Getting Deeper into

C++ Semantics

EECS 230

Spring 2016

The plan

- On language technicalities
- Declarations
  - definitions
  - headers
  - scope
- Functions
  - declarations & definitions
  - formal and actual parameters
  - pass by value, reference, and const reference
- Namespaces
  - using declarations

Language technicalities

C++ is a necessary evil:

Language technicalities

C++ is a necessary evil:

- Evil: because language details can distract from our main task: programming

Language technicalities

C++ is a necessary evil:

- Evil: because language details can distract from our main task: programming
- Necessary: because you can't program without a language

Language technicalities

C++ is a necessary evil:

- Evil: because language details can distract from our main task: programming
- Necessary: because you can't program without a language

This week we learn grammar and vocabulary

Technicalities

- Like in natural language (*e.g.,* English):
  - Don’t get hung up on minor issues
  - There’s more than one way to say everything
- Most design and programmimg concepts are universal, but technicalities are particular

Declarations

A declaration introduces a *name* into a *scope* (region of code):

- gives a type for the named object
- sometimes includes an initializer
- must come before use

Declarations

A declaration introduces a *name* into a *scope* (region of code):

- gives a type for the named object
- sometimes includes an initializer
- must come before use

Examples:

- `int a = 7;`
- `int b;`
- `vector<string> c;`
- `double my_sqrt(double);`

Headers

Declarations are frequently introduced through *headers*:

```
int main()
{
    cout << "Hello, world!\n";
}
```

Error: unknown identifier cout

Headers

Declarations are frequently introduced through *headers*:

```
#include ”eecs230.h”

int main()
{
    cout << "Hello, world!\n";
}
```

`eecs230.h` has declarations for cout, operator<<, etc.

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
```

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
int b;        // but why?
```

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
int b;          // but why?
vector<double> v;
```

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
int b;          // but why?
vector<double> v;
double square(double x) { return x * x; }
```

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
int b;        // but why?
vector<double> v;
double square(double x) { return x * x; }
struct Point { int x, y; };
```

Definitions

A declaration that (also) fully specifies the declarandum is a *definition*.

Examples:

```
int a = 5;
int b;        // but why?
vector<double> v;
double square(double x) { return x * x; }
struct Point { int x, y; };
```

Examples of non-definition declarations:

```
extern int b;
double square(double);
struct Point;
```

Declarations and definitions

	declarations	definitions
may be repeated	yes	no
must come before use	yes	no

Why both?

To refer to something, we need only its declaration

We can hide its definition, or save it for later

In large programs, declarations go in header files to ease sharing

Declaration example

```
double my_sqrt(double input, double epsilon)
{
      ⋮
}

int main()
{
      ··· my_sqrt(d, eps) ···
}
```

Declaration example

```
int main()
{
    ··· my_sqrt(d, eps) ···        // unknown identifier
}

double my_sqrt(double input, double epsilon)
{
    ⋮
}
```

Declaration example

```
double my_sqrt(double, double);

int main()
{
    ··· my_sqrt(d, eps) ···
}

double my_sqrt(double input, double epsilon)
{
    ⋮
}
```

Library declaration example

In `my_math.h`:

```
double my_sqrt(double, double);
```

In `my_math.cpp`:

```
#include ”my_math.h”

double my_sqrt(double input, double epsilon)
{ · · · }
```

In some other `.cpp` source file:

```
#include ”my_math.h”

int f() { · · · my_sqrt(d, eps) · · · }
```

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; }

void swap(int& a, int& b)
{
    int n = a;
    a = b;
    b = n;
}
```

There are three *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; }

void swap(int& a, int& b)
{
    int n = a;
    a = b;
    b = n;
}
```

There are three *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; }

void swap(int& a, int& b)
{
    int temporary = a;
    a = b;
    b = temporary;
}
```

There are three *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; }

void swap(int& n, int& m)
{
    int temporary = n;
    n = m;
    m = temporary;
}
```

There are three *unrelated* objects named n above

Guidelines for scope

- Declare variables in the smallest possible scope
- Global variables:
    - In general: avoid if you can
    - In EECS 230: avoid (because you can)

Recap: Why functions?

- Chunk a program into manageable pieces (“divide and conquer”)
- Map to our understanding of the problem domain
- Make the program easier to read
- Facilitate code reuse
- Ease testing and distribution of labor

Function syntax

Declaration:

```
return_type fun_name(type1 formal1, type2 formal2, …);
```

(You can omit formal argument names in a declaration)

Function syntax

Declaration:

```
return_type fun_name(type1 formal1, type2 formal2, ...);
```

(You can omit formal argument names in a declaration)

Definition:

```
return_type fun_name(type1 formal1, type2 formal2, ...)
{
    stm; ···
    return expr;
}
```

Using a function

If a function is defined thus:

```
ret f(type1 formal1, type2 formal2, …) body
```

(where *body* is a block or try-catch block)

Using a function

If a function is defined thus:

