Lecture 14b
Persistent Memory & Unique Pointers

Nov. 30th, 2017
Jack Applin, Guest Lecturer
Announcements

• PA9 is due Tuesday
  – Same as PA8, but graded on speed
• Recitation next week : extra help
  – optional / ungraded
• Final : same style as midterms
  – Cumulative, but with emphasis on templates
  – Code will be handed out one week from today
STL Algorithms (review)

• Goals:
  – Provide useful basic algorithms (think ‘sort’)
  – Without regard to the data type operated on
  – Without regard to the data structure it is stored in
  – Over a whole container or a fragment thereof

• Example

  Template<typename ITER, typename VALUE>
  VALUE accumulate(ITER start, ITER end, 
                    VALUE init_value)
  {
    for(ITER iter=start; iter!=end; iter++)
      init_value += *iter;
    return init_value;
  }
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 it’s 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?
Persistent Data

• Persistent Data:
  – Created in one method/function
  – Needed later by others
  – Lifetime of data may be data-dependent

• Old (and bad) solutions
  – Global data
    • Destroys modularity / encapsulation
    • Complicates debugging / testing
    • Memory leaks & double deletes are common
  – Keep data local to main
    • Pass it to functions/methods
    • Not really that different from global variables
Smart Pointers

• Smart pointers are pointer abstractions designed to support memory management
  – Not surprisingly, they are templates
  – Require the \texttt{--std=c++14} compiler flag
  – Warning: \texttt{auto\_ptr} is deprecated and should not be used

• Three good types of smart pointers
  – Unique pointers
  – Shared pointers
  – Weak pointers
Unique Pointer motivation

• You are working on a large project
  – 100,000+ lines of code
  – Written by many people
    • Some have left the project
    • Others are new to the project
• The software runs continuously & in real time
  – E.g. operating system, surveillance system, aircraft control system
• Persistent data of variable duration
  – Network connections (OS)
  – Aircraft trajectories
  – Moving objects (video surveillance)
• How do you guarantee no memory leaks?

You & your team need to be systematic
Memory Management
Strategy #2 : Unique Pointers

• Unique Pointer strategy:
  – Never have more than one pointer to a heap object

• 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
  – Delete the data any time the pointer leaves scope
    • This guarantees no memory leaks

• Enforcing the strategy
  – std::unique_ptr<> is a template that replaces the true pointer
  – The unique_ptr destructor deletes the data if it falls out of scope
    • “Deep” destructor
  – The copy constructor only creates new copies if the old copy is deleted
    • Semantics is “move”, not “copy”
Example (Part 1)

```cpp
#include<memory>
using std::unique_ptr;

unique_ptr<Document> AllocateDocument(char* filename) {
    unique_ptr<Document> doc_ptr(new Document(filename));
    return doc_ptr;
}

void PrintDocument(unique_ptr<Document>& doc_ptr) {
    doc_ptr->Print();
}

void DeAllocateDocument(unique_ptr<Document> doc_ptr) {
    cout << "Address of Document in DeAllocate is = " << &(*doc_ptr) << endl;
}
```
Example (Part 2)

```cpp
int main(int argc, char* argv[]) {
    unique_ptr<Document> doc_ptr = AllocatDocument(argv[1]);

    cout << "Address of Video = " << &(*doc_ptr) << endl;

    PrintVideo(video_ptr);

    DeAllocateVideo(std::move(video_ptr));

    return 0;
}
```
Properties of std::unique_ptr<>

• Initialized with a heap pointer
  – Stylistically, with no intervening named variable
  – unique_ptr<Document> doc_ptr(new Document("foo");

• Unique pointers can be returned
  – This doesn’t create a 2\textsuperscript{nd} copy
  – Doesn’t call copy constructor
  – As in AllocateDocument example

• They can be passed by reference
  – This doesn’t create a 2\textsuperscript{nd} copy
  – void PrintDocument(unique_ptr<Document>& ptr)
  – Passing std::unique<>* is also OK
Properties: ownership

• Only one method/object can “own” the unique pointer
  – Ownership enforced at compile-time

• To pass ownership, use the move() function
  – DeAllocateDocument(std::move(doc_ptr));
  – Move() nulls out the pointer in the calling method
  – Returns a value for which there is a unique_ptr copy constructor
    – void DeAllocateDocument(unique_ptr<Document> ptr)

• When a unique_ptr leaves scope, its destructor deletes the data
  – DeAllocateDocument deletes the video by doing nothing!
More unique_ptr properties

- Assignment deletes previous data
  - `unique_ptr<video> ptr1(new vector<Frame>);`
  - `ptr2 = ptr1;`
  - 2nd line deletes 1st video (what ptr2 used to point at)
  - Assignment nulls out ptr1 (because ptr2 now points there)

- Unique pointers can be used as booleans
  - True iff not equal to NULL

- * and -> overloaded
  - Get method of unique pointers returns the underlying pointer
    - Can thwart the whole enterprise if misused
    - Avoid using get(), except where needed for compatibility

- Swap swaps the contents of 2 unique pointers
  - Doesn’t create any 2nd copies
Unique_ptrs for arrays

• Special version of unique pointers for arrays
  – `std::unique_ptr<int[]>` `ptr(new int[size]);`

• Array version allows indexing
  – `ptr[i]` is unique pointer to ith element
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