Lecture 15a
Persistent Memory &
Shared Pointers
May 4th, 2016

Announcements

• PA10 is due Wednesday
  – questions?
• Recitation: extra help (optional/ungraded)
• Final: same style as midterms
  – Cumulative, but with emphasis on templates
  – Code will be handed out on Friday
• ACM club tonight @ 6
  – Super Smash Bros. game night with pizza
  – Officer elections

By Popular Demand

• Plotting times shows only outliers

Speed Points (half total)

• Red line shows approximate cut-off for speed points

Times < 100 ms

• Roughly linear distribution from 23 – 72 ms

Coming Full Circle ...
Memory Management

• Our first major topic was memory management
• We outlined two strategies:
  – Strategy 0: put everything on the stack
    • Never leaks or double-deletes memory
    • Doesn’t work for dynamic data
    • Doesn’t work for persistent data
  – Strategy 1: header classes
    • Allows dynamic data to act like its on the stack
    • Notice that STL exploits this for all containers
    • Doesn’t work for persistent data
• So what do we do about persistent data?
Memory Management Strategy #2

- Unique Pointer strategy:
  - Never have more than one pointer to heap objects
- Essence of the strategy:
  - Never more than one pointer to a heap object
  - Whenever it’s deleted, NULL it out
  - These two points guarantee no double deletions
  - These guarantees no memory leaks
- Enforcing the strategy:
  - Wrap the pointer in a template object
  - The object’s destructor deletes the data if it falls out of scope
  - “Deep” destructor
  - The object’s copy constructor (and std::move) only permit new copies when the old copy is deleted
  - Semantics is “move”, not “copy”
  - Assignment operator overloaded, too (deletes old value)

Limitations of Unique Pointers

- Unique pointers are appropriate when data is
  - Persistent
  - Can be dynamic, too
- But what if its persistent ... and shared?
  - Imagine the following surveillance system:
    - Tracking module needs data as long as object is in view
    - Identification module needs data until verified
    - Graphics module needs to display object as long as tracking and/or identification is working on it

Memory Management Strategy #3: Shared Pointers

- std::shared_ptr<TYPE> is a template
  - Similar to std::unique_ptr<TYPE>
  - But this time, the semantics are that a pointer is shared among multiple users
- std::shared_ptr acts like a pointer
  - Overloads * and ->
  - Overloads copy constructor
    - But copy constructor is public & available
  - Overloads destructor, assignment

Semantics: reference counting

- Shared pointers implement a type-specific reference counting scheme
  - A mini-garbage collector
  - Used only for data stored in shared pointers
  - More overhead than strategies 0 – 2
  - But only used when you absolutely need it...

Shared pointer pseudo-implementation

- std::shared_pointer<TYPE> is a template
  - So std::shared_pointer<Video> defines a type
  - This type has a hash table static variable
    - Key is pointer address
    - Value is reference count
  - Supports type-specific garbage collection
    - Reference counting
    - Not stop-and-copy

Psuedo-implementation (cont.)

- std::shared_ptr<TYPE>::(TYPE* ptr) constructor
  - Adds address to hash table
  - With reference count 1
- Copy constructor increments reference count
- Destructor
  - Decrements the reference count
  - Deletes data if reference_count is now 0
- Assignment operator
  - Decrements old reference count (maybe deletes)
  - Increments new reference count
Example

1. `#include <memory>`
2. `using namespace std;`
3. `shared_ptr<Video> AllocateVideo(int n)`
4. `{`  
5. `shared_ptr<Video> video_ptr(new Video);`  
6. `for (int i = 0; i < n; i++) { video_ptr->push_back(Pose()); }`  
7. `return video_ptr;`  
8. `}`

Example (cont.)

1. `int main(int argc, char* argv[])`  
2. `{`  
3. `shared_ptr<Video> video_ptr = AllocateVideo(20);`  
4. `cout << "Address of Video = " << &(*video_ptr) << endl;`  
5. `return 0;`  
6. `}`

Questions on Example

- What do lines 5/6 of main print?
  - Number of pointers = 1
- What does UseVideo print from line 14?
  - Number of pointers = 2
- What do lines 11/12 of main print?
  - Number of pointers = 1
- Do lines 4 & 10 of main print the same thing?
  - Yes
- What does line 14 in UseVideo print?
  - The same thing

More Questions

- Does the address ever change?
  - No, its not that kind of garbage collector
  - Reference counting does not copy
- How do I handle my graphics example?
  - Where I need to display the object until the other modules are done with it.....
    - Hmm....

Weak Pointers

- Weak pointers are constructed from shared pointers
  - `std::weak_ptr<Type>(shared_ptr<Type> ptr)`
  - Part of memory management strategy #3
- Weak pointers do not change the reference count
  - Their constructors do not increment it
  - Their destructors do not decrement it
- When referencing a weak pointer
  - `*, ->` and bool check the reference count
    - Value is NULL if reference count is 0
    - Value is address otherwise

More Advanced Use

- Imagine a video contains many poses
  - You have a shared pointer to a video
    - But you want to pass a pointer to one of its poses
- Every shared (and weak) pointer contains 2 addresses
  - One is to the block of memory
    - In this case, the video
  - The other is to an address in that block
    - In this case, the pose
- The constructor is:
  - `std::shared_ptr<T>(T* block, T* addr);`
- `*` and `->` use the address, not the block