An Introduction to Programming
through C++

with

Manoj Prabhakaran

Lecture 11

Anatomy (and Dynamics) of a Program (ctd.)

Namespaces, Scopes, Lifetimes

Based on material developed by Prof. Abhiram G. Ranade

Today

- Anatomy: What it looks like (Syntax)
  - Declarations, functions, expressions, … organised into files and header files
- Dynamics: How it behaves (Semantics)
  - Conditional execution, accessing variables, function call and the stack, …
- Today
  - Namespaces: Recap and examples
  - Scope of a variable: Recap and examples
  - Lifetime of a variable
  - Static variables

Namespaces

Recap

- Suppose you write a function `to_string` as follows:

```
#include <simplecpp>
string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    cout << to_string(a) << " vs. " << to_string(b) << endl;
}
```

```
non-zero vs. 1
```

- Why did this happen?
  - Standard library already has a function (included via `<simplecpp>`)
    `string to_string (int)`
    (but no function that takes a short — so it was not an error to define ours)
  - For the call `to_string(b)`, the compiler used this library function (which is a better fit than using our function which takes a `short`)

Recap

Namespaces

- To keep entities (functions, types, variables) in a library separate from ours
  - `to_string` vs. `std::to_string`
- `<simplecpp>` has a statement `using namespace std;` which made all the entities in std namespace available without the qualifier `std::`
- We shall instead use the standard header `<iostream>`

Risky!

```
#include <iostream>
std::string to_string(short x) { return x==0 ? "zero" : "non-zero"; }
int main() {
    short a = 1; int b = 1;
    std::cout << to_string(a) << " vs. " << to_string(b) << std::endl;
}
```

Invokes our `to_string`, with b cast into a short.
Only `std::to_string` invokes the one from the library.

Namespaces

- Conventions to avoid unexpected conflicts
  - Every library should (and typically does) keep the entities they define within a separate (hopefully unique) namespace
    - E.g., std, boost, …
  - Programmers access entities in a library by explicitly specifying the namespace (e.g. `std::to_string(…)`, `std::string`, etc.)
  - But if desired, a programmer can shorten `nspace::entity` to just `entity` (say, because it is used in a lot of places in the program), by adding the statement `using nspace::entity;`
  - Alternately, one can write `using namespace nspace;` and the prefix `nspace::` can be dropped for *all the entities* in `nspace` (Bad idea!)

Namespaces

- How does a library place its entities in a namespace?
  - Declare them inside a block of the form

    `namespace nspace { ... }`
- In a file, the same namespace can have multiple namespace blocks
  - Typically, included from different header files provided by the library
- Within the namespace block, the name prefix can be omitted for already defined entities (e.g., P instead of `geo::P`)

```
// global namespace (empty name)
int P; // a global variable, ::P
namespace geo {
struct P {int x,y;}; // geo::P
}
...
namespace geo {
void move(P&,P); // geo::move
                 // geo::P referred
                 // to as simply P
}

...

void geo::move(geo::P& p, geo::P d) {
  p.x += d.x; p.y += d.y;
}
```

Example

Demo

numbers.h

```
namespace num {
int GCD(int, int);
int LCM(int, int);
bool coprimes(int,int);
bool covers(int w, int x);
bool PFE(int w, int x);
int reduce(int w, int x);
}
```

numbers.cpp

```
#include "numbers.h"
#include <cmath>

int num::LCM(int a, int b) {
    return std::abs(a*b)/GCD(a,b); // GCD is num::GCD
}
bool num::coprimes(int a, int b) {
    return GCD(a,b) == 1;
}
...
```

prog.cpp

```
#include <iostream>
#include "numbers.h"

using std::cout; using std::cin; using std::endl;

int main() {
  cout << "Enter 2 positive numbers: ";
  int a, b; cin >> a >> b;
  if (a<=0 || b<=0) return -1;
  cout << (num::PFE(a,b) ? "":"Not ") << "PFE" << endl;
  cout << "GCD(a,b) = " << num::GCD(a,b) << endl;
}
```

```
$ g++ -c prog.cpp         # this produces prog.o
$ g++ -c numbers.cpp      # this produces numbers.o
$ g++ prog.o numbers.o    # this produces a.out
```

