Lecture 04
Scoping, Constructors & Destructors
February 8th, 2016

Announcements
• PA2 is due on Wednesday
  – Questions?
• I have email about Quiz #3
  – Will look into them

What about Hidden Arguments?
• Every method has a hidden argument that is the object being operated on
• By default, hidden arguments are pass by reference
• What if a method doesn’t change the object it is called on?

Int Herd::Size() const;

Declares the hidden argument constant

Accessors vs Mutators
• A method with a constant hidden argument is called an accessor
  – Because it doesn’t change the object it accesses
• A method without a non-constant hidden argument is called a mutator
  – Because it can and presumably does change the object

New Term: Scope
• The scope of a variable refers to where it can be accessed.
• Local variables
  – enter scope when they are declared
  – Leave scope when the block they were declared in ends
    • At the }
• Arguments
  – Enter scope when the method/function is called
  – Leave scope when the method/function returns

Why Scope Matters
• Constructors are called when objects enter scope
• Destructors are called when they leave scope
• This is why you rarely need to invoke a destructor on a local variable
  – It happens automatically on return
• This is why } can trigger an error/crash
  – Error in a destructor
### Scoping & Parameter Passing

- **Pass by reference parameters**
  - Do not create new objects (only references)
  - No objects (just references) enter or leave scope
  - Do not trigger constructors
  - Do not trigger destructors
- **Pass by value & Pass by reference**
  - Different in terms of side effects
  - Different in terms of copying, which implies...
  - Different in terms of constructors/destructors
- **Pass by constant reference is still pass by reference**

### Example: How many destructors are called on `}`?

```cpp
using std::ifstream;
Complex Add(ifstream& istr)
{
  Complex c1, c2;
  c1.Read(istr);
  c2.Read(istr);
  c1.Add(c2);
  return c1;
}
```

### Special Case: Return Values

- **Return values:**
  - Are copied back to the calling function directly
  - Do not trigger destructors in the returning function
  - Do not trigger constructors in the calling function
- **All other local objects trigger destructors when the function returns.**

### Return Value Example

- **How many constructors are called, and where?**

```cpp
Quagga MakeQuagga()
{
  Quagga novel_quagga = Quagga(100);
  return novel_quagga;
}
```

```cpp
int main(int argc, char* argv[])
{
  std::cout << MakeQuagga() << endl;
}
```

### Review

- **Local variables are stored on the stack**
  - They might be primitives
  - They might be objects
- **Constructor are invoked...**
  - Where local variable are declared
  - Where parameters are passed by value
- **Destructors are invoked...**
  - When local variables fall out of scope
  - When non-reference parameters fall out of scope
- **References are not themselves objects**
  - Do not trigger constructors/destructors

### New Bug (not possible in Java)

```cpp
Complex& Foo::Bar()
{
  Complex c1;
  // code to compute the value of c1;
  return c1;
}
```
New Bug Fix

Complex Foo::Bar()
{
    Complex c1;
    // code to compute the value of c1;
    return c1;
}

Memory Management

• Java (unlike C++) keeps objects on the heap
  – Objects are created by ‘new’
  – Objects live indefinitely, until
    • No more references to them exist
    • And the garbage collector gets rid of them
  – Therefore, no destructors

• C++ has no garbage collector
  – More efficient
  – Permits real-time processing
  – The #1 source of errors in C++
    • Memory leaks
    • Invalid references/pointers

Memory Management Strategy #0

• Use local variables, avoid pointers
  – Constructed when they enter scope
  – Destructed when they leave scope
• No memory leaks are possible
  – Memory reclaimed when function returns
• Invalid references almost impossible
  – Unless you return an invalid reference

Limits to Strategy #0

• Limited scope
  – Local variables only live as long as the
    method/function that declares them
• Fixed size
  – The size of the stack frame must be known at
    compile time
  – Therefore, the size of local variables must be
    known at compile time

When data is persistent or dynamic, we need another strategy