Initialization

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

How? 🤔:

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

Direct initialization

```cpp
#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;
}
```

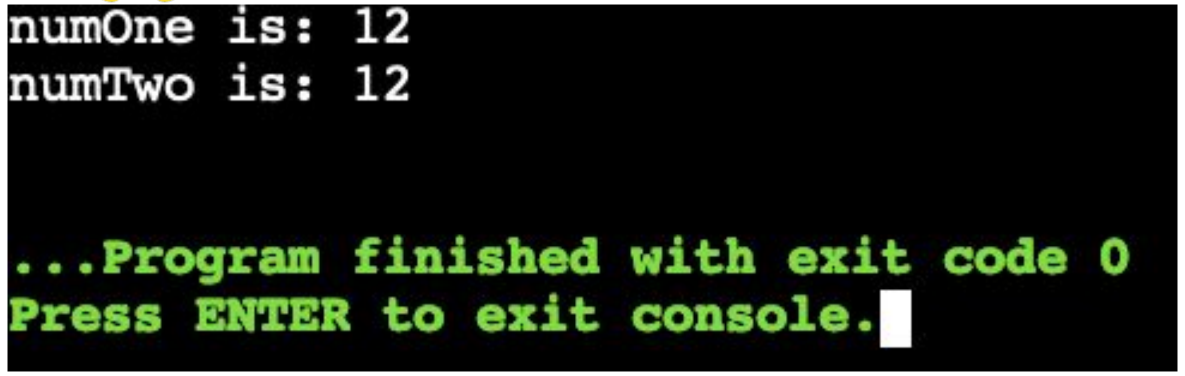
Notice!!:

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.
```

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)

