Smart Pointers

Peter Chapin

CIS-3012, C++ Programming

Vermont State University

Raw Pointers

- Traditional C-style pointers are called *raw* pointers
  - They are nothing more than machine addresses
  - Essentially, they are integers but have a different type

```
// Dynamically allocate space for an integer, initialized to 42.
int *p = new int{42};

// …

// Reclaim the dynamically allocated memory
delete p;
```

The Problems with Raw Pointers

- Raw pointers are very error-prone to use
  - Dynamic memory could get deleted twice (double delete), causing UB*
  - Dynamic memory might never get deleted (memory leak), wasting space
  - Dynamic memory could be accessed after being deallocated (use-after-free), causing UB
- Many bugs in C programs are attributed to mishandling memory!

* Undefined Behavior

Garbage Collection?

- Many programming languages support *garbage collection*
- The runtime system periodically (or in some other way) invokes a *garbage collector* to reclaim the memory held by objects that are no longer accessible to the program.
- The JVM in Java does this, for example
  - Very common; most languages do garbage collection

The Problems with Garbage Collection

- In the old days, the garbage collector could stall the program for significant time while it executed
  - Not an issue with today's advanced garbage collectors
- Even today, there is runtime cost of garbage collection that can be hard to evaluate
  - This is an issue for real-time systems
  - … although real-time garbage collection systems do exist
- The garbage collector is a large body of code
  - … an issue for highly constrained systems

Manual Memory Management

- C (and C++) require the programmer to explicitly decide when allocated memory is released
  - … using `free( )` in C
  - … using **`delete`** (or **`delete`** [] for arrays) in C++
- Pros:
  - Simplified runtime system reduces code size
  - Execution time characteristics are more deterministic
- Cons:
  - Easy to get wrong!

Smart Pointers

- C++ 2011 (and beyond) has smart pointers to help address this
- A smart pointer is a container that holds a single raw pointer
- Uses RAI to ensure that the raw pointers are deallocated appropriately and without leakage
- Frees the programmer from worrying so much about this issue and improves program reliability

<u>You Still Have to Use Them Properly!</u>

Unique Pointers

- A *unique pointer* has **exclusive** access to a dynamically allocated object
  - *No other pointer of any kind points at the object!*

```
// All smart pointers require this header
#include <memory>

// Declare p as a unique_ptr that wraps around the raw pointer returned by new.
std::unique_ptr<int> p{ new int{ 42 } };

*p = 84;   // Overloaded operators make using the unique_ptr natural.

// No explicit deallocation needed.
// The destructor of unique_ptr takes care of that.
```

Library Helper

- Starting with C++ 2014, the preferred way to create a `unique_ptr` is with the helper function template `std::make_unique`

```
// Using ‘auto’ removes the need to type the (obvious) type of ‘p’
auto p1 = std::make_unique<int>( 42 );

// std::make_unique takes arguments that are passed to the constructor
auto p2 = std::make_unique<std::string>( ‘x’, 1024 );
    // Create a dynamically allocated string consisting of 1024 ‘x’ characters
```

- The ability of `std::make_unique` to take a variable number of parameters of various types is because of a C++ 2011 feature called variadic templates

No Copying

- *Unique pointers cannot be copied!*
  - Doing so would result in two pointers that point at the same object, completely negating the purpose of unique pointers!
- Isn’t that limiting?
  - Yes, it is. However, we haven’t met `std::shared_ptr` yet. ☺
- Unique pointers can, however, be *moved*
  - Transfers ownership to the destination of the move
  - The original owner no longer tries to delete the object; it is considered *empty*
  - A default constructed `std::unique_ptr` is also in an empty state

Examples

```
auto p1 = std::make_unique<int>( 42 );
std::unique_ptr<int> p2;   // Default constructor creates an empty unique pointer

p2 = p1;   // Compile error! Copying not supported.

// Transfer ownership to p2.
// The destructor of p1 will no longer delete the object
p2 = std::move( p1 );
```

Unique Pointers and Functions

- Unique pointers can be returned from functions
  - The return value is *moved*
- Unique pointers can be passed into functions
  - … using `std::move`
  - ... or by reference
- This means ownership of an object can be passed into a function and returned from a function in a (mostly) natural way

Traditional Binary Tree Nodes

```
template<typename T>
struct TreeNode {
    T data;
    TreeNode *left;
    TreeNode *right;
};
```

```
// Recursively crawl over the tree, deleting the nodes.
void destroy_tree( TreeNode *node );
```

Binary Tree Nodes with Unique Pointer

```
template<typename T>
struct TreeNode {
    T data;
    std::unique_ptr<TreeNode> left;
    std::unique_ptr<TreeNode> right;
};


        // Destructor of TreeNode destroys ‘left’ and ‘right’
        // That triggers the deletion of the child nodes, etc., recursively
        delete root;
```

Release

- Sometimes you need to get the raw pointer back out of the unique pointer. Use the `release` method

```
auto p = std::make_unique<int>( 42 );

// Do things with p

int *pi = p.release( );  // p no longer owns the object.
```

Shared Pointers

- Multiple shared pointers can point at the same object…
  - … but they track how many such pointers exist
  - … and delete the object only when the last shared pointer disappears
- This means that a `std::shared_ptr` *can* be copied

