

CS106L Lecture 3:

Initialization & References 🦄

Autumn 2024

Fabio Ibanez, Jacob Roberts-Baca

Attendance



<https://tinyurl.com/initandrefF24>

Quick reminder

First assignment goes out on **Friday, October 4th** and is due **Friday, October 11th**.

Anonymous Feedback Form

<https://tinyurl.com/feedbackF24>

On pacing



A quick recap

1. **auto**: a keyword that tells the compiler to deduce the type of an object or variable

A quick recap

1. **auto**: a keyword that tells the compiler to deduce the type of an object or variable
 - a. Use at your discretion
 - b. Typically when the type is **annoyingly** verbose to write out

```
#include <iostream>
#include <string>
#include <map>
#include <unordered_map>
#include <vector>

int main()
{
    std::map<std::string, std::vector<std::pair<int, std::unordered_map<char, double>>>>
    complexType;

    /// what does this do? We'll find out in the iterators lecture!
    std::map<std::string, std::vector<std::pair<int, std::unordered_map<char, double>>>>::iterator
    it = complexType.begin();

    // vs
    auto it = complexType.begin();

    return 0;
}
```


A quick recap

1. **auto**: a keyword that tells the compiler to deduce the type of an object or variable
 - a. Use at your discretion
 - b. Typically when the type is *annoyingly* verbose to write out
2. **Structs** are a way to bundle many variables into one type

Plan

1. Initialization
2. References
3. L-values vs R-values
4. Const
5. Compiling C++ programs

Initialization

What?: “Provides initial values at the time of construction” - cppreference.com

Initialization

What?: “Provides initial values at the time of construction” - cppreference.com

How? 🤔:

1. Direct initialization
2. Uniform initialization
3. Structured Binding

Initialization

What?: “Provides initial values at the time of construction” - cppreference.com

How? 🤔:

1. Direct initialization

2. Uniform initialization

3. Structured Binding

Direct initialization

```
#include <iostream>
```

```
int main() {  
    int numOne = 12.0;  
    int numTwo(12.0);  
  
    std::cout << "numOne is: " << numOne << std::endl;  
    std::cout << "numTwo is: " << numTwo << std::endl;  
  
    return 0;  
}
```

Notice!!:

is 12.0 an int?

Direct initialization

```
#include <iostream>

int main() {
    int numOne = 12.0;
    int numTwo(12.0);

    std::cout << "numOne is: " << numOne << std::endl;
    std::cout << "numTwo is: " << numTwo << std::endl;

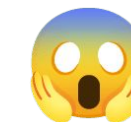
    return 0;
}
```

Notice!!:

is 12.0 an int?

NO

C++ Doesn't Care



```
numOne is: 12
numTwo is: 12
```

```
...Program finished with exit code 0
Press ENTER to exit console.
```


Problem? 🤔

```
#include <iostream>
```

```
int main() {  
    // Direct initialization with a floating-point value  
    int criticalSystemValue(42.5);  
  
    // Critical system operations...  
    // ...  
  
    std::cout << "Critical system value: " << criticalSystemValue << std::endl;  
  
    return 0;  
}
```

Problem? 🤔

```
Critical system value: 42
```

```
...Program finished with exit code 0  
Press ENTER to exit console. 
```

Recall

```
#include <iostream>
```

```
int main() {  
    int numOne = 12.0;  
    int numTwo(12.0);  
  
    std::cout << "numOne is: " << numOne << std::endl;  
    std::cout << "numTwo is: " << numTwo << std::endl;  
  
    return 0;  
}
```

Notice!!:

is 12.0 an int?

NO

C++ Doesn't Care



```
numOne is: 12  
numTwo is: 12
```

```
...Program finished with exit code 0  
Press ENTER to exit console.
```

What happened? 🤔

```
#include <iostream>
```

```
int main() {  
    // Direct initialization with a floating-point value  
    int criticalSystemValue(42.5);  
  
    // Critical system operations...  
    // ...  
  
    std::cout << "Critical system value: " << criticalSystemValue << std::endl;  
  
    return 0;  
}
```

The user intended to save a float, 42.5, into **criticalSystemValue**

What happened? 🤔

```
#include <iostream>
```

```
int main() {  
    // Direct initialization with a floating-point value  
    int criticalSystemValue(42.5);  
  
    // Critical system operations...  
    // ...  
  
    std::cout << "Critical system value: " << criticalSystemValue << std::endl;  
  
    return 0;  
}
```

C++ doesn't care in this case, it doesn't type check with direct initialization

What happened? 🤔

```
#include <iostream>

int main() {
    // Direct initialization with a floating-point value
    int criticalSystemValue(42.5);

    // Critical system operations...
    // ...

    std::cout << "Critical system value: " << criticalSystemValue << std::endl;

    return 0;
}
```

So C++ said “Meh, I’ll store 42.5 as an int,” and we possibly now have an error. This is commonly called a **narrowing conversion**

Initialization

What?: “Provides initial values at the time of construction” - cppreference.com

How? 🤔:

1. Direct initialization
- 2. Uniform initialization**
3. Structured Binding

Uniform initialization (C++11)

```
#include <iostream>

int main() {
    // Notice the brackets
    int numOne{12.0};
    float numTwo{12.0};

    std::cout << "numOne is: " << numOne << std::endl;
    std::cout << "numTwo is: " << numTwo << std::endl;

    return 0;
}
```

Notice!!:

the curly braces!

With uniform
initialization C++
does care about
types!

Uniform initialization (C++11)

```
#include <iostream>
```

```
int main() {  
    // Notice the brackets  
    int numOne{12.0};  
    float numTwo{12.0};
```

```
    std::cout << "numOne is: " << numOne << std::endl;  
    std::cout << "numTwo is: " << numTwo << std::endl;
```

```
narrowing_conversion.cpp:5:16: error: type 'double' cannot be narrowed to 'int' in  
initializer list [-Wc++11-narrowing]
```

```
    int numOne{12.0};
```

^~~~

```
narrowing_conversion.cpp:5:16: note: insert an explicit cast to silence this issue
```

```
    int numOne{12.0};
```

^~~~

```
        static_cast<int>( )
```

```
1 error generated.
```

Notice!!:

the curly braces!

With uniform

initialization C++

Uniform initialization (C++11)

```
#include <iostream>
```

```
int main() {  
    // Notice the brackets  
    int numOne{12.0};  
    float numTwo{12.0};
```

```
    std::cout << "numOne is. :endl;  
    std::cout << "numTwo is. :endl;
```

```
narrowing_conversion.cpp:5:16: error: type 'double' cannot be narrowed to 'int' in  
initializer list [-Wc++11-narrowing]
```

```
    int numOne{12.0};
```

```
narrowing_conversion.cpp:5:16: note: insert an explicit cast to silence this issue
```

```
    int numOne{12.0};
```

```
        static_cast<int>( )
```

```
1 error generated.
```



Notice!!:

the curly braces!

With uniform

initialization C++

Uniform initialization (C++11)

```
#include <iostream>

int main() {
    // Notice the brackets
    int numOne{12};
    float numTwo{12.0};

    std::cout << "numOne is: " << numOne << std::endl;
    std::cout << "numTwo is: " << numTwo << std::endl;

    return 0;
}
```

Notice!!:

12 instead of 12.0



Uniform initialization (C++11)

```
#include <iostream>

int main() {
    // Notice the brackets
    int numOne{12};
    float numTwo{12.0};

    std::cout << "numOne is: " << numOne << std::endl;
    std::cout << "numTwo is: " << numTwo << std::endl;

    return 0;
}
```

```
numOne is: 12
numTwo is: 12
```

Notice!!:

12 instead of 12.0



Uniform initialization (C++11)

Uniform initialization is awesome because:

1. It's **safe**! It doesn't allow for narrowing conversions—which can lead to unexpected behaviour (or critical system failures :o)

Uniform initialization (C++11)

Uniform initialization is awesome because:

1. It's **safe**! It doesn't allow for narrowing conversions—which can lead to unexpected behaviour (or critical system failures :o)
1. It's **ubiquitous** it works for all types like vectors, maps, and custom classes, among other things!

Uniform initialization (Map)

```
#include <iostream>
#include <map>

int main() {
    // Uniform initialization of a map
    std::map<std::string, int> ages{
        {"Alice", 25},
        {"Bob", 30},
        {"Charlie", 35}
    };

    // Accessing map elements
    std::cout << "Alice's age: " << ages["Alice"] << std::endl;
    std::cout << "Bob's age: " << ages.at("Bob") << std::endl;

    return 0;
}
```

Uniform initialization (Map)

```
#include <iostream>
#include <map>

int main() {
    // Uniform initialization of a map
    std::map<std::string, int> ages{
        {"Alice", 25},
        {"Bob", 30},
        {"Charlie", 35}
    };

    // Accessing map elements
    std::cout << "Alice's age: " << ages["Alice"] << std::endl;
    std::cout << "Bob's age: " << ages.at("Bob") << std::endl;

    return 0;
}
```

```
Alice's age: 25
Bob's age: 30
```

Uniform initialization (Vector)

```
#include <iostream>
#include <vector>

int main() {
    // Uniform initialization of a vector
    std::vector<int> numbers{1, 2, 3, 4, 5};

    // Accessing vector elements
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Uniform initialization (Vector)

```
#include <iostream>
#include <vector>

int main() {
    // Uniform initialization of a vector
    std::vector<int> numbers{1, 2, 3, 4, 5};

    // Accessing vector elements
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

1 2 3 4 5


Recall

List Initialization

```
StanfordID id;  
id.name = "Jacob Roberts-Baca";  
id.sunet = "jtrb";  
id.idNumber = 6504417;
```

