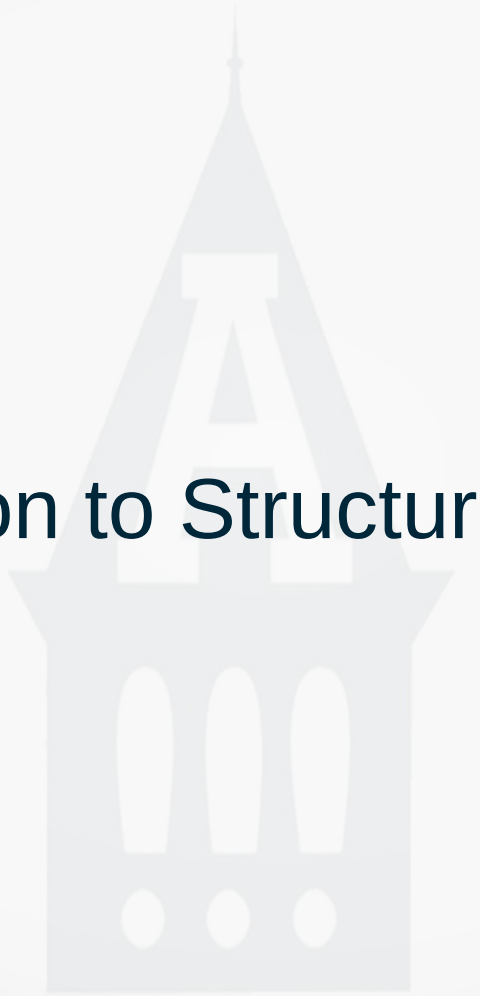
# CS 3460



## Introduction to Structured Bindings

# Structured Bindings

- Allow binding of sub-objects (of a complex type) or array elements to be bound to values in an identifier list.
  - okaaayyy, what is that?

# Structured Bindings

- Allow binding of sub-objects (of a complex type) or array elements to be bound to values in an identifier list.
  - okaaayyy, what is that?
- Easier to show…

```
std::array<int, 4> primes{2, 3, 5, 7};
auto [one, two, three, four] = primes;
std::cout << one << ", " << two << ", " << three << ", " << four << std::endl;
```

- `one`, `two`, `three`, `four` are placed into the current scope
- The left side is called the *identifier-list*
  - `auto` is required
  - Number of elements in identifier-list must match the number of elements on the right

# Structured Bindings

- Can also bind by reference…

```
std::array<int, 4> primes{2, 3, 5, 7};
auto& [one, two, three, four] = primes;
std::cout << one << ", " << two << ", " << three << ", " << four << std::endl;
```

- Number of elements in identifier-list must match the number of elements on the right

# Complex Type

- Let’s take a look a binding from a complex (non-primitive) type

```
class MyClass
{
  public:
    MyClass(int a, double b) :
        m_a(a),
        m_b(b)
    {
    }

    int m_a;
    double m_b;
};
```

- Can bind as...

```
MyClass data(1, 2.2);
auto [x, y] = data;
std::cout << x << ", " << y << std::endl;
```

# Complex Type

- This is called *destructuring*
  - Separating items in the structure into individual items
- Order of class declaration is order of binding in the identifier-list
- Only public members are bound
  - In fact, source can ***only*** have public members!

# Destructuring Example

- Consider the following code...

```
std::map<int, std::string> lookup{
    {1, "Logan"},
    {2, "Smithfield"},
    {3, "Hyrum"},
    {4, "Nibley"},
    {5, "Hyde Park"}};
```

```
for (const auto& item : lookup)
{
    std::cout << item.first << " - " << item.second << std::endl;
}
```

# Destructuring Example

- Consider the following code...

```
std::map<int, std::string> lookup{
     {1, "Logan"},
     {2, "Smithfield"},
     {3, "Hyrum"},
     {4, "Nibley"},
     {5, "Hyde Park"}};
```

```
for (const auto& item : lookup)
{
     std::cout << item.first << " - " << item.second << std::endl;
}
```

```
for (const auto& [key, value] : lookup)
{
     std::cout << key << " - " << value << std::endl;
}
```

# Returning Multiple Values

- Three historical approaches…
  1. Use reference parameters
     - Requires passing in uninitialized arguments
     - Have to declare before use
  2. Use `std::tuple` combined with `std::tie`
  3. Use custom data structure with public only members

- Example of `std::tie`

```
int d;
double e;
std::string f;
std::tie(d, e, f) = returnMultipleItems();
std::cout << d << ", " << e << ", " << f << "" << std::endl;
```

# Returning Multiple Values

- Here is how to do it with structured bindings...

```
auto returnMultipleItems()
{
    return std::make_tuple(1, 2.0, std::string("Hi Mom"));
}
```

```
auto [a, b, c] = returnMultipleItems();
std::cout << a << ", " << b << ", " << c << "" << std::endl;
```