```cpp
#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++ ***<u>does</u>*** 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;
```

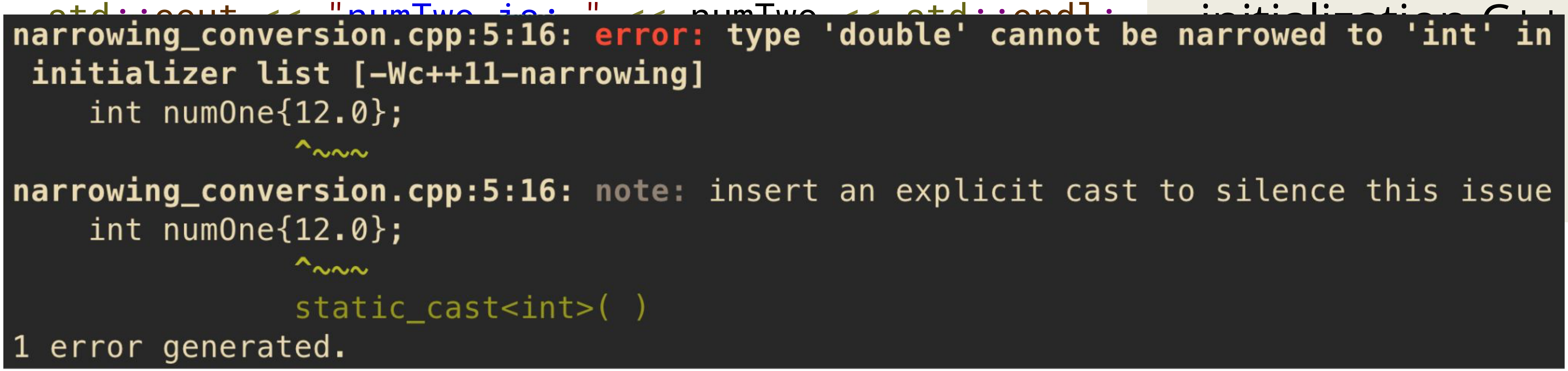
```
}
```

Notice!!:

the curly braces!

With uniform

initialization C++



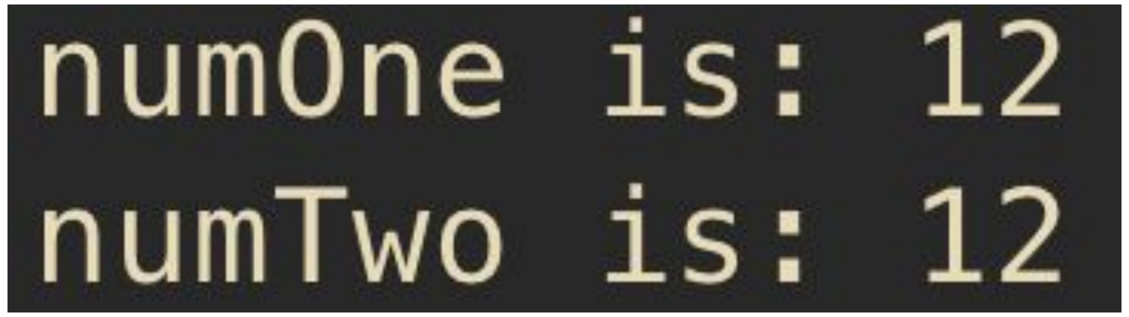
```
narrowing_conversion.cpp:5:16: error: type 'double' cannot be narrowed to 'int' in
 initializer list [-Wc++11-narrowing]
    int numOne{12.0};
               ^~~~
narrowing_conversion.cpp:5:16: note: insert an explicit cast to silence this issue
    int numOne{12.0};
               ^~~~
               static_cast<int>( )
1 error generated.
```

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



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

Uniform initialization (C++11)

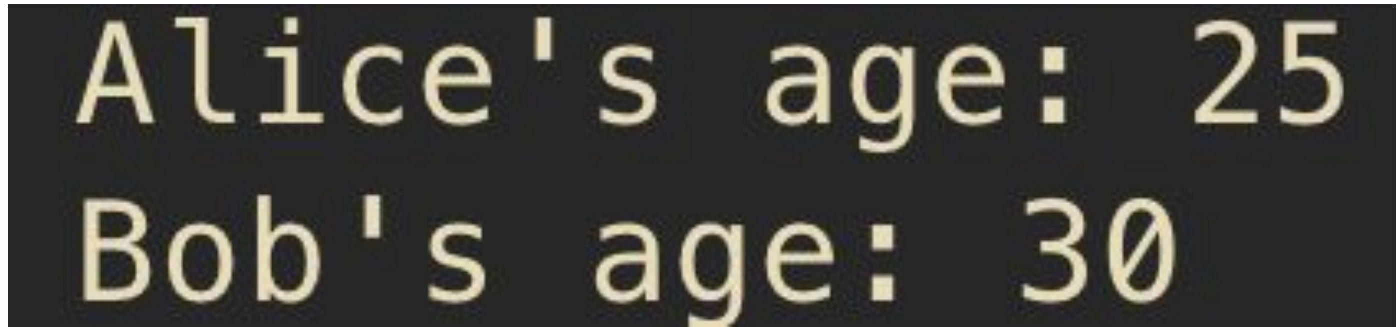
Uniform initialization is awesome because:

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)

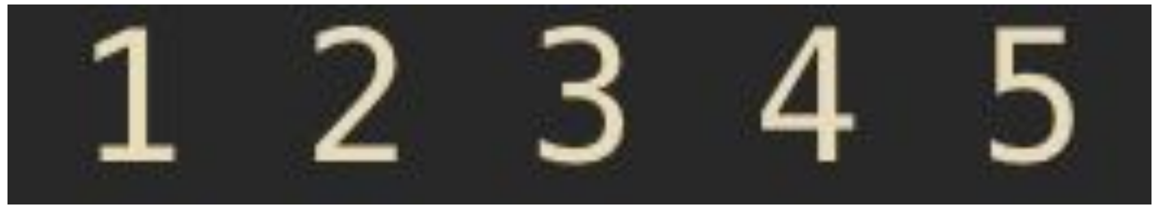
```cpp
#include <iostream>
#include <map>

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)