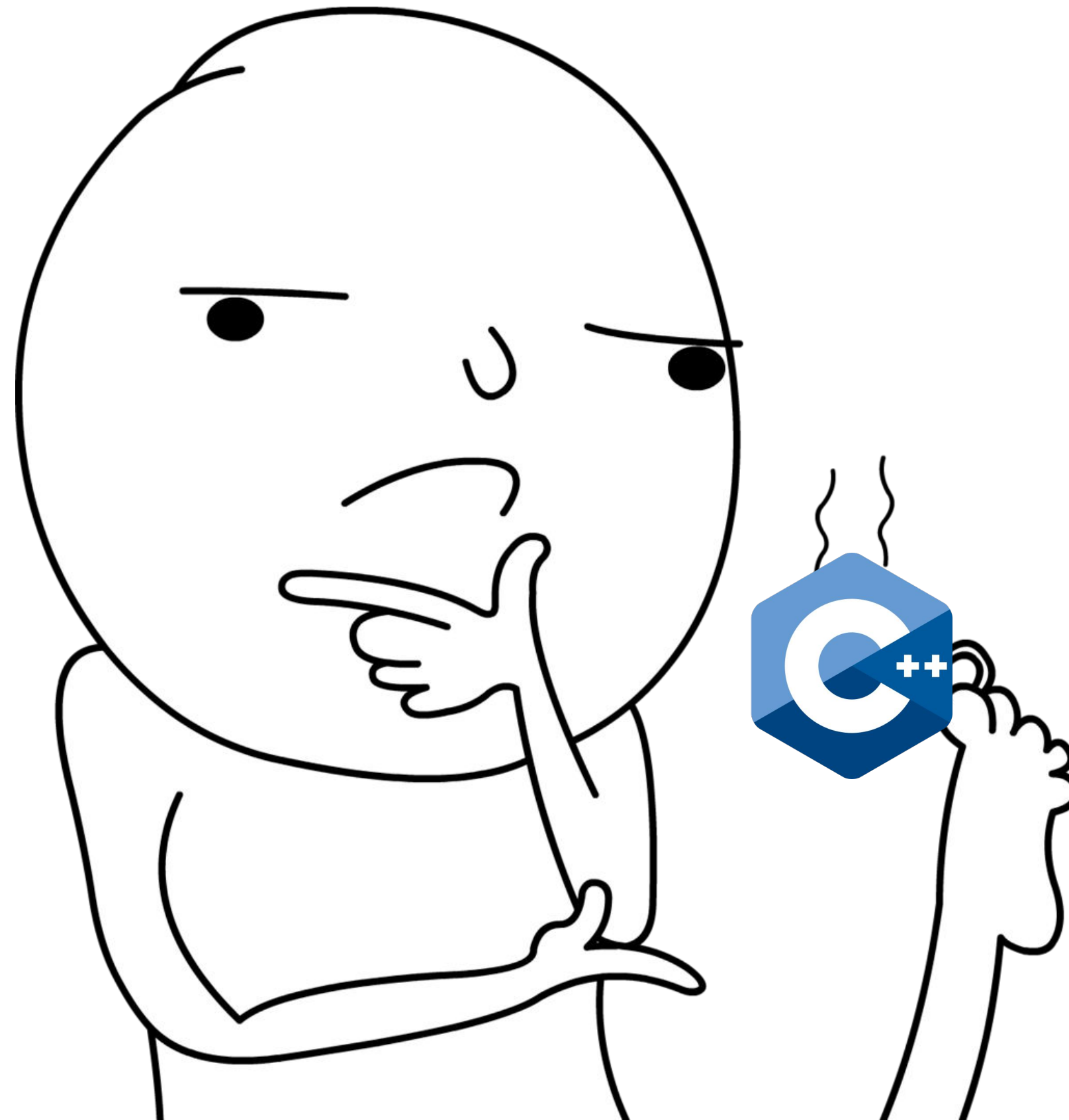


We'll learn more
about this next time!



```
// Order depends on field order in struct. '=' is optional  
StanfordID jrb = { "Jacob Roberts-Baca", "jtrb", 6504417 };  
StanfordID fi { "Fabio Ibanez", "fibanez", 6504418 };
```

What questions do we have?



Initialization

What?: “Provides initial values at the time of construction” - cppreference.com

How? 🤔:

1. Direct initialization
2. Uniform initialization
- 3. Structured Binding**

Structured Binding (C++ 17)

- A useful way to initialize some variables from data structures with fixed sizes at compile time

Structured Binding (C++ 17)

- A useful way to initialize some variables from data structures with fixed sizes at compile time
- Ability to access multiple values returned by a function

Structured Binding (C++ 17)

```
std::tuple<std::string, std::string, std::string> getClassInfo() {  
    std::string className = "CS106L";  
    std::string buildingName = "Thornton 110";  
    std::string language = "C++";  
    return {className, buildingName, language};  
}  
  
int main() {  
    auto [className, buildingName, language] = getClassInfo();  
    std::cout << "Come to " << buildingName << " and join us for " << className  
              << " to learn " << language << "!" << std::endl;  
  
    return 0;  
}
```

Structured Binding (C++ 17)

```
std::tuple<std::string, std::string, std::string> getClassInfo() {  
    std::string className = "CS106L";  
    std::string buildingName = "Thornton 110";  
    std::string language = "C++";  
    return {className, buildingName, language};  
}
```

Notice - uniform initialization!

```
int main() {  
    auto [className, buildingName, language] = getClassInfo();  
    std::cout << "Come to " << buildingName << " and join us for " << className  
              << " to learn " << language << "!" << std::endl;  
  
    return 0;  
}
```

Structured Binding (C++ 17)

```
std::tuple<std::string, std::string, std::string> getClassInfo() {  
    std::string className = "CS106L";  
    std::string buildingName = "Thornton 110";  
    std::string language = "C++";  
    return {className, buildingName, language};  
}  
  
int main() {  
    auto [className, buildingName, language] = getClassInfo();  
    std::cout << "Come to " << buildingName << " and join us for " << className  
              << " to learn " << language << "!" << std::endl;  
  
    return 0;  
}
```

Structured Binding (C++ 17)

```
#include <iostream>
#include <tuple>
#include <string>

std::tuple<std::string, std::string, std::string> getClassInfo() {
    std::string className = "CS106L";
    std::string buildingName = "Turing Auditorium";
    std::string language = "C++";
    return {className, buildingName, language};
}

int main() {
    auto classInfo = getClassInfo();
    std::string className = std::get<0>(classInfo);
    std::string buildingName = std::get<1>(classInfo);
    std::string language = std::get<2>(classInfo);

    std::cout << "Come to " << buildingName << " and join us for " << className
               << " to learn " << language << "!" << std::endl;
    return 0;
}
```

Structured Binding (C++ 17)

```
#include <iostream>
#include <tuple>
#include <string>

std::tuple<std::string, std::string, std::string> getClassInfo() {
    std::string className = "CS106L";
    std::string buildingName = "Turing Auditorium";
    std::string language = "C++";
    return {className, buildingName, language};
}

int main() {
    auto classInfo = getClassInfo();
    std::string className = std::get<0>(classInfo);
    std::string buildingName = std::get<1>(classInfo);
    std::string language = std::get<2>(classInfo);

    std::cout << "Come to " << buildingName << " and join us for " << className
               << " to learn " << language << "!" << std::endl;
    return 0;
}
```

Structured Binding (C++ 17)

```
std::tuple<std::string, std::string, std::string> getClassInfo() {  
    std::string className = "CS106L";  
    std::string buildingName = "Thornton 110";  
    std::string language = "C++";  
    return {className, buildingName, language};  
}
```

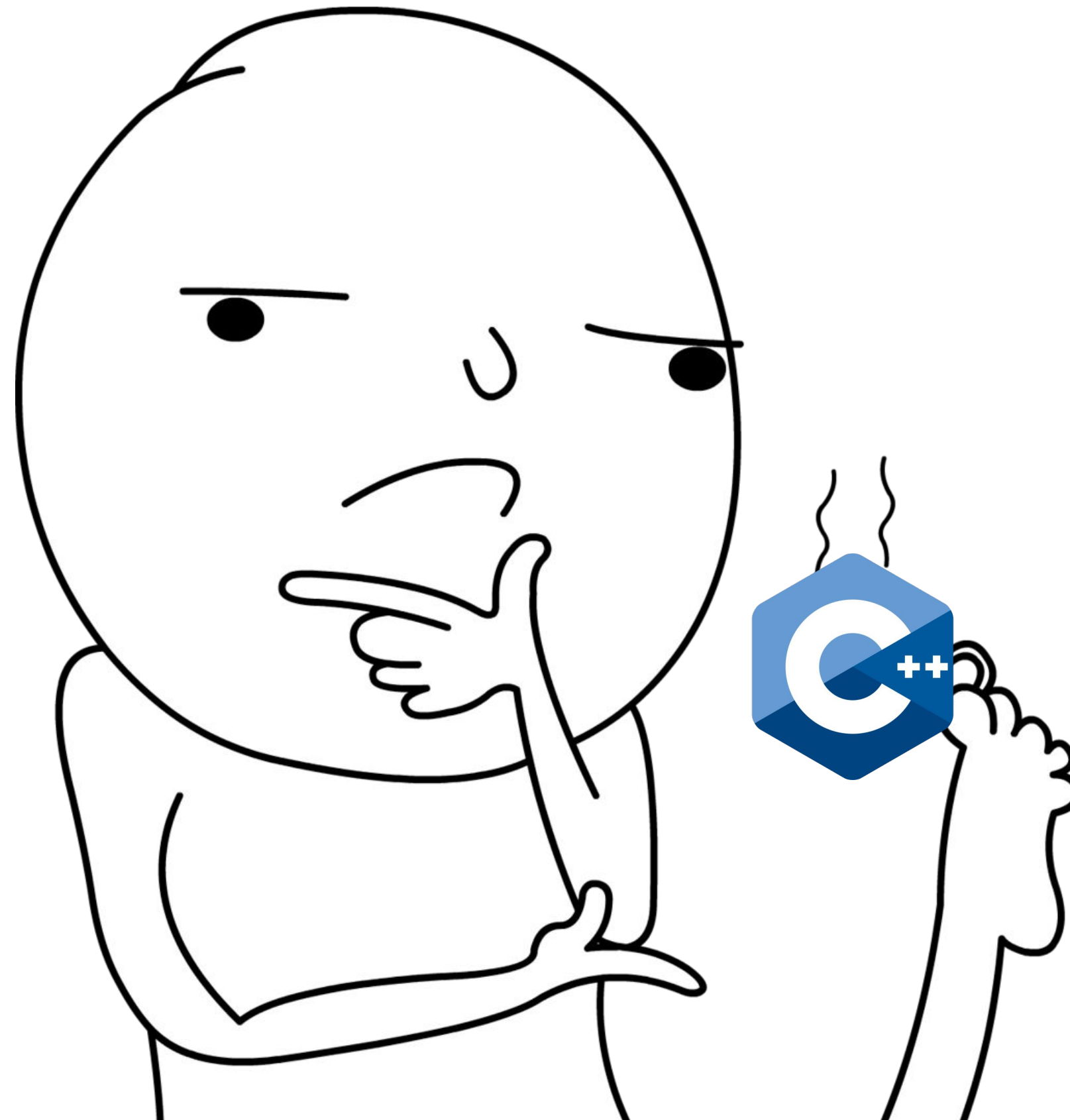
```
int main() {  
    auto [className, buildingName, language] = getClassInfo();  
    std::cout << "Come to " << buildingName << " and join us for "  
               << " to learn " << language << "!" << std::endl;  
  
    return 0;  
}
```



Structured Binding (C++ 17)

- A useful way to initialize some variables from data structures with fixed sizes at compile time
- Ability to access multiple values returned by a function
- Can use on objects where the size is **known at compile-time**

What questions do we have?



