Lecture 15a
Persistent Memory & Unique Pointers
May 2nd, 2016

Announcements
• PA10 is due Wednesday
  – Distinguish between
    • Poses within a video
    • Coordinates within a pose
  – Sort, swap & left operate on poses
  – Rotate operates on coordinates
  – Transform operates on both
  – Other questions?
• Recitation: extra help (optional / ungraded)
• Final: same style as midterms
  – Cumulative, but with emphasis on templates
  – Code will be handed out on Friday

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?

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 –std=c++14 compiler flag
  – Warning: auto_ptr is deprecated and should not be used
• Three types
  – 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 connection (OS)
  – Aircraft record
  – Moving object
• How do you guarantee no memory leaks?

You & your team need to be systematic
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 its 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 only permits new copies when the old copy is deleted
  - Semantic is "move", not "copy"

Example (Part 1)
```cpp
#include <memory>
using namespace std;

unique_ptr<Video> AllocateVideo(int n)
{
  unique_ptr<Video> video_ptr(new Video());
  for(int i = 0; i < n; i++)
  {
    video_ptr->addPose();
  }
  return video_ptr;
}

void UseVideo(unique_ptr<Video>& vptr)
{
  cout << "Size of video = " << vptr->size() << endl;
}

void DeAllocateVideo(unique_ptr<Video> vptr)
{
  cout << "Address of Video in DeAllocate is = " << &(*vptr) << endl;
}
```

Properties of std::unique_ptr<>

- **Initialized with a heap pointer**
  - Stylistically, with no intervening named variable
  - `unique_ptr<Video> video_ptr;`
- **Passing ownership**
  - Use `std::move()` function
  - `std::move(video_ptr);`
- **They can be passed by reference**
  - `void DeAllocateVideo(unique_ptr<Video> vptr)`
- **They can be used as booleans**
  - True iff not equal to NULL
- **Assignment deletes previous data**
  - `unique_ptr<Video> ptr1(new Video());`
  - `ptr1 = unique_ptr<Video>(new Video());`
- **Unique pointers can be used as booleans**
  - True if not equal to NULL
- **Move overloaded**
  - Get method of unique pointers returns the underlying pointer
  - Can thwart the whole enterprise if misused
- **Swap swaps the contents of 2 unique pointers**
  - Doesn’t create any 2nd copies

Example (Part 2)
```cpp
int main(int argc, char* argv[])
{
  unique_ptr<Video> video_ptr = AllocateVideo(20);
  cout << "Address of Video = " << &video_ptr << endl;
  UseVideo(video_ptr);
  DeAllocateVideo(std::move(video_ptr));
  return 0;
}
```

Properties : ownership

- **Only one method/object can "own" the unique pointer**
  - Ownership enforced at compile-time
- **To pass ownership, use the move() function**
  - `DeAllocateVideo(std::move(video_ptr));`
  - Move() nulls out the pointer in the calling method
  - Returns a value for which there is a unique_ptr copy constructor
  - `void DeAllocateVideo(unique_ptr<Video> vptr)`
- **When a unique_ptr leaves scope, its destructor deletes the data**
  - `DeAllocateVideo` deletes the video by doing nothing!

More unique_ptr properties

- **Assignment deletes previous data**
  - `unique_ptr<Video> ptr1(new Video());`
  - `ptr1 = unique_ptr<Video>(new Video());`
  - 2nd line deletes 1st video
Unique_ptrs for arrays

- Special version are unique pointers for arrays
  - std::unique_ptr<int[]> ptr(new int[size]);
- Array version allows indexing
  - ptr[i] is unique pointer to ith element