Announcements

- Midterm #1: Sept. 26th (week from Tuesday)
  - Code distributed one week from today
- PA2 test cases & answers posted
- Quiz #4 next Tuesday (before class)
- PA3 due next Tuesday
  - Any questions?

Comments on PA1 & PA2

- Lots of questions (Piazza/email) that were answered in class
- Lots of Tuesday night questions
- A few people not using objects
  - No one has ever passed the class this way
- Reminder: all code must be your own
  - Includes not using another student’s previous assignments

Review (again)

- 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

Arrays

- An array is
  - A contiguous block of memory
  - Containing instances of 1 type of data
- The data type of an array is a pointer
  - Quagga* and Quagga[] are the same data type
- The Brackets [] are used to offset into arrays
  - Arr[0] : 1st element
  - Arr[1] : 2nd element
  - Arr[n-1] : Nth element

Warning about Arrays

- Arrays are primitive data types (pointers)
  - They are not objects (class instances)
  - You cannot ask them their size
  - The bracket operator adds offsets
    - This is why indices start at 0!
  - The bracket operator does not check bounds
    - Imagine the following:

Quagga qarray[5];
Quagga quik = qarray[9];
C-Style Strings

• Before object oriented programming was invented, C programmers needed strings
• Strings in C are just arrays of char
  – Problem: how to keep track of their length?
  – Solution: arrays of char end in a NULL byte
    • Convention, but very strong convention
    • Only applies to arrays of char
    • C-string libraries assume NULL byte endings
    • Functions such as strlen and strcmp

Remember the main?

int main(int argc, char* argv[])

• The argv argument is an array of c-style strings
  – Can be written as char* argv[]
  – Can be written as char** argv

Argv in memory

> PA3 foo.pgm bar.pgm

Dynamic Data

• This is how we can allocate dynamic data.
• Image file headers begin by telling you the image width and height
  – Read the file header, discover the image is 512x512
  ```
  int* image = new int[width * height];
  ```
  – Now you have an array whose size was not known until run-time

What About Vectors?

• Vector is a header class
• Header classes are Memory Management Strategy #1
• A simple vector class might look like...
  ```
  class intvector {
    public:
    private:
      int* data;
      int size;
  }
  ```

Header Classes

• Header classes manage heap memory
• They allocate & de-allocate memory
• Properly implemented, they allow you to treat dynamic memory like a stack variable
• To do this, they rely on
  – Constructors
  – Destructors
  – The Assignment operator

* Your book calls these “The Big 3”
Example: An Int Vector

class intvector {
    public:
        intvector(int sz);  // Constructor allocates memory on heap
        ~intvector();       // Destructor deletes memory on heap
    int at(int index) const;  // Get value from a constant vector
    int& at(int index);      // Change values in a (non-constant) vector
    
    protected:
        int* data;
        int size;
};

Example (continued)

intvector::intvector(int sz) : size(sz) {
    data = new int[size];                  // Allocate memory when intvector comes into scope (is created)
}

int vector::~intvector() {
    delete[] data;                        // Deallocate memory when intvector drops out of scope (or is deleted)
    data = NULL;
    size = 0;
}

int intvector::at(int index) const
{
    if ((index >= 0) && (index < size)) return data[index];
    else throw std::exception();               // At methods check bounds (unlike [])
}

Header Classes

• The goal of a header class is to make dynamic data structures act like local variables
  – int vectors are local variables
    • They are objects on the stack
    • That contain a pointer to data on the heap
  – Like all local variables, their constructor is called when they come into scope
    • Under the hood, their constructor dynamically allocates heap memory
  – When they leave scope, their destructor is called
    • Under the hood, this deletes the heap memory
• For those of you using vectors and/or strings in PA1/2/3, this is what you were doing!

Improving intvector

class intvector {
    public:
        intvector(int sz);  // Constructor allocates memory on heap
        ~intvector();       // Destructor deletes memory on heap
    int at(int index) const;
    int& at(int index);  // Get value from a constant vector
    
    protected:
        int* data;
        int size;
};

Is this legal?

• Two methods with the same name and arguments?
  – The hidden argument isn’t the same
    • It is of type const intvector in one
    • It is of type intvector in the other
• Why declare both methods?
  – The 1st lets you get value from a constant vector
  – The 2nd lets you change values in a (non-constant) vector
Using Both Methods

```cpp
void Foo(intvector& a, const intvector& b) {
    int test;
    if (b.size() > 0) {
        test = b.at(0);
        if (a.size() > 0) {
            a.at(0) = test;
        }
    }
}
```

Make intvector Dynamic

```cpp
class intvector {
public:
    intvector(int sz);
    ~intvector();
    int at(int index) const;
    int& at(int index);
    void push_back(int value);
protected:
    int* data;
    int size;
};
```

Implementing `push_back`

```cpp
void intvector::push_back(int value) {
    int* temp = data;
    data = new int[size+1];
    for(int i=0; i < size; i++) {
        data[i] = temp[i];
    }
    data[size] = value;
    size++;
    delete [] temp; // discuss
}
```

Problem: why does this crash?

```cpp
void Foo(intvector a) {
    ...
}
void Quagga::Bar() {
    intvector iv(0);
    iv.push_back(37);
    Foo(iv);
}
```

Copy Constructors

- When an object is passed by value, its **copy constructor** is called.
- Copy constructors take a constant reference to the same type object as their arguments
  ```cpp
  Quagga::Quagga(const Quagga& src)
  ```
- The default copy constructor copies the values of the parameter to the new instance
  - Including pointers
- Fortunately, you can redefine the copy constructor

Fixing intvector

```cpp
class intvector {
public:
    intvector(int sz);
    intvector(const intvector& src);
    ~intvector();
    int at(int index) const;
    int& at(int index);
    void push_back(int value);
protected:
    int* data;
    int size;
};
```
Implementing the Copy Constructor

```cpp
intvector::intvector(const intvector& src) {
    size = src.size;
    data = new int[size];
    for(int i=0; i < size; i++) {
        data[i] = src.data[i];
    }
}
```

• Are these legal? Yes. Size and data are protected, which means only intvectors can access them.
• It would be legal if they were private, too.