Plan

1. Initialization
- 2. References**
3. L-values vs R-values
4. Const
5. Compiling C++ programs

References

What?: “Declares a name variable as a reference”

tldr: a reference is an alias to an already-existing

thing - cppreference.com

References

What?: “Declares a name variable as a reference”

tldr: a reference is an alias to an already-existing

thing - cppreference.com

How? 🤔:

Use an ampersand (&)

The & and the how

```
int num = 5;  
int& ref = num;  
  
ref = 10;    // Assigning a new value through the  
reference  
std::cout << num << std::endl;    // Output: 10
```

num is a variable of type `int`, that is assigned to have the value 5

The & and the how

```
int num = 5;  
int& ref = num;  
  
ref = 10;    // Assigning a new value through the  
reference  
std::cout << num << std::endl;    // Output: 10
```

ref is a variable of type `int&`, that is an alias to `num`

The & and the how

```
int num = 5;  
int& ref = num;  
  
ref = 10; // Assigning a new value through the reference  
std::cout << num << std::endl; // Output: 10
```

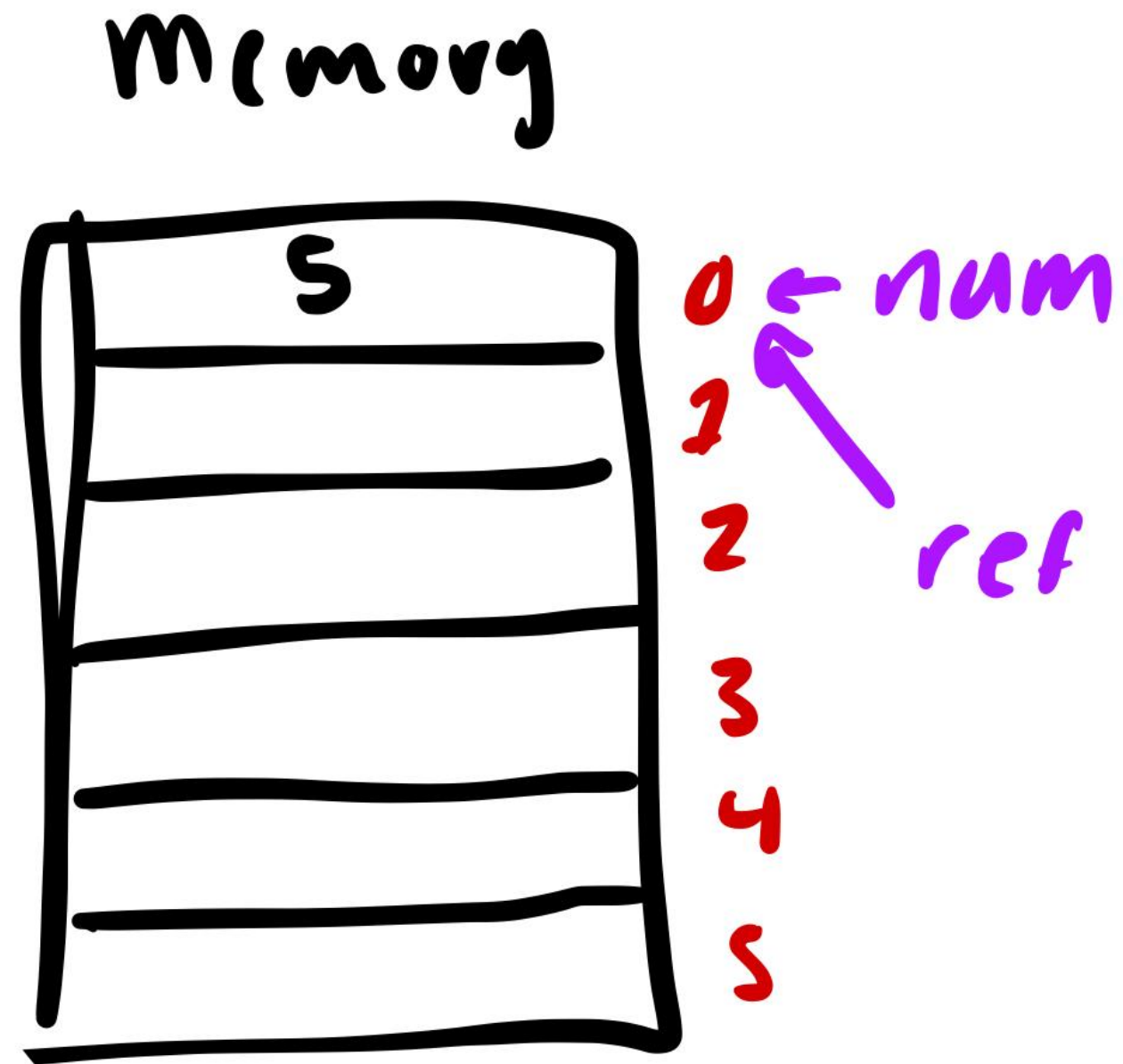
So when we assign 10 to ref, we also change the value of num, since ref is an alias for num

Visually



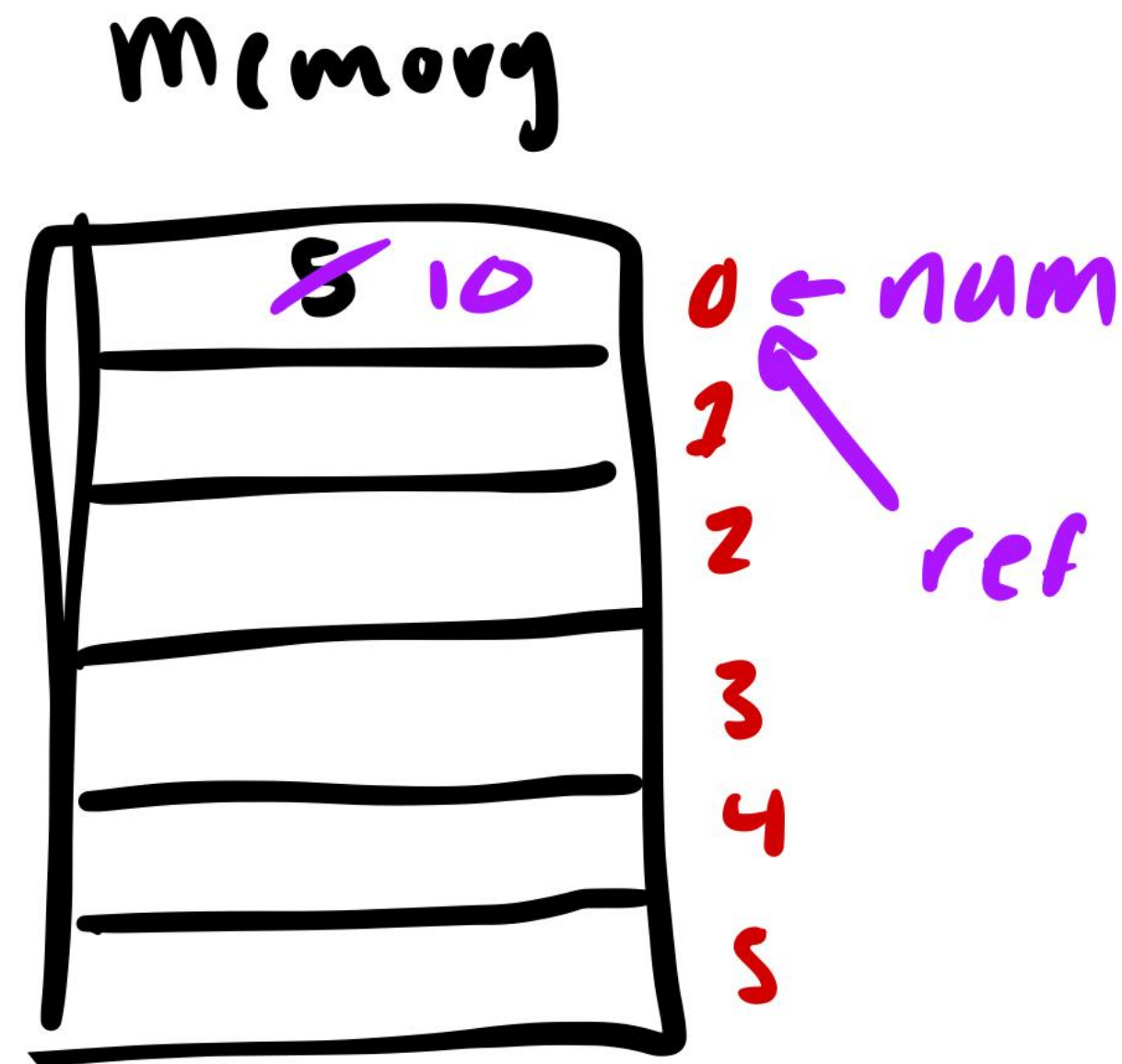
num is a variable of type `int`, that is assigned to have the value 5

Visually



`ref` is a variable of type `int&`, that is an alias to `num`

Visually



So when we assign 10 to *ref*, we also change the value of *num*, since *ref* is an alias for *num*

Pass by reference

In 106B we learn about “pass by reference”. We can apply the same ideas from referenced variables to functions! Take a look:

Pass by reference

In 106B we learn about “pass by reference”. We can apply the same ideas from referenced variables to functions! Take a look:



```
#include <iostream>
#include <math.h>

// note the ampersand!
void squareN(int& n) {
    // calculates n to the power of 2
    n = std::pow(n, 2);
}

int main() {
    int num = 2;
    squareN(num);
    std::cout << num << std::endl;

    return 0;
}
```

Pass by reference

In 106B we learn about “pass by reference”. We can apply the same ideas from referenced variables to functions! Take a look:



```
#include <iostream>
#include <math.h>

// note the ampersand!
void squareN(int& n) {
    // calculates n to the power of 2
    n = std::pow(n, 2);
}

int main() {
    int num = 2;
    squareN(num);
    std::cout << num << std::endl;

    return 0;
}
```

Pass by reference

In 106B we learn about “pass by reference”. We can apply the same ideas from referenced variables to functions! Take a look:

Notice!!: `n` is being passed into `squareN` by reference, denoted by the ampersand!

```
#include <iostream>
#include <math.h>

// note the ampersand!
void squareN(int& n) {
    // calculates n to the power of 2
    n = std::pow(n, 2);
}

int main() {
    int num = 2;
    squareN(num);
    std::cout << num << std::endl;

    return 0;
}
```


Pass by reference

In 106B we learn about “pass by reference”. We can apply the same ideas from referenced variables to functions! Take a look:

So what ? : This means that **n** is actually going to be modified inside of **squareN**.

