Lecture 05
Arrays & C-Strings

February 15\textsuperscript{th}, 2016
Announcements

• PA2 grades are posted
  – Along with test files
  – Discussion of newline

• PA3 is due on Wednesday
  – Any questions?

• There is a reading assignment & quiz for Wednesday
  – Unusual, so don’t miss it!
  – Numbered Quiz 4a, covers Sections 4.1 – 4.5
  – Quiz 4b will be due Monday…

• Midterm is a week from Wednesday
  – Code handed out Monday
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] : 1\text{st} element
  - Arr[1] : 2\text{nd} element
  - Arr[n-1]: N\text{th} element

Array of 3 Quaggas

Quagga #1
Quagga #2
Quagga #3
Arrays (continued)

- Arrays can be allocated on the stack
  
  ```
  Quagga qarray[3];
  ```

- If their size is known at run-time.
- Otherwise, allocate them on the heap
  
  ```
  Quagga* qarray = new Quagga[n];
  ```

- Now \( n \) can be a variable
Dynamic Data

• This is how we can allocate dynamic data.
• In PA3, assume you have a Pose class
  – Read the file, discover there are 73 Poses
  
  ```
  int pose_count = 73;
  Pose* pose_array
      = new Pose[pose_count]();
  Pose_arrar[2] = Pose(...);
  ```

• Now you have an array whose size was not known at run-time
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];
Pointer Arithmetic

- Pointers are VM addresses
- + and – are defined for pointers
- The data type determines the unit size

```cpp
Quagga* qarray = new Quagga[5];
Quagga* qptr = qarray + 2;
std::cout << (*qptr == qarray[2]);
std::cout << (qptr == &qarray[2]);
std::cout << std::endl;
```

- What does this print?
- Note that this is how brackets are implemented:
  
  Qarray[2] == *(qarray + 2)
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 & strcmp
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

* Any history buffs in the audience?
Argv in memory

> PA3 foo.txt bar.txt
What About Vectors?

• Vector is a *header class*
• Header classes are Memory Management Strategy #1
• A simple vector class might look like…

```cpp
class intvector {
  public:

  private:
    int* data;
    int size;
}
```
Header Classes

• Header classes manage heap memory
• The 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);
    ~intvector();
    int at(int index) const;

protected:
    int* data;
    int size;
};
Example (continued)

```cpp
intvector::intvector(int sz) : size(sz)
{
    data = new int[size];
}

Intvector::~intvector()
{
    delete data;
    data = NULL;
    size = 0;
}

int intvector::at(int index) const
{
    if (index < size) return data[index];
    throw std::exception();
}
```
Using intvector

```cpp
#include<intvector.h>
#include<iostream>
using std::cin;

int main(int argc, char* argv[]) {
    int data_size;
    cin >> data_size; //todo: error check

    intvector ivec(data_size);
    return 0;
}
```

Constructor allocates memory on heap

Destructor deletes memory on heap