# Lifetimes and References

EECS 211 Winter 2018

# Scope

A scope is a region of program text:

- global scope (outside any language construct)
- namespace scope (outside everything but a namespace)
- class scope (inside a class or struct)
- local scope (between { and } braces; includes function scope)
- statement scope (loop variable in a for)

They nest!

# Scope

A scope is a region of program text:

- global scope (outside any language construct)
- namespace scope (outside everything but a namespace)
- class scope (inside a class or struct)
- local scope (between { and } braces; includes function scope)
- statement scope (loop variable in a for)

They nest! Useful because:

- Declarations from outer scopes are visible in inner scopes
- Declarations from inner scopes are not visible in outer scopes
- (Exception: class stuff)

```
int number of bees = 0; // global scope — visible everywhere
void increase bees(); // also global scope
void buzz(int n)
                   // buzz is global, n is local to buzz
ł
    if (number of bees > n) {
         cout << 'b':
         for (int i = 0; // i has statement scope
             i < number of bees;
             ++i)
             cout << 'z':
    }
    increase_bees();
}
```

```
int number of bees = 0; // global scope — visible everywhere
void increase bees(); // also global scope
void buzz(int n)
                    // buzz is global, n is local to buzz
ł
    if (number of bees > n) {
         cout << 'b':
         for (int i = 0; // i has statement scope
             i < number of bees;
             ++i)
             cout << 'z':
    }
    increase_bees();
}
```

```
int number of bees = 0; // global scope — visible everywhere
void increase bees(); // also global scope
void buzz(int n)
                   // buzz is global, n is local to buzz
ł
    if (number of bees > n) {
         cout << 'b':
         for (int i = 0; // i has statement scope
             i < number of bees;
             ++i)
             cout << 'z':
    }
    increase_bees();
}
```

```
int number of bees = 0; // global scope — visible everywhere
void increase bees(); // also global scope
void buzz(int n)
                   // buzz is global, n is local to buzz
ł
    if (number of bees > n) {
         cout << 'b':
         for (int i = 0; // i has statement scope
             i < number of bees;
             ++i)
             cout << 'z':
    }
    increase_bees();
}
```

```
int number of bees = 0; // global scope — visible everywhere
void increase bees(); // also global scope
void buzz(int n)
                   // buzz is global, n is local to buzz
ł
    if (number of bees > n) {
         cout << 'b':
         for (int i = 0; // i has statement scope
             i < number of bees;
             ++i)
             cout << 'z':
    }
    increase_bees();
}
```

Variable names declared in different scopes refer to different objects:

bool is\_even(int n) { return n % 2 == 0; } bool is\_odd(int n) { return n % 2 == 1; }

There are two unrelated objects named n above

Variable names declared in different scopes refer to different objects:

bool is\_even(int n) { return n % 2 == 0; }
bool is\_odd(int m) { return m % 2 == 1; }

There were two *unrelated* objects named n above

#### Lifetimes example

```
double mean(vector<double> w)
ł
    double result = 0;
    for (double wi : w) result += wi;
    return result / w.size();
}
double variance(vector<double> v)
{
    double m = mean(v), total = 0;
    for (double vi : v) total += (vi - m) * (vi - m);
    return total / v.size();
}
double std dev(vector<double> u)
{ return my sqrt(variance(u)); }
```

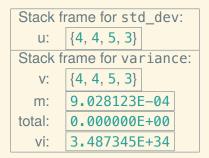
v outlives w, m, and total,

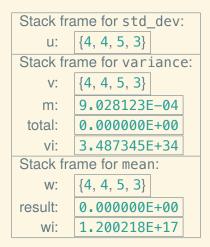
v outlives w, m, and total, which outlive vi,

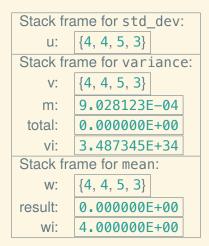
v outlives w, m, and total, which outlive vi, which outlives w and result,

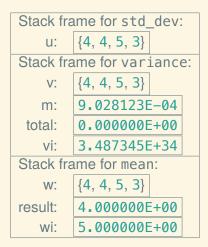
v outlives w, m, and total, which outlive vi, which outlives w and result, which in turn outlive wi.

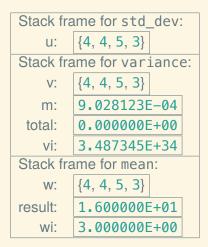
Stack frame for std\_dev: u: {4, 4, 5, 3}

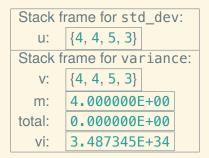


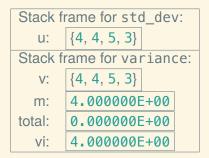


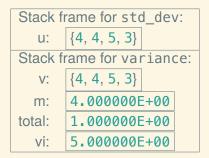






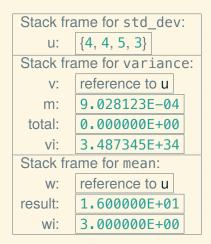






# Const reference example

```
double mean(const vector<double>& w)
ł
    double result = 0;
    for (double wi : w) result += wi;
    return result / w.size();
}
double variance(const vector<double>& v)
{
    double m = mean(v), total = 0;
    for (double vi : v) total += (vi - m) * (vi - m);
    return total / v.size();
}
double std dev(vector<double> u)
{ return my sqrt(variance(u)); }
```



# Copying example: banking

Function **deposit** gets a copy of the vector, and returns a copy of the copy:

```
struct Account {
    double balance;
    std::string owner;
};
```

{

}

```
std::vector<Account> deposit(std::vector<Account> accts,
long acct_number,
unsigned long amount)
```

```
check_deposit(acct_number);
accts[acct_number].balance += amount;
return accts;
```

```
13
```

# Reference example: banking

Function **deposit** *borrows* a reference to the vector and operates on that:

```
struct Account {
    double balance:
    std::string owner;
};
void deposit(std::vector<Account>& accts,
            long acct number,
            unsigned long amount)
ł
    check deposit(acct number);
    accts[acct number].balance += amount;
}
```

# Harmful reference example

You can only borrow something for as long as it exists:

```
std::vector<double>& get_input()
{
    std::vector<double> result;
    i
    return result;
}
```

The vector **result** exists only as long as function **get\_input** is active. So by the time the caller gets it, the reference refers to an object that no longer exists.

# Guidelines for borrowing

To avoid harmful (undefined) behavior:

- Most references should be parameters.
  - The caller should guarantee that the object exists through the call.
  - ► The callee should not save a reference to the object.

# Guidelines for borrowing

To avoid harmful (undefined) behavior:

- Most references should be parameters.
  - The caller should guarantee that the object exists through the call.
  - The callee should not save a reference to the object.
- Returned references are borrowed parts of objects that were passed in.
  - For example, a vector index operation returns a reference to an element.
  - So the caller knows that the part object lives as long as the whole.

- To CLion! -