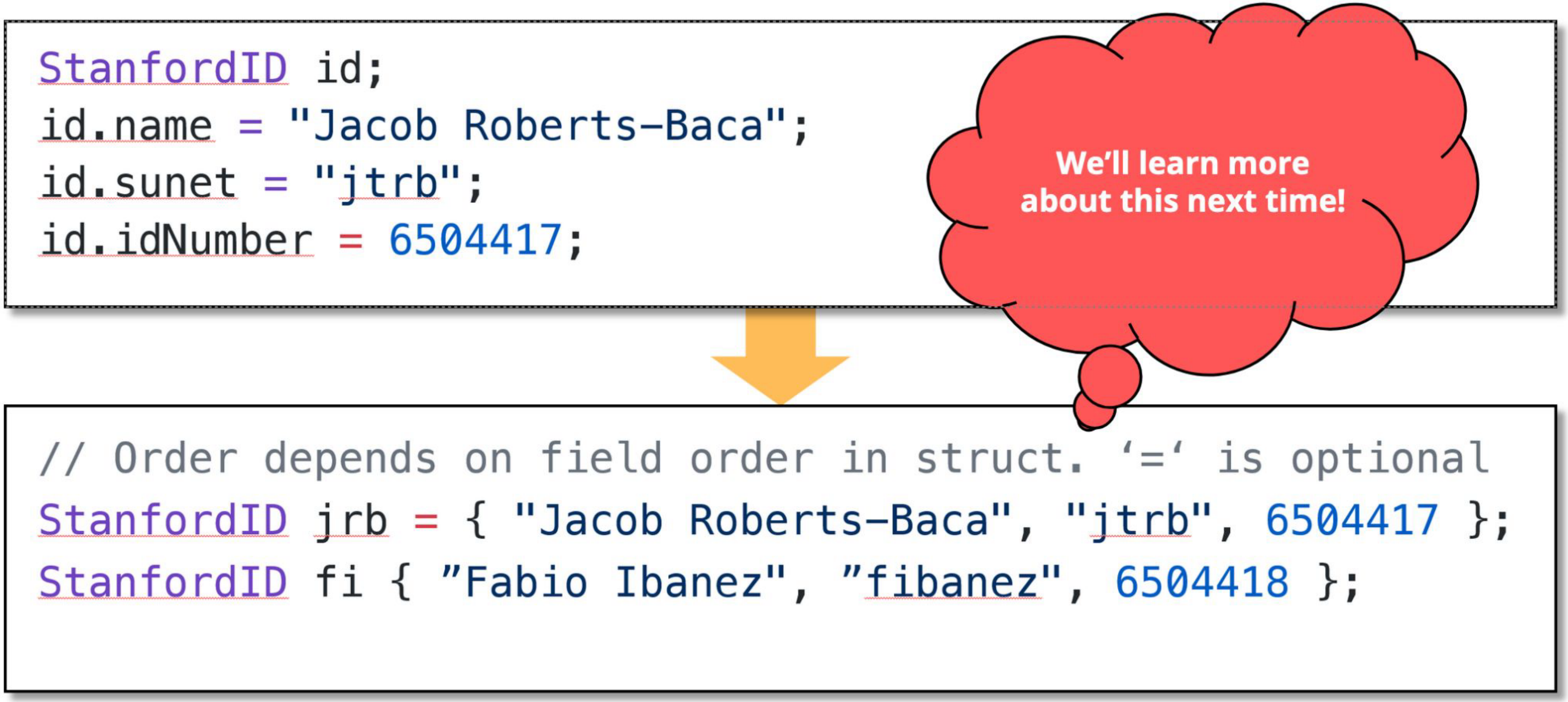
```
#include <iostream>
#include <vector>

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;
}
```



```
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 };
```

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)

```cpp
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;
}
```

Notice - uniform initialization!

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**

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

```cpp
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

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;
}
```

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

A classic reference-copy bug

```cpp
#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: fixed!

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

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

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;
}
```

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!

const

```cpp
#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;
}
```

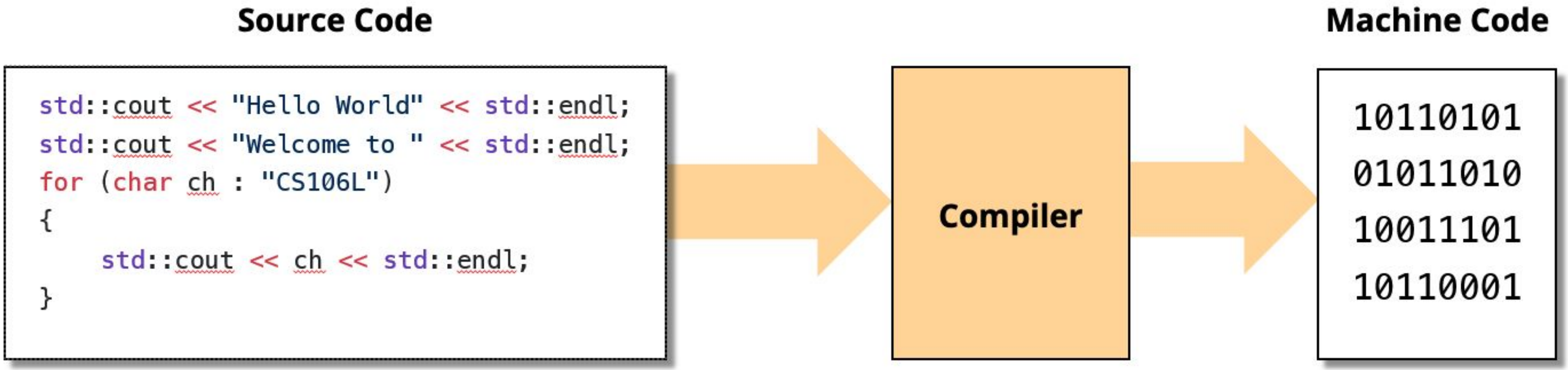
You can’t declare a non-const reference to a const variable

```cpp
#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