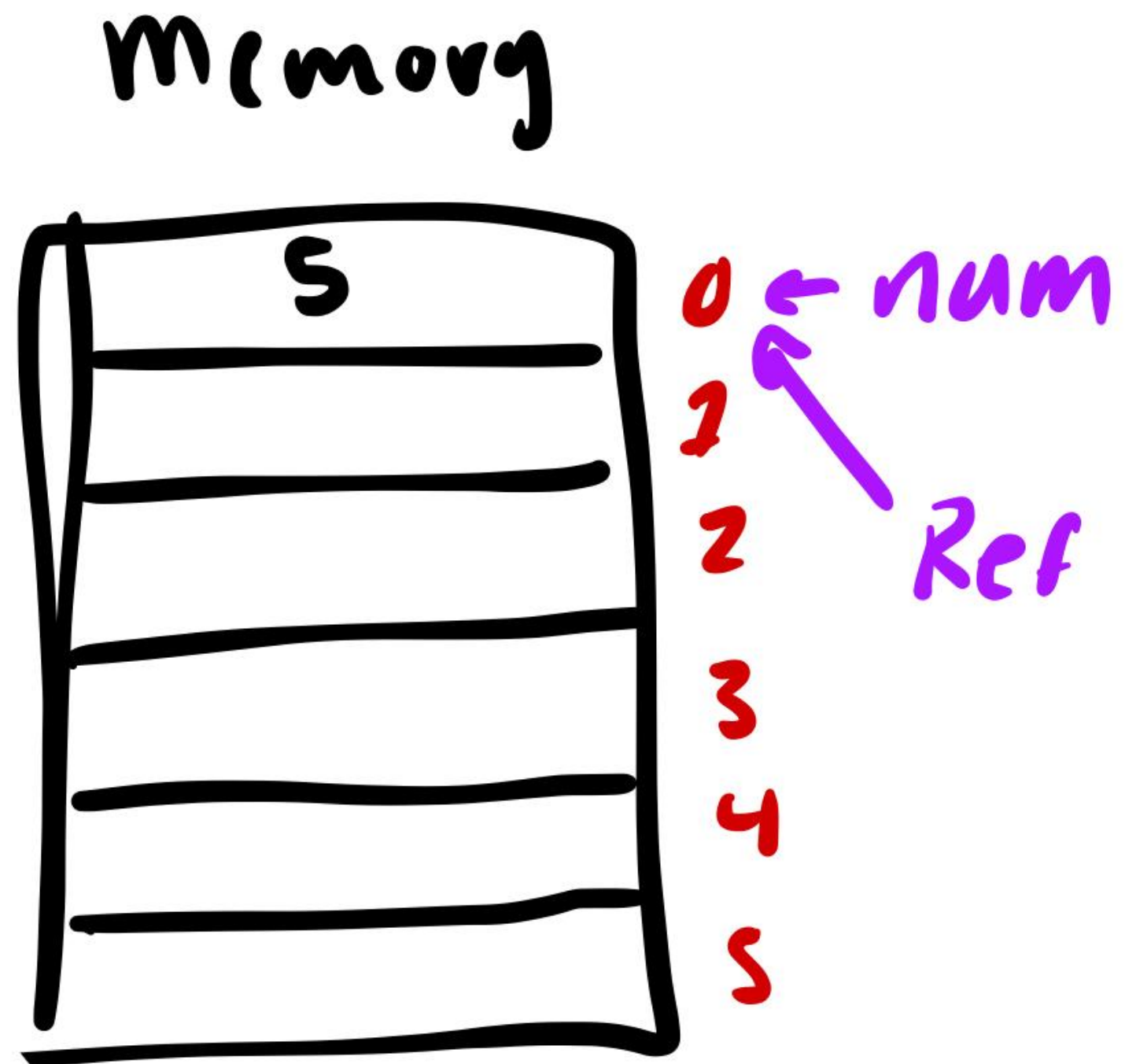
```
#include <iostream>
#include <math.h>

// note the ampersand!
void squareN(int& n) {
    // calculates n to the power of 2
    n = std::pow(n, 2);
}

int main() {
    int num = 2;
    squareN(num);
    std::cout << num << std::endl;

    return 0;
}
```

Recall



A **reference** *refers* to the same memory as its associated variable!

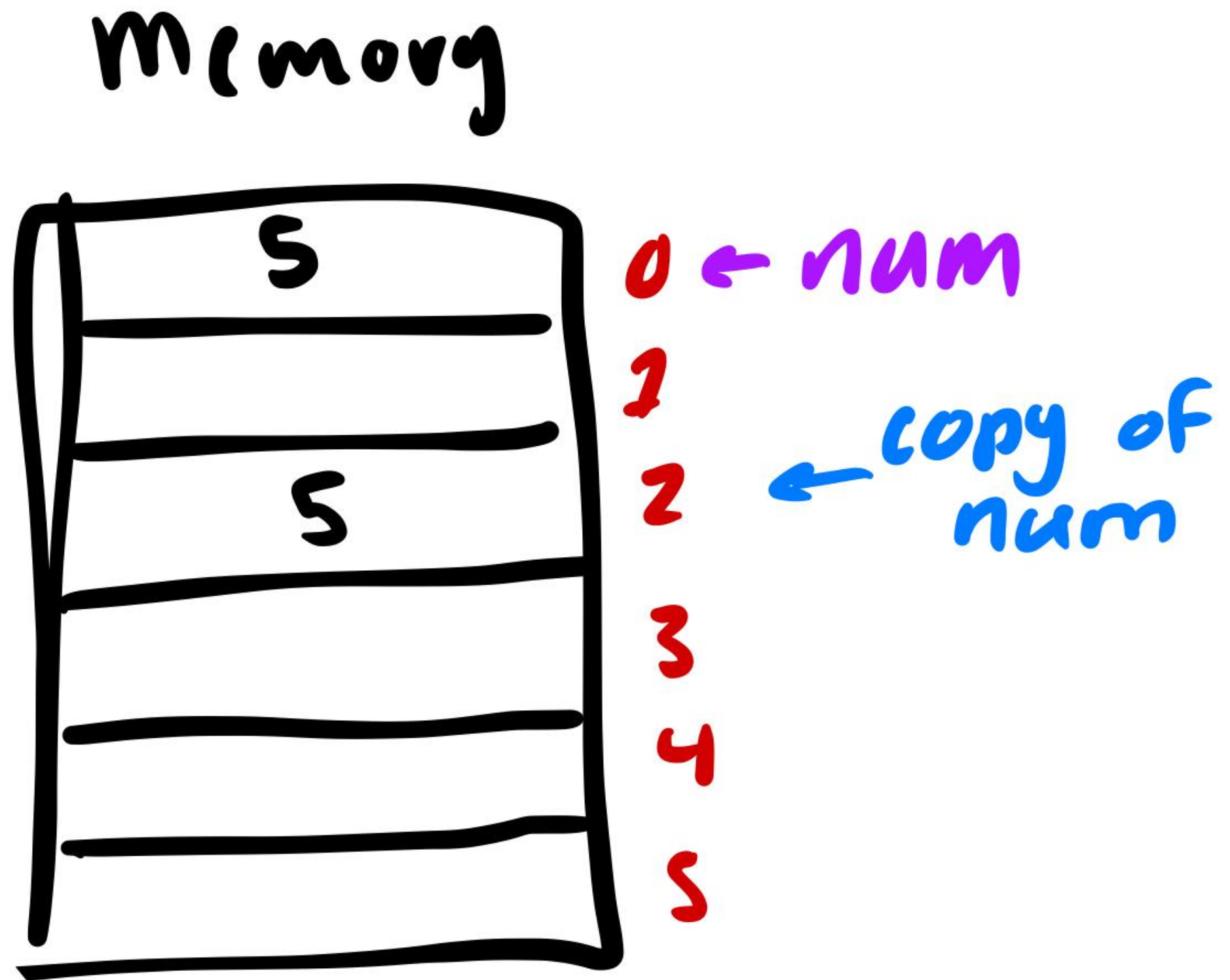
Recall

Passing in a variable by *reference* into a function just means “**Hey take in the actual piece of memory, don’t make a copy!**”

