Lecture 04a
Pointers and Arrays
Sept. 12th, 2017

Announcements
• The schedule said Midterm #1 was one week from today -- but I am changing that
• Midterm #1 rescheduled for two weeks from today
  – September 26th
  – We need another week of material
  – PA3 due Sept. 19th
  – Quiz #4 due Sept. 19th

More Announcements
• My Thursday after-class office hours are changed to Tuesday after class.
• Quiz #3 Due before class today
• Quiz #4 next Tuesday (new)
• PA1 Questions still coming in
  – Suggests students didn’t check PA1 answer until late
  – So far, our script hasn’t made a mistake…
  – Yes, we expect you to conform to the assignment
    • 20 lines of output
    • Return -1 on errors
    • Return 0 on success
• PA2 due today
  – Any questions?
• PA3 due next Tuesday

Review
• Local variables are stored on the stack
  – They might be primitives
  – They might be objects
• Constructor arguments
  – Where
  – Where
• Destructor arguments
  – 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++) 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
  – 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
  – 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

Virtual Memory

- 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
    - For objects, 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
    - Calls the destructor (if an object)
    - Deallocates 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
      - But this is almost always a bug (or malware)
      - It can point into the OS’s half of memory
        - But most 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*

Operations on pointers

- ‘new’ allocates a pointer (from the heap)
- ‘delete’ de-allocates a pointer
  - If the pointer does not point to the heap, your code will crash!
  - Delete does not change the value of the pointer
    - Setting it to NULL after delete is good style
    - Deleting a NULL pointer is a no-op
  - If quik is an instance of Quagga:
    - &quik returns a pointer to quik
    - If qptr is a pointer to a Quagga:
      - *qptr returns the quagga pointed to by qptr
      - qptr->foo() call the foo method of *qptr
      - qptr->bar returns the bar field of *qptr
      - qptr->bar is same as *(qptr).bar
Objects vs Variables

- Imagine the following line of code:
  ```java
  Quagga* qptr = new Quagga(10);
  ```

- Two simple questions:
  - What is the data type of `qptr`?
    - `Quagga*` (pointer to `Quagga`)
  - What is the variable name of the `Quagga` that was allocated?
    - It doesn’t have one!
    - The `Quagga` is an object, but not a variable
    - `qptr` is a variable of type `Quagga*`

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

Arrays (continued)

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

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

- Now `n` can be a variable

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:
    ```java
    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

  ```java
  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:
  ```java
  Qarray[2] == *(qarray + 2)
  ```