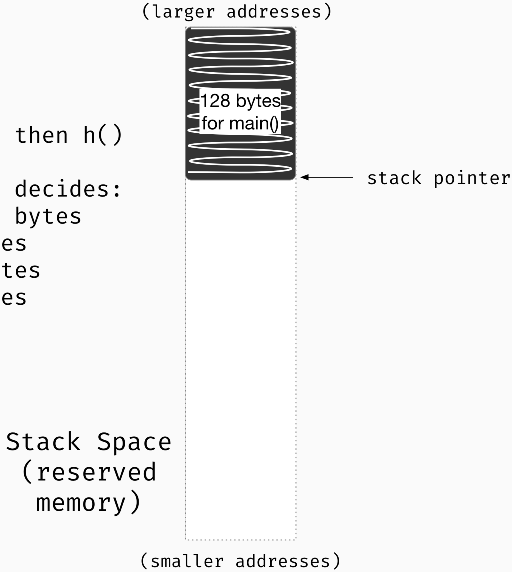
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



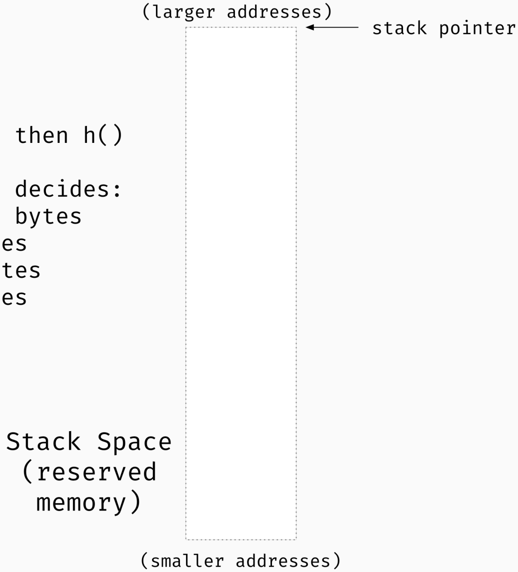
# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



# Function Calls

```
Suppose:
  main calls f()
  f() calls g(), then h()

and the compiler decides:
  main needs 128 bytes
  f needs 64 bytes
  g needs 192 bytes
  h needs 48 bytes
```



# Takeaway: Functions and Memory

Every function in C++ needs a fixed minimum amount of memory while it runs

- Space for function arguments
- Space for local variables
- (Extra space added by the compiler)

Every function

- allocates stack space when it starts (decrease stack pointer)
- releases stack space when it ends (increase stack pointer)

The *compiler* figures out how much stack space each function needs

# Life-Cycles of Data

# The Life-Cycle of C++ Data

Every *individual* piece of data, over the course of its life:

1. **Allocation**: acquire memory for the data
2. **Initialization**: create the data
3. **Use**: read and/or modify the data
4. **Destruction**: clean up the data
5. **Deallocation**: relinquish the data's memory

# A Very Helpful Analogy!

1. **Allocation**: Buy the land
2. **Initialization**: Build the building
3. **Use**: Enjoy the building
4. **Destruction**: Demolish the building
5. **Deallocation**: Sell the land

# For local variables

1. **Allocation**: at the opening { of the function
2. **Initialization**: Line of declaration (for parameters, the opening '{')
   - If you don't specify, default initialization
   - For primitives, default initialization does nothing! (So initial value is undefined).
3. **Use**: from initialization to destruction
4. **Destruction**: ending '}' of the declaring block
   - For primitive types, destruction doesn't do anything
   - But after destruction you can't use the variable
5. **Deallocation**: ending '}' of the function

# Stack? Life Cycles?

```
int triple(int multiplier)                          // 1
{                                                   // 2
   int product = 3 * multiplier;                    // 3
   return product;                                  // 4
}                                                   // 5

int main()                                          // 6
{                                                   // 7
    int myInt;                                      // 8
    cout << "Enter an even number: " << endl;       // 9
    cin >> myInt;                                   // 10
    if (myInt % 2 == 0) {                           // 11
        int result = triple(myInt);                 // 12
        cout << result << endl;                     // 13
    }                                               // 14
    else {                                          // 15
        cout << "Not even!" << endl;                // 16
    }                                               // 17
    return 0;                                       // 18
}                                                   // 19
```

# Stack? Life Cycles?

```
int absCube(int base)           // 1
{                               // 2
    int outcome = base * base;  // 3
    outcome = outcome * base;   // 4
    if (outcome < 0) {          // 5
        outcome = -outcome;     // 6
    }                           // 7
    return outcome;             // 8
}                               // 9

int main()                      // 10
{                               // 11
    int myInt = 0;              // 12
    int myConstant = -3;        // 13
    myInt = absCube(myConstant); // 14

    cout << myInt << endl;      // 15

    return 0;                   // 16
}                               // 17
```

# Arrays

# What are Arrays?

# What are Arrays?

- Collection of homogenous elements
- Contiguous block of memory
- Ordered
- Constant-time access to any element (given its index)
- Cannot be resized once created.

# Why are Arrays?

- Low-level, low-overhead data structure
- Basic building block for other data structures

# Declaring an Array

```
int values[42];
```

(What is `values[5]`?)

~~`int values[]`~~

# Declaring an Array: Variable Size

```
const int DAYS_IN_WEEK = 7;
int payments[DAYS_IN_WEEK];
```

~~`int x = 42;`~~

~~`int values[x];`~~

# Declaring an Array: List Initialization

```
int payments[DAYS_IN_WEEK] = {10, 5, 5, 5, 5, 5, 10};
```

```
int values[42] = {1, 2, 3};
```

(What is `values[5]`?)

# Array Idiom

It's okay to default initialize the elements of an array, if we then *immediately* initialize *all* the elements.

```
int payments[DAYS_IN_WEEK];

for (size_t day = 0; day < DAYS_IN_WEEK; ++day) {
    cin >> payments[day];
}
```

# What happens if we write:

```
int values[3] = {1, 2, 3};
cout << values[10000] << endl;
```