

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)

Scope example

```
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
{
    if (number_of_bees > n) {
        cout << 'b';

        for (int i = 0; // i has statement scope
              i < number_of_bees;
              ++i)
            cout << 'z';
    }

    increase_bees();
}
```

Scope example

```
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
{
    if (number_of_bees > n) {
        cout << 'b';

        for (int i = 0; // i has statement scope
             i < number_of_bees;
             ++i)
            cout << 'z';
    }

    increase_bees();
}
```

Scope example

```
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
{
    if (number_of_bees > n) {
        cout << 'b';

        for (int i = 0; // i has statement scope
             i < number_of_bees;
             ++i)
            cout << 'z';
    }

    increase_bees();
}
```

Scope example

```
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
{
    if (number_of_bees > n) {
        cout << 'b';

        for (int i = 0; // i has statement scope
             i < number_of_bees;
             ++i)
            cout << 'z';
    }

    increase_bees();
}
```

Scope example

```
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
{
    if (number_of_bees > n) {
        cout << 'b';

        for (int i = 0; // i has statement scope
             i < number_of_bees;
             ++i)
            cout << 'z';
    }

    increase_bees();
}
```


Local scope is local

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

Local scope is local

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)
{
    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)); }
```

Object lifetimes are nested!

v outlives w, m, and total,

Object lifetimes are nested!

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

Object lifetimes are nested!

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

Object lifetimes are nested!

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

Stack layout for nested scopes

Stack frame for std_dev:

u: {4, 4, 5, 3}

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34
Stack frame for mean:	
w:	{4, 4, 5, 3}
result:	0.000000E+00
wi:	1.200218E+17

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34
Stack frame for mean:	
w:	{4, 4, 5, 3}
result:	0.000000E+00
wi:	4.000000E+00

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34
Stack frame for mean:	
w:	{4, 4, 5, 3}
result:	4.000000E+00
wi:	5.000000E+00

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34
Stack frame for mean:	
w:	{4, 4, 5, 3}
result:	1.600000E+01
wi:	3.000000E+00

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	4.000000E+00
total:	0.000000E+00
vi:	3.487345E+34

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	4.000000E+00
total:	0.000000E+00
vi:	4.000000E+00

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	{4, 4, 5, 3}
m:	4.000000E+00
total:	1.000000E+00
vi:	5.000000E+00

Const reference example

```
double mean(const vector<double>& w)
{
    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)); }
```

Stack layout for nested scopes

Stack frame for std_dev:	
u:	{4, 4, 5, 3}
Stack frame for variance:	
v:	reference to u
m:	9.028123E-04
total:	0.000000E+00
vi:	3.487345E+34
Stack frame for mean:	
w:	reference to u
result:	1.600000E+01
wi:	3.000000E+00

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;
}
```

Reference example: banking

Function `deposit` *borrow*s 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)
{
    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;  
    :  
    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!* –