Advanced C++

January 22, 2015

Mike Spertus

mike_spertus@symantec.com

Declaring vs Defining

- Declare (how to use)
  - int f();
- Define (actual implementation)
  - int f() { return 7; }
- Put declarations in headers
- Put definitions in
  - Headers if inline
  - Headers if template
  - .cpp files otherwise

Constructors

- `new Student_info()` leaves `midterm`, `final` with nonsense values. (Use the original version. The one with the “pure virtual” method can’t be new’ed!)
- But not `homework`! We’ll understand that momentarily
- Fix as follows:

```
struct Student_info {
  Student_info() : midterm(0), final(0) {}
};
```

Constructor Signatures

```
struct A {
   A(int _i = 0) : i(_i) {};
// Alternate
//    A(int _i = 0) { i = _i; }
  int i;
};
```

Non-virtual base classes

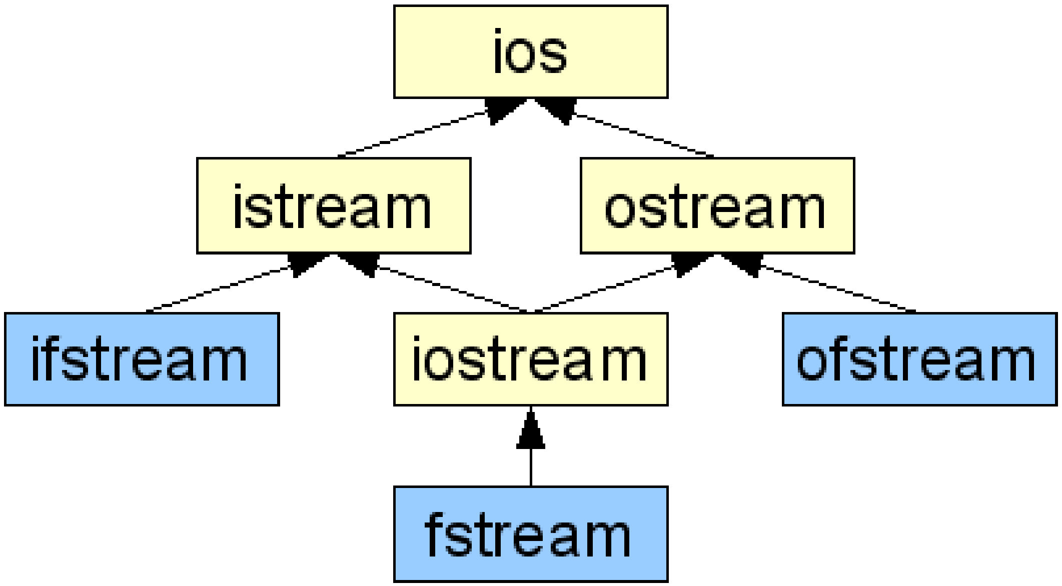
```
class B : public D {};
class C : public D {};
class A : public B, public C {
  // Has two D objects
};
```

Virtual base classes

```
struct B : virtual public D {
  B(int i) : D(i) {}
};
struct C : virtual public D {
  C() : D(5) {}
};
class A : public B, public C {
  // Has one D object
  A() : D(3), B(1){}
};
```

Multiple inheritance

- For a good discussion, see http://www.phpcompiler.org/doc/virtualinheritance.html



Implicit conversions

- Built-in
  - int i = 7;
    long l = i;
    char c = 7;
    char c = i; // No warning, but dangerous!
- Polymorphism
  - Animal *ap = new Dog;
  - Animal a = Dog(); // Legal but almost always wrong! Slicing
- User-defined
  - Constructors
  - Operator overloading
- “Standard Conversions”
  - Defined in clause 4 of the standard

Constructors and typecasts

```
struct A {
  A();
  A(int i);
  A(int i, string s);
  explicit A(double d);
};
A a;
A a0(1, "foo");
A aa = { 1, "foo"};
A a1(7); // Calls A(int)
a1 = 77;// ok
A a3(5.4); // Calls A(double)
a3 = 5.5; // Calls A(int)!!
```