```
ret f(type1 formal1, type2 formal2, …) body
```

(where *body* is a block or try-catch block)

then calling it as f(actual1, actual2, …) means:

1. run its *body* with actual1 copied to formal1, actual2 copied to formal2, etc.
2. replace the function call with the value returned by the function

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << dist(1, 2, 4, -2) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
      cout << ({
            double x1 = 1; double y1 = 2;
            double x2 = 4; double y2 = -2;
            return sqrt(square(x1 - x2) + square(y1 - y2));
      }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(square(1 - x2) + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(square(1 - 4) + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
	cout << ({
		double x1 = 1; double y1 = 2;
		double x2 = 4; double y2 = -2;
		return sqrt(square(-3) + square(y1 - y2));
	}) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(({
            double f = -3; return f * f;
        }) + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(({
            double f = -3; return -3 * -3;
        }) + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(({
            double f = -3; return 9;
        }) + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(9 + square(y1 - y2));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(9 + square(4));
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(9 + 16);
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return sqrt(25);
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << ({
        double x1 = 1; double y1 = 2;
        double x2 = 4; double y2 = -2;
        return 5;
    }) << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    cout << 5 << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    // printed "5"
    cout << '\n';
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    // printed "5\n"
    cout;
}
```

Example of using a function

```
double square(double f) { return f * f; }

double dist(double x1, double y1, double x2, double y2)
{ return sqrt(square(x1 - x2) + square(y1 - y2)); }


int main()
{
    // printed "5\n"
}
```

Parameter modes

Ordinary parameters (*e.g.,* `f(int x, string s)`) get a copy of the actual argument object:

- The callee can make changes, but
- the caller won't see them.

Parameter modes

Ordinary parameters (*e.g.,* `f(int x, string s)`) get a copy of the actual argument object:

- The callee can make changes, but
- the caller won't see them.

Reference parameters (*e.g.,* `f(int& x, string& s)`) get a reference to the actual argument object:

- The callee can make changes, and
- the caller *will* see them.

Parameter modes

Ordinary parameters (*e.g.,* `f(int x, string s)`) get a copy of the actual argument object:

- The callee can make changes, but
- the caller won't see them.

Reference parameters (*e.g.,* `f(int& x, string& s)`) get a reference to the actual argument object:

- The callee can make changes, and
- the caller *will* see them.

Const reference parameters (*e.g.,* `f(const int& x, const string& s)`) get an unchangeable reference to the actual argument object:

- The callee *cannot* make changes, but
- no copying is necessary.

Pass by copy vs. pass by reference

```
int f (int   a) { a = a + 1; return a; }
int g(int& a) { a = a + 1; return a; }

int main()
{
    int x = 0; y = 0;

    cout << "f(x) = " << f(x)  << '\n';
    cout << "g(y) = " << g(y) << '\n';
    cout << "x = "     << x    << '\n';
    cout << "y = "     << y    << '\n';
}
```

What gets printed?

Parameter modes decision tree

```
if (the function wants to change the parameter) {
    pass by reference
} else if (the parameter is small, like an int or double) {
    pass by copy (i.e., an ordinary parameter)
} else {
    pass by const reference
}
```

References aren’t just for arguments

```
vector<string> v;

for (string s : v) …                    // copies each element
for (string& s : v) …                   // doesn’t copy
for (const string& s : v) …             // and doesn’t allow changes
```

Namespaces

What if Anne and Bob both want a function swizzle(int)?

```
namespace anne {
void swizzle(int);
}

namespace bob {
void swizzle(int);
}
```

Namespaces

What if Anne and Bob both want a function swizzle(int)?

```
namespace anne {
void swizzle(int);
}

namespace bob {
void swizzle(int);
}
```

From outside, we can refer to anne::swizzle and bob::swizzle

using

What if we almost always want anne::swizzle and are tired of typing anne:: over and over?

using

What if we almost always want anne::swizzle and are tired of typing anne:: over and over? Two options:

- A using *declaration* brings one name into scope:

```
using anne::swizzle;
```

using

What if we almost always want anne::swizzle and are tired of typing anne:: over and over? Two options:

- A using *declaration* brings one name into scope:

  ```
  using anne::swizzle;
  ```

- A using *directive* brings a whole namespace into scope:

  ```
  using namespace anne;
  ```

using

What if we almost always want `anne::swizzle` and are tired of typing `anne::` over and over? Two options:

- A `using` *declaration* brings one name into scope:

  ```
  using anne::swizzle;
  ```

- A `using` *directive* brings a whole namespace into scope:

  ```
  using namespace anne;
  ```

The C++ standard library is defined in namespace `std`, so ordinarily you'd refer to `std::cout`, `std::string`, etc.

using

What if we almost always want anne::swizzle and are tired of typing anne:: over and over? Two options:

- A using *declaration* brings one name into scope:

  ```
  using anne::swizzle;
  ```

- A using *directive* brings a whole namespace into scope:

  ```
  using namespace anne;
  ```

The C++ standard library is defined in namespace std, so ordinarily you'd refer to std::cout, std::string, etc.

But `eecs230.h` contains a using namespace std; directive

(Rude! Don't put using directives in headers.)

Next time

Defining your own types!