Lecture 15a
Persistent Memory & Shared Pointers

Dec. 5th, 2017
Jack Applin, Guest Lecturer
Announcements

• PA9 is due today
• Recitation : extra help (optional / ungraded)
• Final : same style as midterms
  – Cumulative, but with emphasis on templates
  – Code will be handed out on Thursday
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
    • Still doesn’t work for persistent data

• So what do we do about persistent data?
Strategy #2 : Unique Pointers

- **Unique Pointer strategy:**
  - Never have more than one pointer to heap objects

- **Essence of the strategy:**
  - Disallow copying
    - "move" allows copy, if source is nulled out
    - Same with assignment (=)
  - When a pointer falls out of scope, delete it
  - Null out any unique pointer that is deleted
    - To avoid double deletes

- **Enforcing the strategy**
  - Copy constructor is private (compile time error)
  - Destructor deletes the data if it falls out of scope
  - `std::move`
    - Makes a copy
    - Nulls out source
  - Assignment operator deletes destination, then nulls out source
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
    • Copy constructor does not null out source
  – Overloads destructor, assignment
Semantics: reference counting

• Shared pointers implement a type-specific reference counting scheme
  – Used only for data stored in shared pointers
  – More overhead than strategies 0 – 2
  – Only use it when you absolutely need it…
Shared pointer pseudo-implementation

- `std::shared_pointer<TYPE>` is a template
  - So `std::shared_pointer<Document>` 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 std::shared_ptr;
3. shared_ptr<Document> AllocateDocument(char* filename)
4. {
5.    shared_ptr<Document> doc_ptr(new Document(filename));
6.    return doc_ptr;
7. }
8. void UseDocument(shared_ptr<Document>& doc_ptr)
9. {
10.   cout << "Number_of_pointers = " << doc_ptr.use_count() << endl;
11.   cout << "Address of Document in UseDocument is = " << &(*doc_ptr) << endl;
12. }
13. void DeAllocateDocument(shared_ptr<Document> doc_ptr)
14. {
15.   cout << "Address of Document in DeAllocate is = " << &(*doc_ptr) << endl;
16. }
Example (cont.)

1. int main(int argc, char* argv[])
2. {
3.     shared_ptr<Document> doc_ptr = AllocateDocument(20);
4.     cout << "Address of Document = " << &(*doc_ptr) << endl;
5.     cout << "Number_of_pointers = "
6.         << doc_ptr.use_count() << endl;
7.     UseDocument(doc_ptr);
8. 
9.     DeAllocateDocument(doc_ptr);
10.   cout << "Address of Document = " << &(*doc_ptr) << endl;
11.   cout << "Number_of_document_ptrs = "
12.      << doc_ptr.use_count() << endl;
13.   return 0;
14. }

2017-12-04
CS253 Fall 2017 © Jack Applin & Bruce Draper
Questions on Example

- What do lines 5/6 of main print?
  - Number of pointers = 1
- What does UseDocument print from line 10?
  - Number of pointers = 1
- 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 11 in UseDocument print?
  - The same thing
More Questions

• Does the address ever change?
  – No, it’s not that kind of garbage collector
  – Reference counting does not copy

• How do I handle my motivating example?
  – Where I need to display an 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
  – You can’t. * and -> don’t work.
  – Construct a shared_ptr<> from it; use that.
Weak Pointers (II)

• Impact:
  – The data might get deleted out from under them.
  – Your code needs to check them every time they are used.
  – Check via .expired().