Type conversion operators

```
struct seven {
  operator int() { return 7; }
};
struct A { A(int); }
int i = seven();
A a = 7;
A a = seven(); // Illegal, two user-
// defined conversions not allowed
```

Explicit conversions

- Old-style C casts (Legal but bad!)
  - `char *cp f(void *vp) { return (char *)vp; }`
- New template casting operators
  - static_cast<T>
    - Like C casts, but only makes conversions that are always valid. E.g, convert one integral type to another (truncation may still occur).
  - dynamic_cast<T*>
    - Casts between pointer types. Can even cast a Base* to a Derived* but only does the cast if the target object really is a Derived*.
    - Only works when the base class has a vtable (because the compiler adds a secret virtual function that keeps track of the real run-time type of the object).
    - If the object is not really a T *, dynamic_cast<T*> returns 0;
  - reinterpret_cast<T*>
    - Does a bitwise reinterpretation between any two pointer types, even for unrelated types. Never changes the raw address stored in the pointer. Also can convert between integral and pointer types.
  - const_cast<T>
    - Can change constness or volatileness only

Copy constructors

- Classes can have constructors that show how to make copies.
- Signature is `T(T const &)`
- A default copy constructor is almost always generated
  - Calls the copy constructors of all the base classes and members in the same order we discussed before
  - `T(T const &) = delete;`

Order of construction

- Virtual base classes first
  - Even if not immediate
- First base class constructors are run in the order they are declared
- Next, member constructors are run in the order of declaration
- This is defined, but very complicated
  - Best practice: Don’t rely on it
  - Good place for a reminder: Best practice: don’t use virtual functions in constructors

Constructor ordering

```
class A {
public:
  A(int i) : y(i++), x(i++) {}
  int x, y;
  int f() { return x*y*y; }
};
```

- What is `A(2).f()`?

Answer: 18! (x is initialized first, because it was declared first. Order in constructor initializer list doesn't matter)

Destructor ordering

- Reverse of constructor ordering
- Begin by calling total object destructor
- Then members in reverse order of declaration
- Then non-virtual base classes in reverse order
- Virtual base classes

Object duration

- Automatic storage duration
  - Local variables
  - Lifetime is the same as the lifetime of the function/method
- Static storage duration
  - Global and static variables
  - Lifetime is the lifetime of the program
- Dynamic storage duration
  - Lifetime is explicit
  - Created with “new” destroyed with “delete”
- In all cases, the constructor is called when the object is created and the destructor is called when the object isdestroyed

Static storage duration

- What orders are the constructors of static storage duration objects called?
- In each source file, they are constructed in order
- Static/global variables in different source files are constructed in undefined orders
- This creates interesting issues

Static storage duration

```
#include <iostream>
using namespace std;
struct A {
A() { cout << "Creating an A object" << endl;}
};
A static_a;

int main()
{ ... }
```

- Prints "Creating an A object" before main is run because all global and static objects need to be constructed before starting the main program.

Will cout be safe to use?

- There's something a little worrisome here. cout is a global object defined in the C++ run-time libary. The <iostream> header declares it as:

  ```
  extern ostream cout;
  ```

- How do we know `cout` will be initialized before `static_a`?
- Remember, order of static initialization is undefined for global objects defined in different source files

When does the global variable cout get constructed?

- If the standard library ignored the issue, it might or might not work, depending on whether `cout` or `static_a` is initialized first.
  - Unacceptable for static constructors not to be allowed to write to `cout`.
- Fortunately, there is a static method `ios_base::init()` that initializes the standard streams.

Can we force `cout` to be initialized before `static_a`?