Passing by value

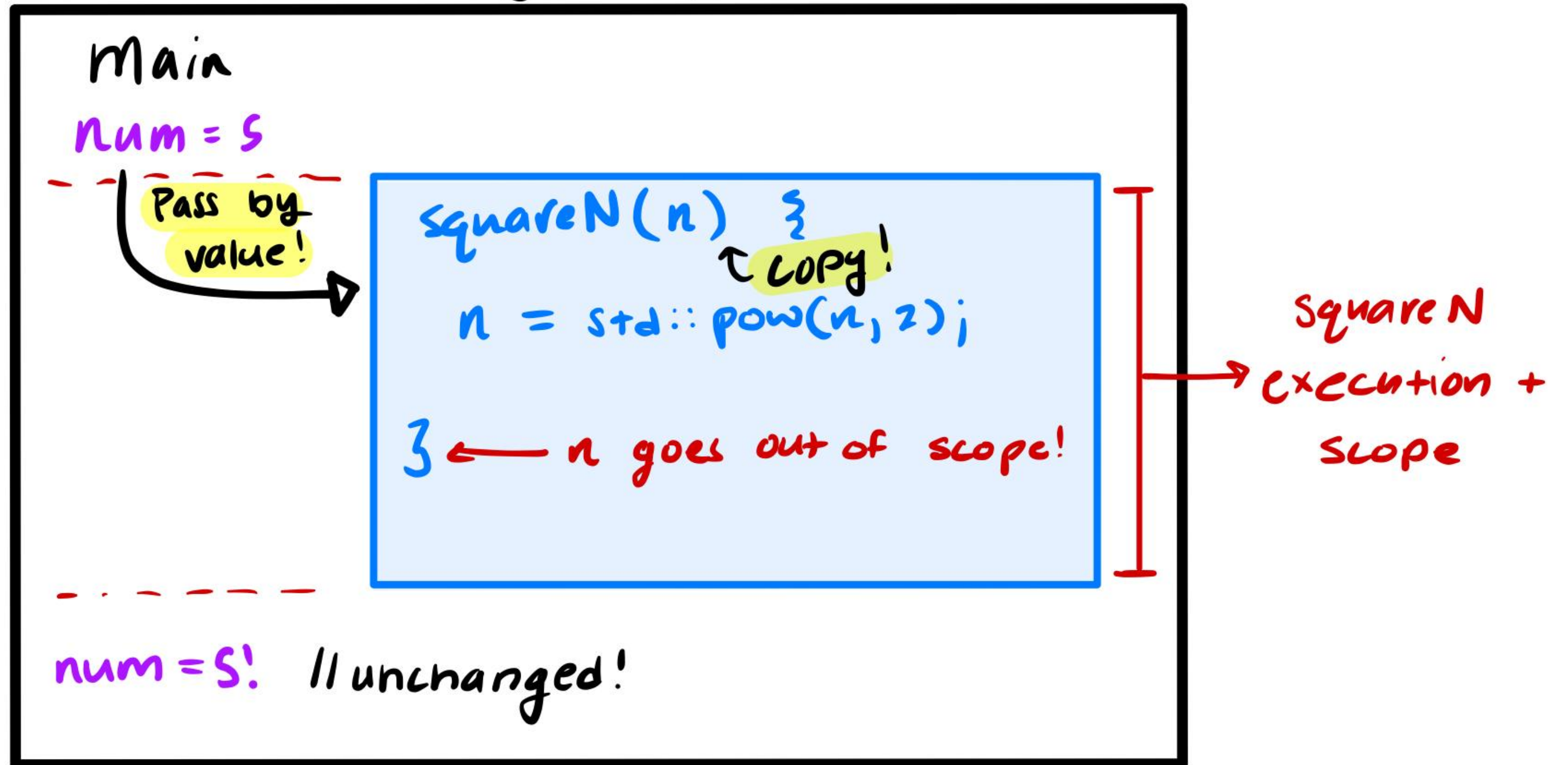
Passing in a variable by *value* into a function just means “**Hey make a copy, don’t take in the actual variable!**”

What does that look like?

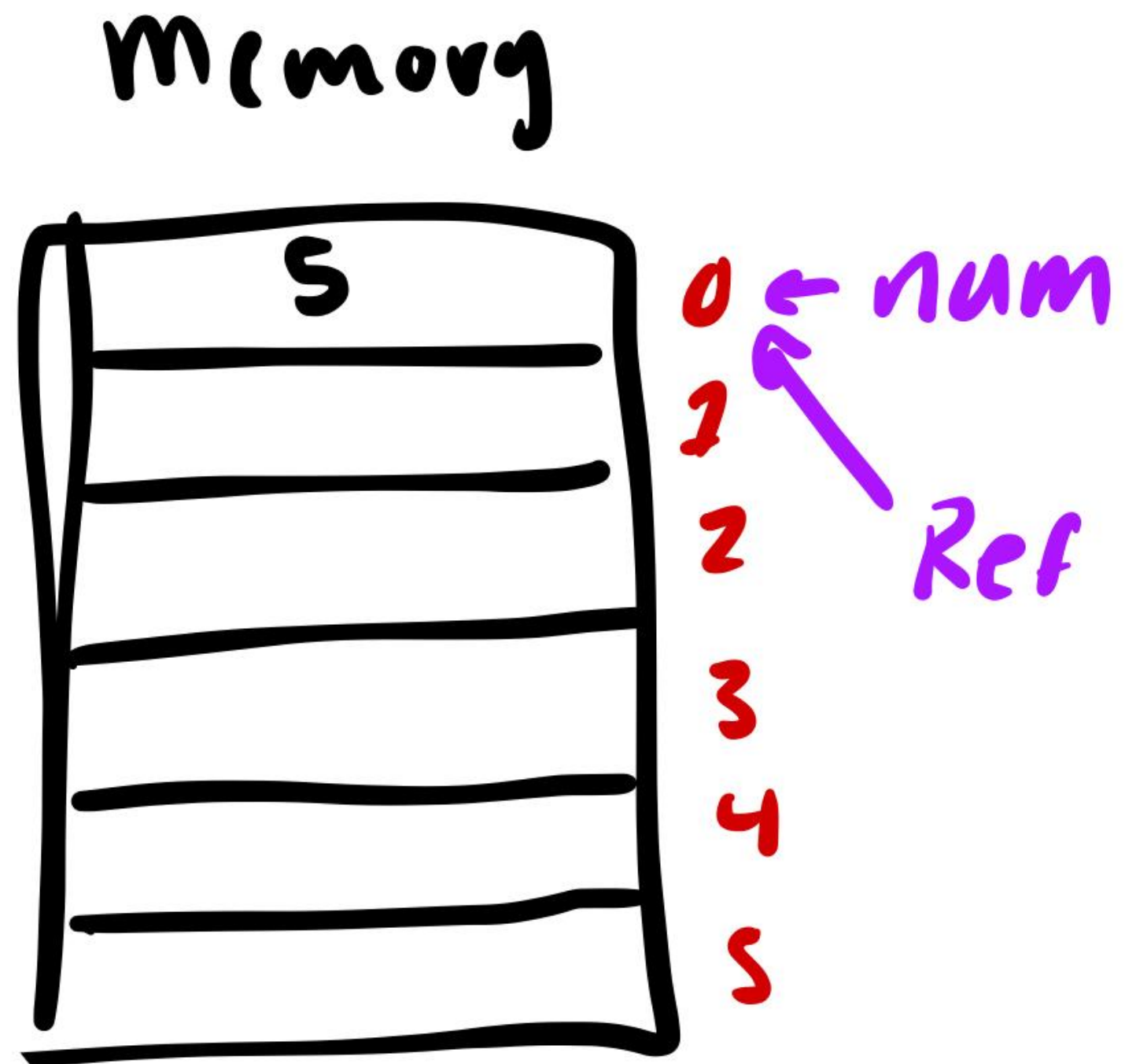


Passing by value (makes a copy)

Pass by value !