```
#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;
}
```

Compiling C++ Programs



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

```
10110101
01011010
10011101
10110001
```

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
```

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

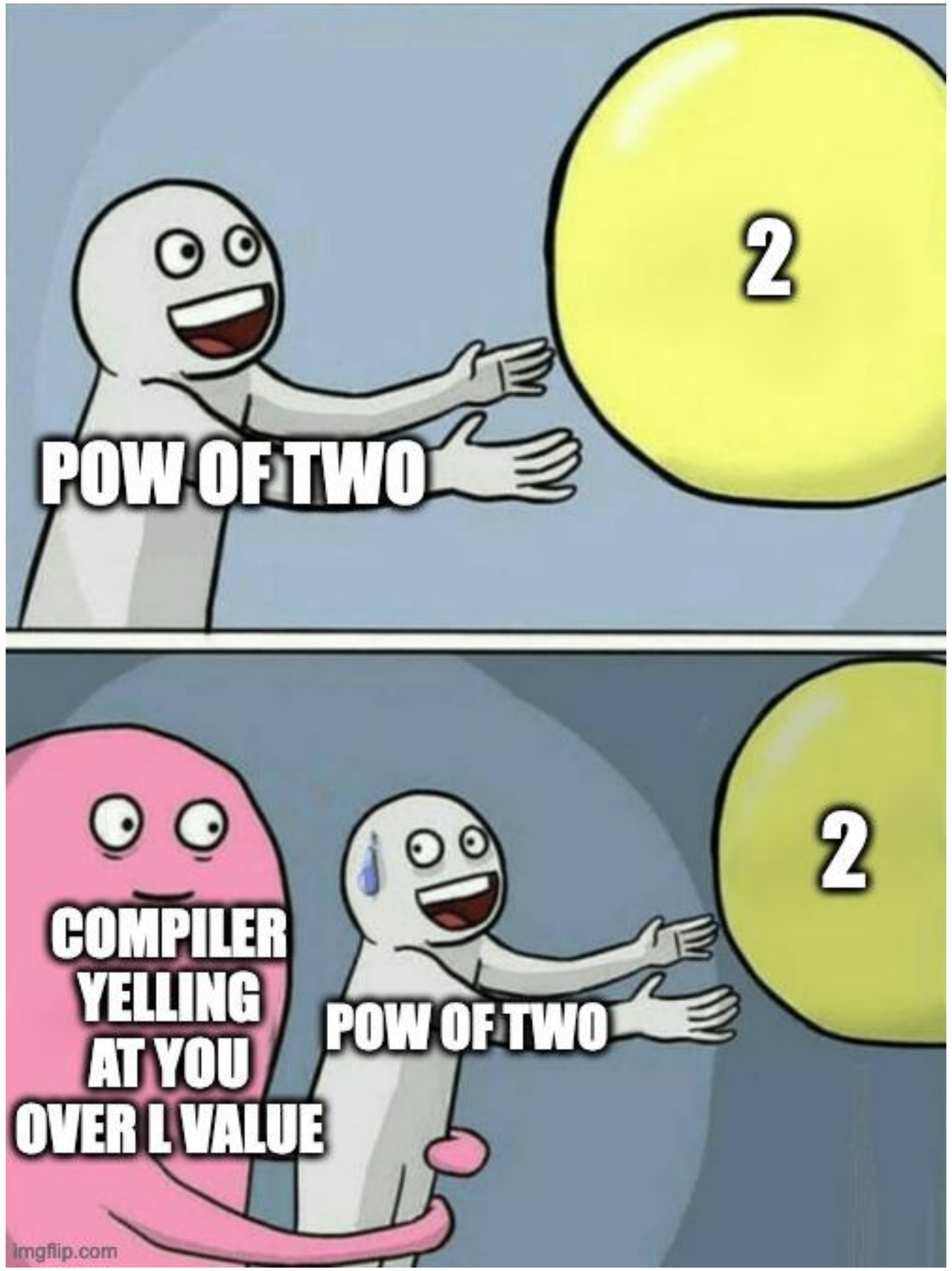
What you need to know

- 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