Lecture 03a
Functions & Methods
September 5th, 2017

Announcements
• Reading:
  – Chapter 2 due today
  – Chapter 3 (& Quiz) for next Tuesday
• Programming Assignment #1 is due today
  – Output counts for valid input; otherwise, error
  – Errors graded by return value; error messages recommended
  – Any other questions?
• Programming Assignment #2 is due one week from today

Cascading Error Messages
• The g++ compiler can produce a lot of error messages
  – Pages worth (and the more command doesn't work)
• Two reasons
  – C++ allows so much to be overloaded/redefined
  – Compiler tries to continue past errors
• Your best approach:
  – Focus on the first error message
  – Fix it
  – Recompile

int main(int argc, char* argv[])
• int (return value)
  – 0 if code executes without error
  – -1 if an error occurs
• Argc : number of command line elements
  – Includes the name of your executable file
  – For PA1, argc should be 2
  • Executable file name
  • Input file name
• Argv : pointers to command line c-style strings
  – We will talk about pointers & c-style strings later
  – For PA1 & PA2, argv[1] is the input file name
  • You can pass it to the ifstream constructor (see example)

Good style for main functions
• What should your main function do?
  – Check the number of command line arguments
    • Return -1 if incorrect
  – Check each command line argument
    • Is it the right type?
    • If it's a filename, can you open it?
    • Return -1 if not
  – If there is an error
    • Tell user what the arguments should have been (Usage)
  – Create one or two high level objects
  – Invoke them
  – Return 0

Progress
• We are now done with Chapter 1 of Weiss
  – Did we cover everything in class?
  – No, that's what the text is for
  – We covered the biggest & newest topics
    • .cpp & .h files
    • Independent compilation
    • main
    • A little I/O
  – Weiss also covers data types, some syntax
    • But the differences from Java are small
    • So I leave them to your (slightly outdated) text
Vendors & Strings

• Weiss (2.2) introduced vectors
• What are vectors?
  – Primitives?
  – Classes?
  – Something else?
• They are templated classes
• We will discuss templates later
• Data types go in the angle brackets
  – vector<int> can contain only integers
  – vector<Point3D> can contain only Point3Ds

An Example Class

class Quagga {
public:
  Quagga(double s) : size(s) {}
  inline void Grow() {size *= 1.1;}
  double Size() {return size;}
protected:
  double size;
};

What does this print?

#include <iostream>
#include <Quagga.h>

int main(int argc, char* argv[]) {
  Quagga quip(100);
  Quagga quak(100);
  quak.Grow();
  std::cout << quip.Size() << " "
            << quak.Size() << std::endl;
  return 0;
};

But why does it print that?

• How does the Grow() method know which Quagga to grow?
• What does the syntax quak.Grow() mean?

• Every method has a hidden argument
  – An instance of its object class is passed as an argument
  – Unless the method is declared static
  – The hidden argument is used to access object fields

Functions vs Methods

• Functions are not parts of objects
  – Main is an example of a function
  – Note that there is no main object
  – Functions operate on arguments, produce return values
    • Technically they can access global data, but this is a bad idea
• Methods are functions with hidden arguments
  – The type of the hidden argument is the class the method belongs to
• The following are equivalent
  – void Quagga::Grow() {size *= 1;}
  – void Grow(Quagga& q) {q.size *= 1;}
• Java only has methods; C++ has methods & functions

CS270 refresher

• Every function/method has a stack frame
  – Except (sometimes) inline functions
• The stack frame holds:
  – Internal data (e.g. the function pointer)
  – Memory for local variables
• The stack frame is pushed onto the stack when a function is called
• The stack frame is popped off the stack when it returns
How are arguments passed?

• Two methods:
  – Call by value
    • The value in the calling function is copied to the stack frame of the called function
  – Call by reference
    • The value in the calling function is the value in the called function
    • More formally, the argument is a reference to a value in the calling function

How are arguments passed...

• In Java?
  – Primitives are passed by value
  – Objects are passed by reference

• In C++?
  – Data can be passed either way
  – If the argument is datatype name it’s pass by value
    • void Grow(Quagga q)
  – If the argument is datatype& name it’s pass by reference
    • void Grow(Quagga& q)
  – Hidden arguments are pass by reference
  – Otherwise, you choose

How do you choose?

• Do you want the method to change the value in the calling function?
  – If so, pass by reference
  – For example, the grow method
  – This is called a side effect
• Is the argument large?
  – If so, pass by reference
  – Avoid the cost of copying data
  – For example, if the data is a video or a database...
• Otherwise
  – Pass by value
  – Avoid unintentional interactions between calling/called methods

Rules of Thumb

• If a method does not change an argument:
  – Pass it by value
    • So readers know it doesn’t change
    • So the compiler knows it doesn’t change
  – Otherwise, pass by reference
    • Signaling the side effect
• But wait!
  – What about efficiency?
  – What if the argument is large?

Constant Reference

• Pass by constant reference
  void Herd::Join(const Quagga& q)

  Promise to readers and the compiler not to change the argument

3 ways to pass arguments

• Pass by value: void Herd::Join(Quagga q);
  – Join is given a copy of q
• Pass by ref: void Herd::Join(Quagga& q);
  – Join has access to the original q
  – Assumption is that it modifies q
• Pass by const ref:
  void Herd::Join(const Quagga& q);
  – Join has access to original q
  – 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;

Accessors vs Mutators

- A method with a constant hidden argument is called an accessor
  - Because it doesn't change the object is accesses
- A method without a non-constant hidden argument is called a mutator
  - Because it can and presumably does change the object

Examples

- Look at the Quagga::Grow method. Four options:
  - void Quagga::Grow(double scale)
    {weight *= scale;}
  - void Quagga::Grow(double& scale)
    {weight *= scale;}
  - void Quagga::Grow(const double& scale)
    {weight *= scale;}
  - void Quagga::Grow(const double& scale) const
    {weight *= scale;}