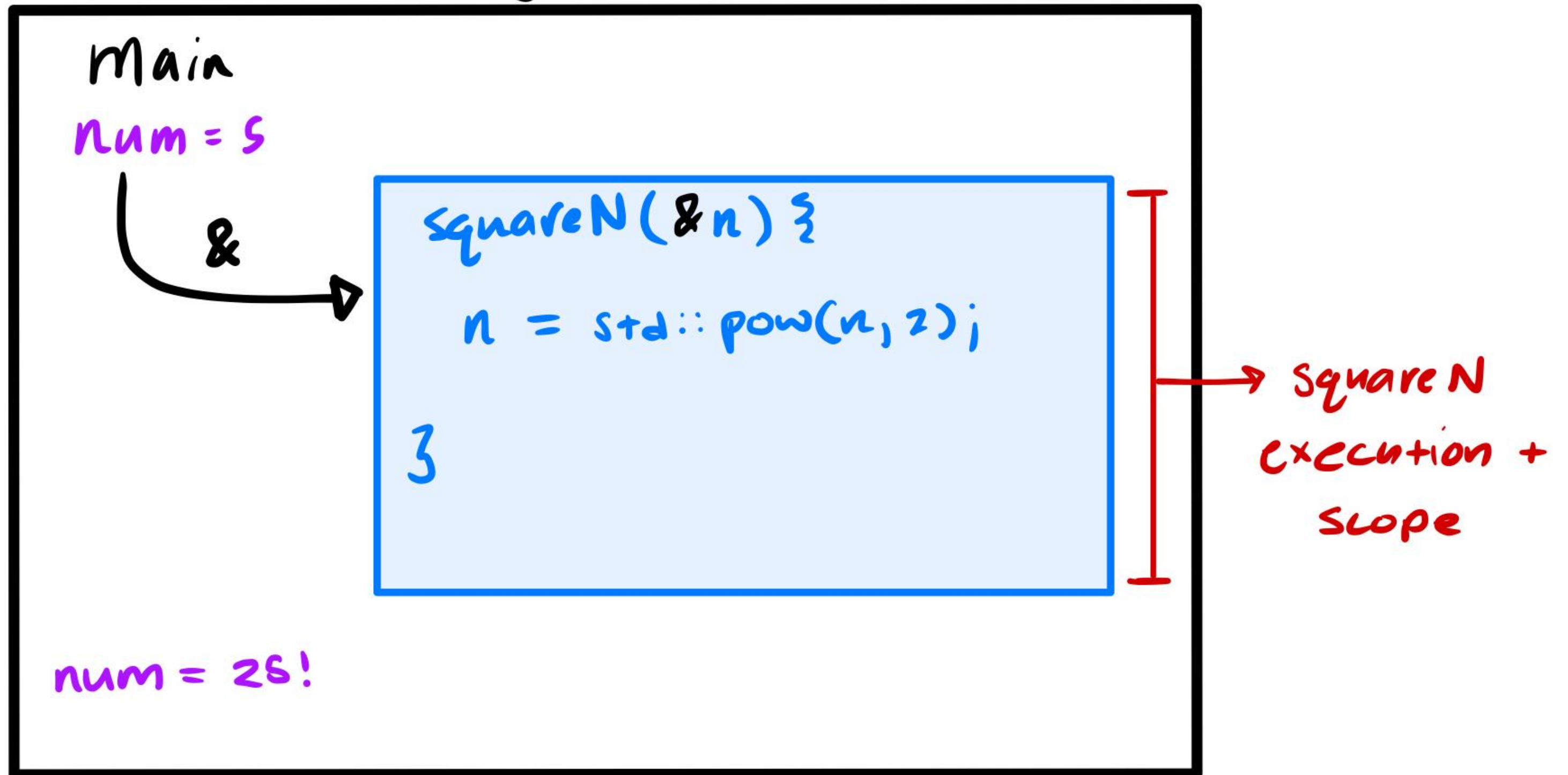
Recall



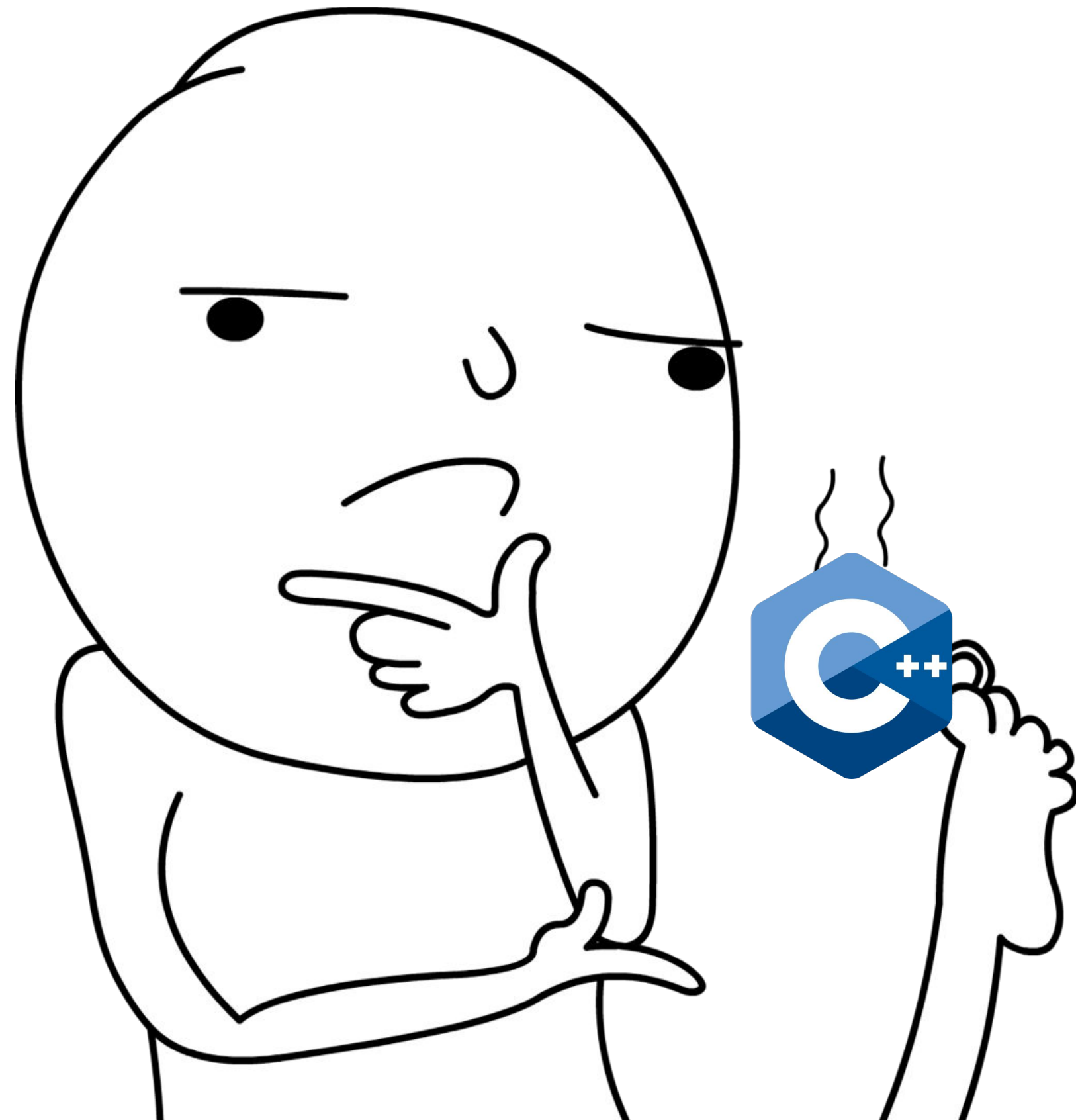
A **reference** *refers* to the same memory as its associated variable!

Passing by reference

Pass by reference!



What questions do we have?



OK! Let's take a look at an edge case!

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```

A classic reference-copy bug

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```



But nums is
passed in by
reference...

A classic reference-copy bug

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```

**Note the structured
binding!**

🤔 But nums is
passed in by
reference...

A classic reference-copy bug

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```

We're **not**
modifying nums
in this function!

A classic reference-copy bug

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```

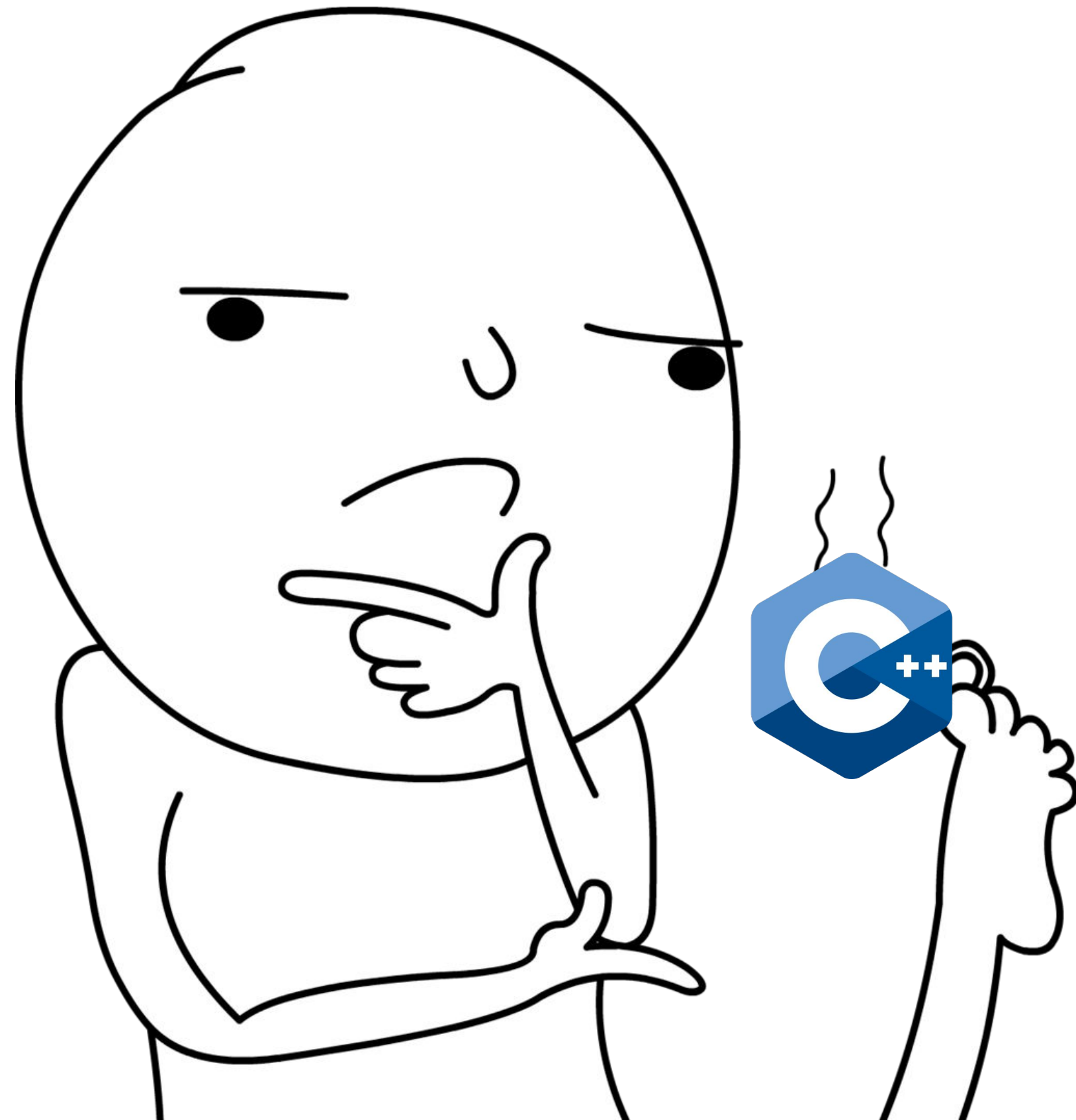
We are
modifying the
std::pair's
inside of nums

A classic reference-copy bug: fixed!

```
#include <iostream>
#include <math.h>
#include <vector>

void shift(std::vector<std::pair<int, int>> &nums) {
    for (auto& [num1, num2] : nums) {
        num1++;
        num2++;
    }
}
```


What questions do we have?



Plan

1. Initialization
2. References
- 3. L-values vs R-values**
4. Const
5. Compiling C++ programs

l-values and r-values

An l-value

An **l-value** can be to the left or the right of an equal sign!

l-values and r-values

An l-value

An **l-value** can be to the left or the right of an equal sign!

What's an example?

x can be an l-value for instance because you can have something like:

```
int y = x
```

✓ AND ✓

```
x = 344
```

l-values and r-values

An l-value

An **l-value** can be to the left or the right of an equal sign!

What's an example?

x can be an l-value for instance because you can have something like:

```
int y = x
```

✓ AND ✓

```
x = 344
```

An r-value

An **r-value** can be ★ ONLY ★ to the right of an equal sign!

l-values and r-values

An l-value

An **l-value** can be to the left or the right of an equal sign!

What's an example?

x can be an l-value for instance because you can have something like:

```
int y = x
```

✓ AND ✓

```
x = 344
```

An r-value

An **r-value** can be ★ONLY★ to the right of an equal sign!

What's an example?

21 can be an r-value for instance because you can have something like:

```
int y = 21
```

l-values and r-values

An l-value

An **l-value** can be to the left or the right of an equal sign!

What's an example?

x can be an l-value for instance because you can have something like:

```
int y = x
```

✓ AND ✓

```
x = 344
```

An r-value

An **r-value** can be ★ONLY★ to the right of an equal sign!

What's an example?