- Sure, use a static constructor ourselves

```
#include <iostream>
using namespace std;
struct ForceInitialization {
  ForceInitialization() { ios_base::Init(); }
};
ForceInitialization forceInitialization;
struct A {
  A() { cout << "Creating an A object" << endl; }
};
A static_a;
...
```

Abstracting into a header

- We will need to include ForceInit in any file that might use cout during static initialization, so extract it into a header ForceInit.

```
#ifndef FORCE_INIT_H
#define FORCE_INIT_H
#include <iostream>
struct ForceInit {
 ForceInit() { ios_base::Init(); }
};
static ForceInit forceInit;
#endif
```

Static vs. Global

- In the file above, we needed to make forceInit static, so multiple files didn’t define the same global variable.
- However, the previous file still isn’t right because ios_base::Init() will be called once for each source file, and we only want to call it once.

Preventing multiple initialization

```
#include <iostream>
namespace cspp51044 {
struct ForceInit {
   ForceInit() {
      if(count == 0) {
         count = 1;
         ios_base::Init();
      }
   }
private:
   static int count;
};
 static ForceInit forceInit;
}
```

So useful, it’s already there

- This idiom is extremely useful, and is actually part of the iostream header, so as long as you include iostream above where you use cout, you’re (almost) OK

Tear down

- Automatic and static duration objects are destroyed at the end of their scope in the reverse order they were created:

```
struct A {
  A() { cout << "A() "; }
  ~A() { cout << "~A()"; }
};
struct B {
  B() { cout << "B()"; }
  ~B() { cout << "~B()"; }
};
void f() {
  A a;
  B b;
}
int main() { f(); return 0; }
// Prints A() B() ~B() ~A()
```

HW3.1

- Write a program that prints “Hello, world!” with the following main function:

```
int
main()
{
    return 0;
}
```

- Extra credit—Give a solution that depends on constructor ordering. The more intricate the dependence, the greater the extra credit.

HW 3.2

- An object of a class that implements operator() is called a functor. Functors are objects that can be used as if they were functions.
- For a simple (but useless) example of the syntax, you can look at http://www.devx.com/tips/Tip/13197 (free registration).
- More usefully, define a class `Nth_Power` so that code like the following prints cubes.

```
int main()
{
    vector<int> v = { 1, 2, 3, 4, 5 };
    Nth_Power cube(3);
    cout << cube(7) << endl; // prints 343
    // print first five cubes
    transform(v.begin(), v.end(),
              ostream_iterator<int>(cout, ", "), cube);
}
```

HW 3.3 – Extra credit

- Define classes D and B such that D inherits from B and create a B *b, such that `dynamic_cast<D*>(b)` and the c-style cast `(D*)b` give different results.
- You can demonstrate they give different results simply by printing them as pointers:

```
cout << dynamic_cast<D*>(b) << endl;
cout << (D*)b;
```

- Which one is better?
- If you wanted to get the C-style behavior but still don't want to use "bad" C++ casts, what C++ cast would you use?

HW3.4—Extra credit

- The const modifier can appear on either side of most types.
  - “int const” and “const int” mean exactly the same thing
- Which do you think is better?
- Hint: Think about pointer types

HW 3.5 – Practice with classes

- Your assignment is to implement the "Animal Game." The idea is that you chose a secret animal. The computer then asks you questions about the animal, terminating with a guess. If the guess is right, the computer wins, if it is wrong you win. But as part of winning, you have to provide your animal, and a differentiating yes/no question. (See, http://www.animalgame.com/ for an example. You will do a text version of course).

HW 3.6—Extra Credit

- Your goal in this problem is to call a method that doesn't exist!
- Recall that a pure virtual method is not given a body:
  struct A { … virtual void f() = 0; };
- Normally, in any class you instantiate the pure virtual method f is overridden, so you don't call a non-existent method
  - In fact, it's illegal to "new" a class with pure virtual methods
- Your task is to write a program that calls A::f() even though it has no body.
  - What happens when it tries to run A::f()? (Obviously, something bad)
- See the note on slide 26 for a hint