Recap

Scope of Variables

- In C++, a variable can be used only where its declaration is "visible"
  - Visible only within the "block" it is declared in
  - And only after it is declared
  - Scope of a variable: region in the code where it is visible
- A variable cannot be declared twice within the same block
  - However can declare a new variable with the same name (but possibly a different type) in a "sub-block"
  - In its scope, the new variable "shadows" the old one

```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
      // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

scope of x

Recap

Scope of Variables

```
void f(int x) {
   ...
}
```

```
{
   ...
}
```

```
for(int x=0;;) {
   ...
}
```

```
while(condition) {
   ...
}
```

```
if(condition) {
   ...
}
```

```
{
  {
    // not visible here (before declaration)
    int x;
    // visible here
    {
      // visible here
    }

    // visible here
  }
  // not visible here (outside the block)
}
```

scope of x

- A few different kinds of blocks (more later):
  - A function's body (including parameter declarations)
  - A block of statements enclosed in braces
  - A for loop (including declarations in the initialisation)
  - A while or do-while statement (condition can have declarations)
  - If-Else statement (condition can have declarations; visible in both if & else parts)

Scope of Variables

Demo

```
int g; // a global variable. remains visible till the end of the file
...

void f(int x) { // x is visible inside the body of the function
  int y; // visible from here till the end of the function
  for(int g=x; g<3; g–) { // a new local g! visible till
    ...                    // the end of the for statement.
  } // now this g goes out of scope. global g visible again.
  {  // start of a new scope
    g = x + 1;  // this refers to the global g
    float g; // this is a different g! global g not visible.
  }  // now this g goes out of scope. global g visible again.

  g++;  // global g
}        // here x, y go out of scope.
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
  - i.e., control goes outside the block in which it was defined

```
{
  int c=0; // c "created" here
  while(c<12){
     int x = 2;  // x "created" in each iteration
     x++; c += x;
  } // at the end of each iteration x "destroyed"
} // here c is "destroyed"
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
  - i.e., control goes outside the block in which it was defined

```
for(int c=0 /* c "created" here */; c<12; ) {
  int x = 2;   // x "created" in each iteration
  x++; c += x;
}  // at the end of each iteration x "destroyed", but c is alive
// on exiting the loop, c is "destroyed"
```

Lifetime of Variables

- A variable is *created* (a "box" allocated for it) when control reaches its declaration
- It gets destroyed when the variable "goes out of scope"
- But a variable stays alive when it is shadowed

```
void f(int x) {// in each call of f, x is created and initialised
  x = g(x);     // until g returns x is not visible, but stays alive
  { int x = 1; /* parameter x alive, but not visible */ }
  for(int c=0 /* c "created" here */; c<12; ) {  // parameter x visible
    int x = 2;   // x "created" in each iteration. parameter x shadowed.
    x++; c += x;
  }  // at the end of each iteration x "destroyed", but c is alive
  // on exiting the loop, c is "destroyed"
  return x; // parameter x's value to be returned. x is destroyed.
}
```

Lifetime of Variables

Demo

- When a variable is created for a basic data type, or a struct containing basic data type members, memory is allocated, but may contain arbitrary values
- But for more complex data types, usually there is an initialisation
  - E.g., string is initialised as an empty string
- We will see later how to define your own data types and specify what needs to be done when a variable of that type is created and when it is destroyed

Static Variables in Functions

- Global variables (possibly declared in a namespace) are useful as they stay alive throughout the program.
  - But they can be modified from many points in the program, making it hard to debug
- A local variable in a function can be declared to be static, so that it behaves like a global variable in terms of lifetime, but a local variable in terms of scope
  - Like a global variable, the lifetime of a static variable starts when it is first accessed, and lasts till the end of the program
  - However, the scope is limited to the function: can only be accessed from within the function

Static Variables in Functions

- Example:
- Here, p will be initialised on the first call to the function
- Even after the function returns, p remains alive
- In subsequent calls, the value of p at the end of the previous invocation is retained (initialisation skipped)

```
struct posn { double x, y, deg; };

posn move_track(double step, double turn) {
    left(turn); forward(step);

    static posn p = {0, 0, 0};

    p.deg += turn;
    p.x += step*cosine(p.deg);
    p.y += step*sine(p.deg);
    return p;
}
```