21 can be an r-value for instance because you can have something like:

```
int y = 21
```

✗ BUT NOT ✗

```
21 = x
```

l-value and r-value PAIN

```
#include <stdio.h>
#include <cmath>
#include <iostream>

int squareN(int& num) {
    return std::pow(num, 2);
}

int main()
{
    int lValue = 2;
    auto four = squareN(lValue);
    auto fourAgain = squareN(2);
    std::cout << four << std::endl;
    return 0;
}
```



l-value and r-value PAIN

```
#include <stdio.h>
#include <cmath>
#include <iostream>
```

is `int& num` an l-value?

```
int squareN(int& num) {
    return std::pow(num, 2);
}

int main()
{
    int lValue = 2;
    auto four = squareN(lValue);
    auto fourAgain = squareN(2);
    std::cout << four << std::endl;
    return 0;
}
```



l-value and r-value PAIN

```
#include <stdio.h>
#include <cmath>
#include <iostream>
```

```
int squareN(int& num) {
    return std::pow(num, 2);
}
```

```
int main()
{
    int lValue = 2;
    auto four = squareN(lValue);
    auto fourAgain = squareN(2);
    std::cout << four << std::endl;
    return 0;
}
```

is int& num an l-value?



l-value and r-value PAIN

```
#include <stdio.h>
#include <cmath>
#include <iostream>
```

```
int squareN(int& num) {
    return std::pow(num, 2);
}
```

```
int main()
{
    int lValue = 2;
    auto four = squareN(lValue);
    auto fourAgain = squareN(2);
    std::cout << four << std::endl;
    return 0;
}
```

is `int& num` an l-value?

It turns out that `num` is an l-value! But Why?

1. Remember what we said about r-values are temporary. Notice that `num` is being passed in by reference!
1. We **cannot** pass in an r-value by reference because they're temporary!

l-value and r-value PAIN

```
#include <stdio.h>
#include <cmath>
#include <iostream>
```

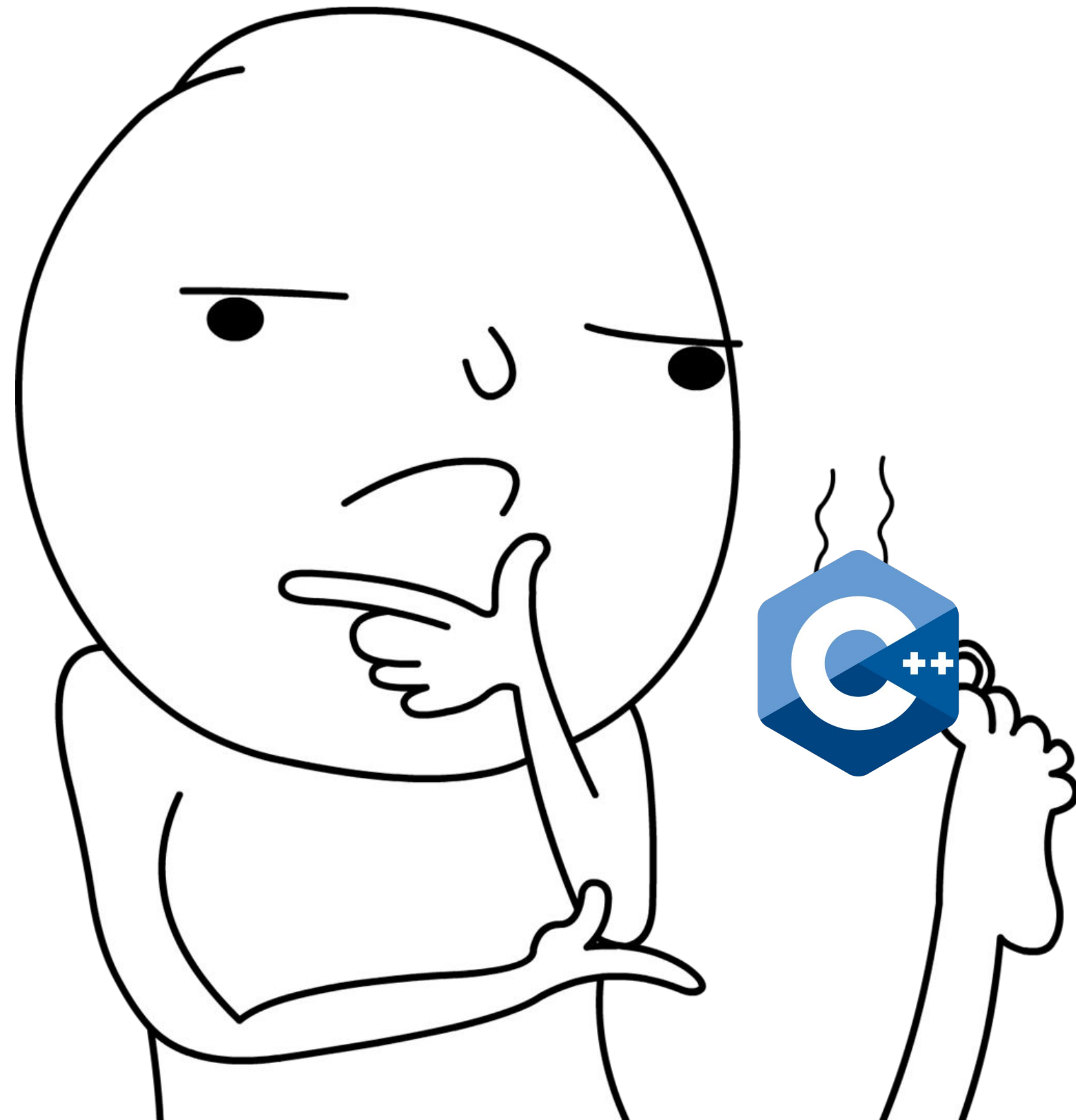
Well what happens?

```
int squareN(int& num) {
    return std::pow(num, 2);
}
```

```
int main()
{
    int lv = 4;
    auto f = squareN(lv);
    auto f2 = squareN(4);
    std::cout << f << f2 << std::endl;
    return 0;
}
```

```
lvalue_pain.cpp:5:5: note: candidate function not viable: expects an lvalue for 1st
argument
int squareN(int& num) {
    ^
1 error generated.
```

What questions do we have?



Plan

1. Initialization
2. References
3. L-values vs R-values
- 4. Const**
5. Compiling C++ programs

const

What?:

A qualifier for objects that declares they cannot be modified – cppreference.com

const

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec{ 1, 2, 3 };    /// a normal vector
    const std::vector<int> const_vec{ 1, 2, 3 };    /// a const vector
    std::vector<int>& ref_vec{ vec };    /// a reference to 'vec'
    const std::vector<int>& const_ref{ vec };    /// a const reference

    vec.push_back(3);
    const_vec.push_back(3);
    ref_vec.push_back(3);
    const_ref.push_back(3);

    return 0;
}
```


const

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec{ 1, 2, 3 };    ///< a normal vector
    const std::vector<int> const_vec{ 1, 2, 3 };    ///< a const vector
    std::vector<int>& ref_vec{ vec };    ///< a reference to 'vec'
    const std::vector<int>& const_ref{ vec };    ///< a const reference

    vec.push_back(3);    ///< this is ok!
    const_vec.push_back(3);
    ref_vec.push_back(3);
    const_ref.push_back(3);

    return 0;
}
```

const

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec{ 1, 2, 3 };    /// a normal vector
    const std::vector<int> const_vec{ 1, 2, 3 };    /// a const vector
    std::vector<int>& ref_vec{ vec };    /// a reference to 'vec'
    const std::vector<int>& const_ref{ vec };    /// a const reference

    vec.push_back(3);    /// this is ok!
    const_vec.push_back(3);    /// no, this is const!
    ref_vec.push_back(3);
    const_ref.push_back(3);

    return 0;
}
```

const

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec{ 1, 2, 3 };    /// a normal vector
    const std::vector<int> const_vec{ 1, 2, 3 };    /// a const vector
    std::vector<int>& ref_vec{ vec };    /// a reference to 'vec'
    const std::vector<int>& const_ref{ vec };    /// a const reference

    vec.push_back(3);    /// this is ok!
    const_vec.push_back(3);    /// no, this is const!
    ref_vec.push_back(3);    /// this is ok, just a reference!
    const_ref.push_back(3);

    return 0;
}
```

const

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec{ 1, 2, 3 };    /// a normal vector
    const std::vector<int> const_vec{ 1, 2, 3 };    /// a const vector
    std::vector<int>& ref_vec{ vec };    /// a reference to 'vec'
    const std::vector<int>& const_ref{ vec };    /// a const reference

    vec.push_back(3);    /// this is ok!
    const_vec.push_back(3);    /// no, this is const!
    ref_vec.push_back(3);    /// this is ok, just a reference!
    const_ref.push_back(3);    /// this is const, compiler error!

    return 0;
}
```

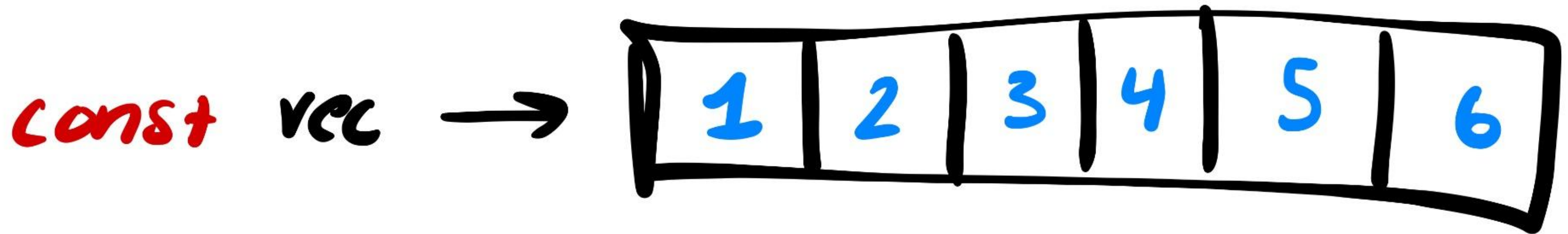
You can't declare a non-const reference to a const variable

