Lecture 9: Standards

- The C++ language is big and growing!
- A standards committee decides every few years what additions/changes/deprecation should happen in the language

https://isocpp.org/std/the-committee

- C++ standardized in 1998
- C++03 – Mainly bug fixes
- C++11 – Introduced nullptr, auto, smart pointers
- C++14, C++17 – Lots of changes

https://en.wikipedia.org/wiki/C%2B%2B14

Memory management

- We’ve already been using some features from C++11
- We’re going to focus more on memory management today, and specifically the use of smart pointers from C++11
- Smart pointers can make memory management easier
- They keep track of how many other pointers point at their object
- And can automatically delete themselves when there no pointers left that reference their object
- No memory leaks!

Memory management

- Before we jump onto using smart pointers, we need to discuss memory management in more detail
- When you “create” a variable there are two places it could be stored
- The stack and the heap
- We’ve already been using both
- All memory allocated on the stack is known at compile time
- The heap stores things that aren’t

The stack

- The stack is just like a stack of plates
- A “stack frame” is created every time you enter a function
- It stores the values of variables passed in/out and local variables
- The stack is of a fixed size (set by the OS), on Windows the default stack size is 1MB, on Linux 8MB.
- This limits how many nested functions you could call, or the size/number of local variables you can store
- “Stack overflow error” is when the stack doesn’t have any more space left

Scoping

- We’ve seen previously a discussion on “scope” of variables
- This is how long they “live”
- It’s easy to see why variables don’t live beyond their function when you think about stack frames
- Once your program exists main(), the OS then handles deletion of all the memory your program used, even if you leaked memory

Time to write some code!

- Create “test_main_scoping.cpp”

Time to write some code!

- Create “test_main_stack.cpp”
- We’re going to build a picture of the stack with a trivial example and think about the heap

Smart pointers

- We know about the problem of not calling delete on pointers that have been dynamically allocated with new
- Memory leaks!
- If your program runs for a long time or does a lot of things, this can be a big problem
- A few types of smart pointers
- unique_ptr<>
- shared_ptr<>
- weak_ptr<>
- They are not a replacement for “raw” pointers
- You cannot do pointer arithmetic on them!

Time to write some code!

- Create main_test_unique_ptr.cpp

unique_ptr<>

- unique_ptr<> can point at one thing only and deletes that object when it is deleted (e.g., if it goes out of scope)
- Makes sure there is only one copy of an object exists
- Does not support copying

Time to write some code!

- Create main_test_unique_ptr.cpp

shared_ptr<>

- shared_ptr<> do “reference counting”. If you create a new shared_ptr<> pointing at the same object, it increments a counter
- When that counter equals 0, the destructor is called and the object destroyed
- The standard provides a macro to make allocating shared_ptr more efficient/convenient
- make_shared<T>()
- This allocates memory for both our type and our reference counter close to each other in memory

Time to write some code!

- Create main_test_shared_ptr.cpp

weak_ptr<>

- weak_ptr<> is like a shared_ptr<>, but without ownership
- So you can give someone a weak_ptr<>, and regardless of how many you give out, it doesn't change your "strong" reference counter
- So you still control ownership of your memory
- Think about our Matrix class
- If we defined a "getter" method that returned a shared_ptr<> to our values array, when we tried to delete values other people may have a shared_ptr<> to it
- So the reference may not be zero and it might not be deleted when we want

weak_ptr<>

- You have no idea if the thing a weak_ptr<> is pointing to is going to be destroyed half way through you working on it though
- Can call expired() to test if the reference count is zero
- But again you have no guarantee the weak_ptr<> stays valid after you've called expired()
- Calling lock() returns a shared_ptr<> if the reference count is non-zero, a nullptr if nots
- Normally if you want to work with a weak ptr, you normally "promote" it to a shared_ptr<>, then make sure you delete a shared_ptr when you are done

Time to write some code!

- Create main_test_weak_ptr.cpp

Homework

- Modify the Matrix.cpp and Matrix.h class to use smart pointers. What type of smart pointer do you think you might want to use?
- Write a “getter” that returns a weak_ptr<> to our values array
- ADVANCED – Write a matrix inversion routine that takes in a **diagonal** sparse matrix and returns its inverse
- SUPER ADVANCED – Construct a Jacobi iterative method to solve an SPD linear system using this sparse inverse