Lecture 03b
Scope
September 7th, 2017

Announcements
- Reading/Quiz: Chapter 3 due Tuesday
- Programming Assignment #1
  - Your tile may or may not make a subdirectory
  - Other questions?
- Programming Assignment #2 is due Tuesday
  - Your tar file MUST make a subdirectory called PA2
    - Work in a directory called PA2
    - Clean it of .obj & executable files
    - From parent directory, type ‘tar –cvf PA2.tar PA2/*’
  - Other questions?

3 ways to pass arguments
- Pass by value: void Herd::Join(Quagga q);
  - Join is given copy of q
- Pass by ref: void Herd::Join(Quagga & q);
  - Join is given access to q in calling function
    - Assumption is that it modifies q
- Pass by const ref:
  void Herd::Join(const Quagga & q);
  - Join is given access to q in calling function
  - Join is prevented from modifying q

Argument Passing
- In complex code
  - Objects are passed as arguments more than primitives
  - Side effects of arguments are rare
    - Better avoided when possible
- Therefore
  Most arguments are passed by constant reference

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

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 }
- Parameters
  - Enter scope when the method/function is called
  - Leave scope when the method/function returns
    - Think in terms of the method’s outer { and }
**Why Scope Matters**

- Constructors are called when objects enter scope.
- Destructors are called when they leave scope.
- This is why you don't call destructors on local variables.
  - It happens automatically on return.
- This is why } can trigger an error/crash.
  - Error in a destructor.

**Scoping & Parameter Passing**

- References are not objects (they are primitives).
- Pass by value parameters:
  - Create new objects.
  - Trigger constructors.
  - On return, trigger destructors.
- Pass by reference parameters:
  - Do not create new objects.
  - No object enters or leaves scope.
  - Just references, which are primitives.
  - No constructors are triggered.
  - No destructors are triggered.
- Pass by constant reference is still "by reference".

**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.
  - This includes no copy constructors.
- All other local objects trigger destructors when the function returns.

**Return Value Example**

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

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

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

**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;
}
```

**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;
}

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

Memory Management

- Java (unlike C++) does its own memory management
  - Primitives live on the stack until function returns
  - 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
  - Programs allocate and deallocate their own memory
  - 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
  - Contructed 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

Dynamic Data

- Sometimes, the size of data is unknown at compile time
- Examples from PA2
  - We don’t know how long a string is until run-time
  - We don’t know how many words are in a document until run-time
- Dynamic data can’t be stored on the stack

When data is persistent or dynamic, we need another strategy

But wait, what about a vector of strings?
Virtual Memory (CS270)

- Every process gets its own virtual memory
- Half belongs to the OS
- In your process' half:
  - The stack starts at one end
  - The heap starts at the other
  - Hopefully, they never meet

The Heap

- The heap is memory under programmer control
  - Starts at the other end of VM
  - The 'new' operator:
    - Allocates memory on the heap
    - If an object, calls the class constructor to initialize it
    - Returns a pointer to the allocated object
  - The 'delete' operator:
    - Takes a pointer to an object as a parameter
    - If an object, calls the destructor
    - Dealslocate the memory
      - Available for use by 'new' again
  - Memory Management: every 'new' needs a matching 'delete'

Pointers

- A pointer is a typed VM address
  - As such, it is its own entity (datum)
  - It is not the same thing as the object it points to
  - It can point into the heap or the stack
    - It can point anywhere in virtual memory
    - It can point to uninitialized or unallocated memory
      - Almost always a bug (or malware)
    - It can point into the OS's half of memory
      - Many attempts to use such an address will flag an error (core dump)

Pointer Data Types

- Assume I define a class called Quagga
- Then...
  - Quagga is a data type
  - Quagga* is a data type
    - A pointer to a Quagga
  - Quagga** is a data type
    - A pointer to a pointer to a Quagga
  - Quagga*** is a data type
    - ... and so on
  - Quagga& is also a data type
    - But is not the same as Quagga*