```
#include <iostream>
#include <vector>

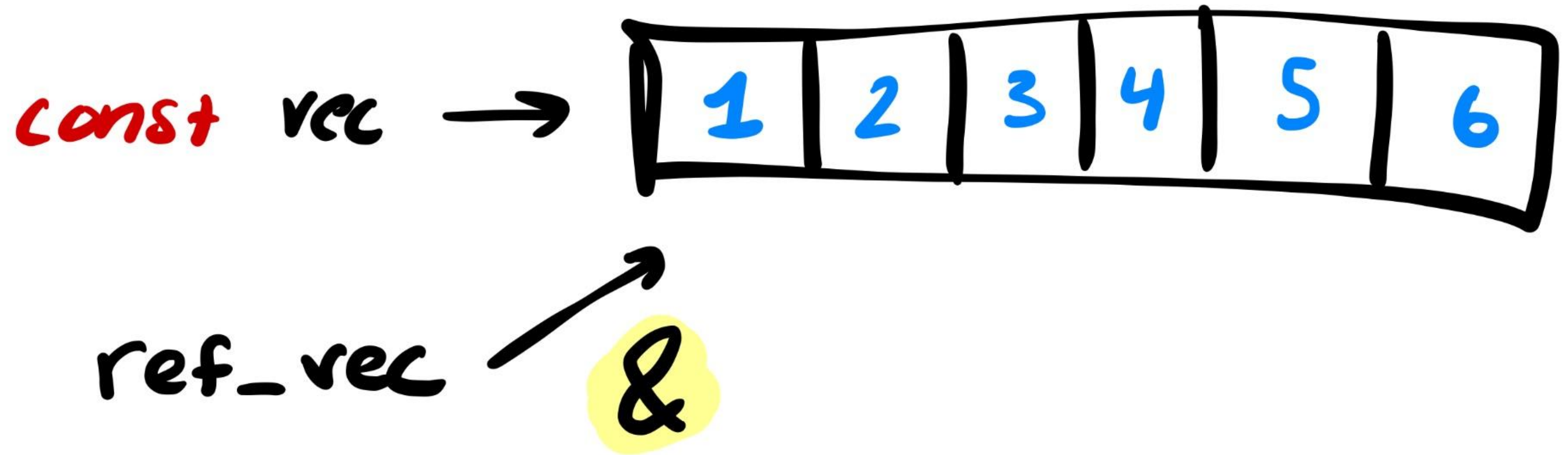
int main()
{
    /// a const vector
    const std::vector<int> const_vec{ 1, 2, 3 };
    std::vector<int>& bad_ref{ const_vec };    /// BAD

    return 0;
}
```

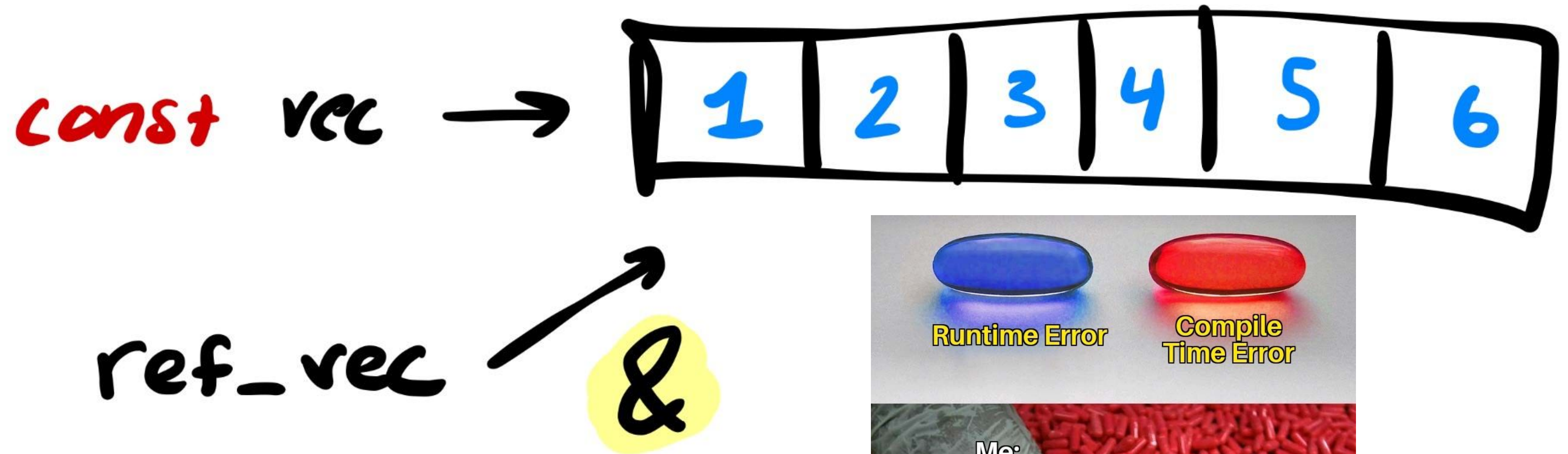
You can't declare a non-const reference to a const variable



You can't declare a non-const reference to a const variable



You can't declare a non-const reference to a const variable



[meme](#)
[sauce](#)

You can't declare a non-const reference to a const variable

```
#include <iostream>
#include <vector>

int main()
{
    /// a const vector
    const std::vector<int> const_vec{ 1, 2, 3 };
    const std::vector<int>& bad_ref{ const_vec }; /// Good!

    return 0;
}
```

Plan

1. Initialization
2. References
3. L-values vs R-values
4. Const
- 5. Compiling C++ programs**

Compiling C++ Programs

Everything you need to know about compiling a program for your first assignment.

We'll be making use of VSCode which makes C++ compilation quite easy.

Compiling C++ Programs

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

Machine Code

```
10110101  
01011010  
10011101  
10110001
```

What you need to know

- C++ is a compiled language

What you need to know

- C++ is a compiled language
- There are computer programs called compilers

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```


What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```

This is the compiler
command

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```

This specifies the c++
version you want to
compile in

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```

This is the source file

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```

This means that you're going to give a specific name to your executable

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp -o main
```

In this case it's main

What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- **Here is how to compile a program using g++**

```
g++ -std=c++11 main.cpp
```

This is also valid, your
executable will be
something like a .out

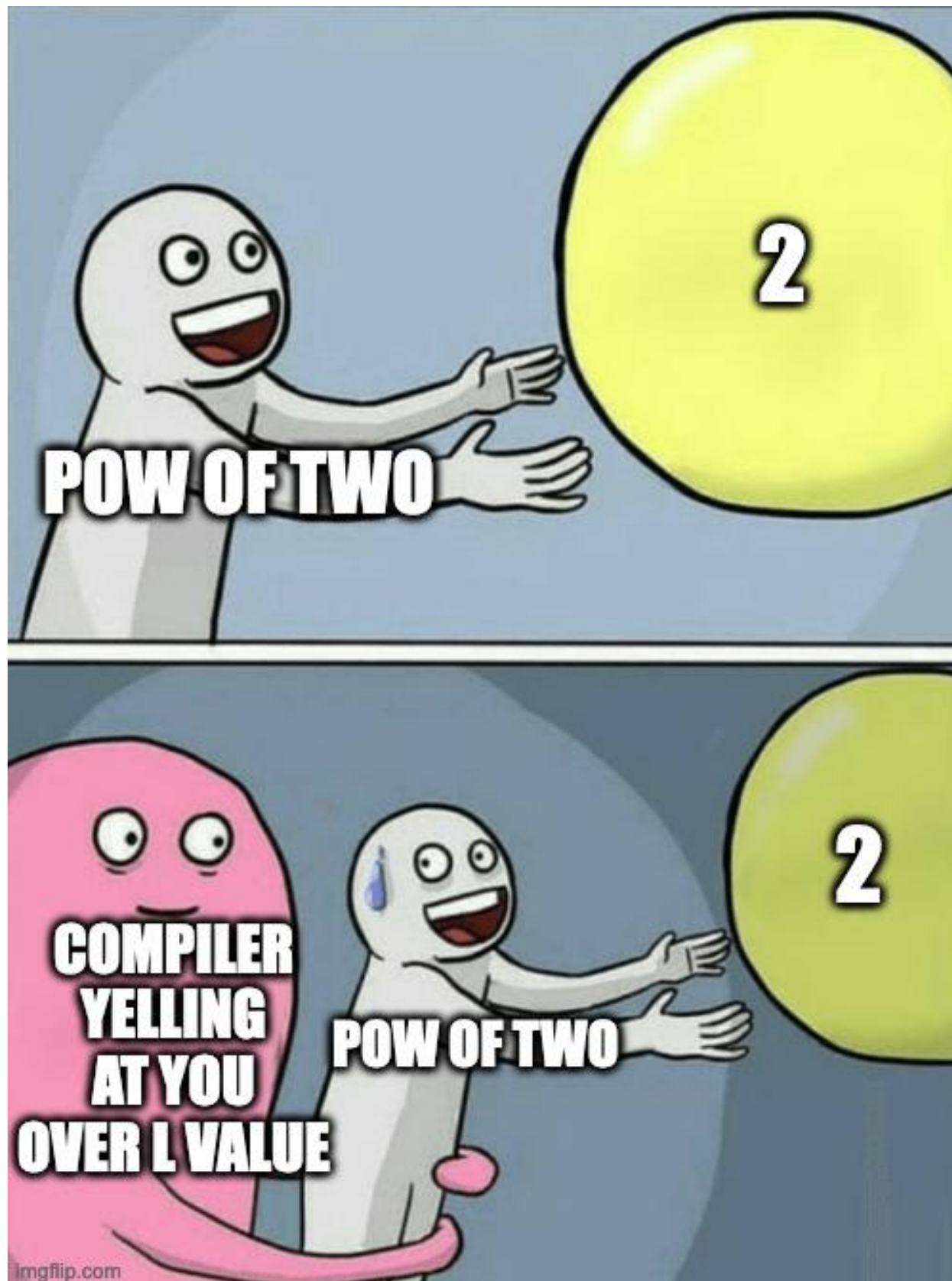
What you need to know

- C++ is a compiled language
- There are computer programs called compilers
- A few popular compilers include clang and g++
- *Here is how to compile a program using g++*

```
g++ -std=c++11 main.cpp
```

- This is all you need for now! We will talk about large project compilation in another lecture and explore things like **CMAKE** and **make**!

A recap of today!



In conclusion

- Use uniform initialization — it works for all types and objects!
- References are a way to alias variables!
- You can only reference an l-value!
- Const is a way to ensure that you can't modify a variable