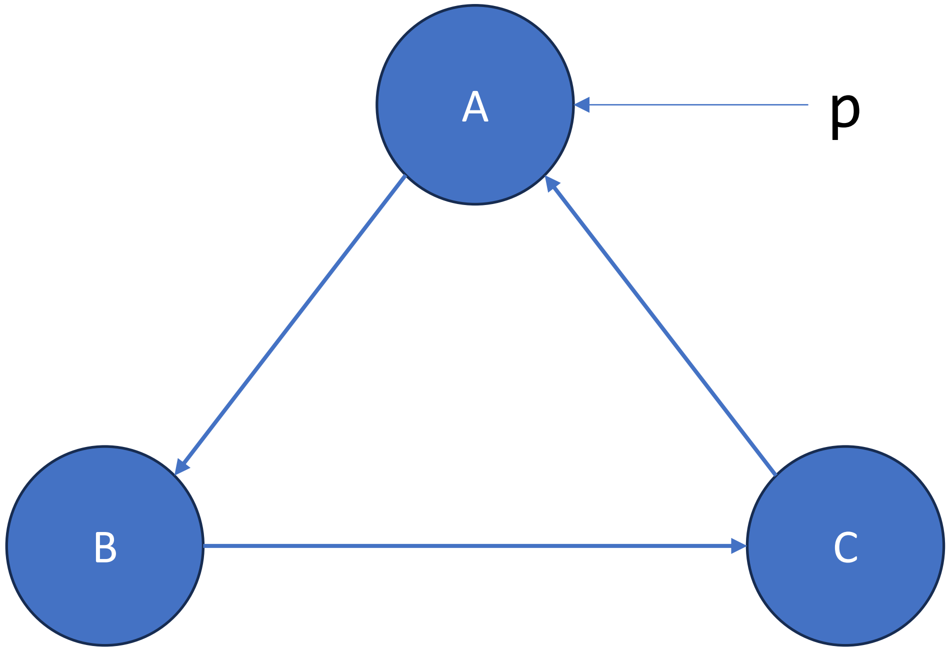
```
auto p1 = std::make_shared<int>( 42 );

// The pointers p1 and p2 point at the same object.
// The object is deleted only when both p1 and p2 are destroyed.
std::shared_ptr<int> p2{ p1 };

// Prints 2 because two shared pointers are involved.
std::cout << p2.use_count( ) << std::endl;
```

More Compelling Example

```
std::vector<std::shared_ptr<int>> pVec;
std::list<std::shared_ptr<int>> pList;

auto p = std::make_shared<int>( 42 );

// Add pointers to the same object to two different containers
pVec.push_back( p );
pList.push_back( p );

// The objects get deleted only when both containers are destroyed
```

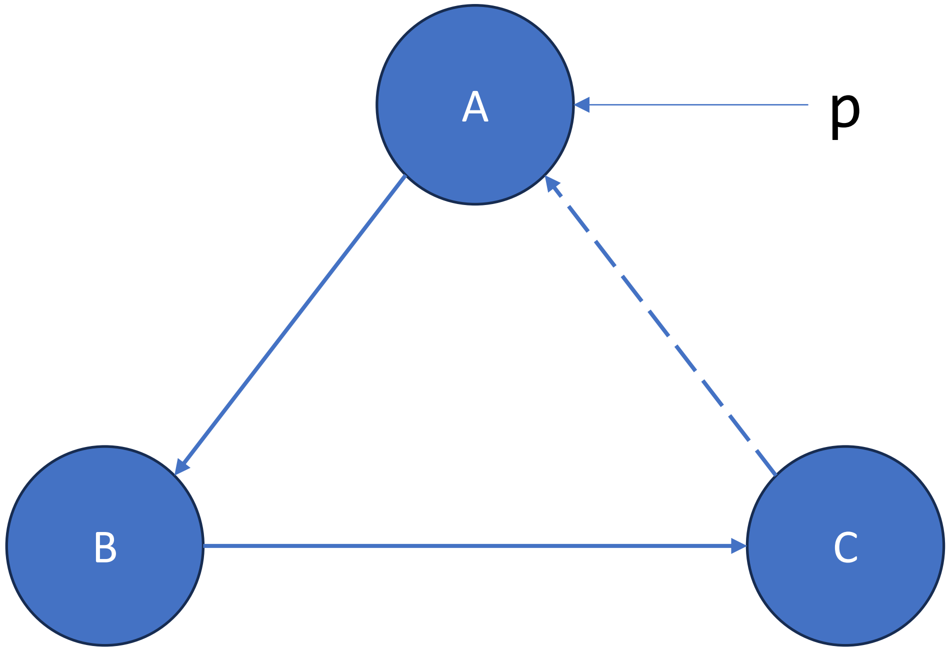
The Problem with Shared Pointers

- If each node contains a shared pointer to another node in a *cycle*…
  - … destroying the shared pointer p won't delete any nodes…
  - … because A still has another shared pointer that points at it
- <u>The nodes A, B, and C can leak!</u>



Weak Pointers

- Replace the pointer in C with a `std::weak_ptr`
  - Weak pointers don’t “own” the object to which they point and won’t delete it when they are destroyed
  - This avoids a double delete of A
- <u>Destroying p triggers removal of A, B, and C</u>



Weak Pointer Operations

- Weak pointers have very few operations
  - You cannot access the object to which they point (without first converting them into a `std::shared_ptr`)
  - This is surprising but makes sense… a `std::weak_ptr` might not actual be pointing at something (it might have been deleted). In such a case we say the `std::weak_ptr` has *expired*
- To convert a `std::weak_ptr` to a `std::shared_ptr`:
  - Use the `lock` method (returns a shared pointer, which will be empty if the weak pointer is expired)
  - Construct a shared pointer from the weak pointer (which throws an exception if the weak pointer is expired)

Creating Weak Pointers

- Shared pointers can be converted into weak pointers implicitly…
  - … by way of assignment to a weak pointer…
  - … or constructing a weak pointer from a shared pointer
- Shared pointers can be dereferenced like ordinary pointers (with the same operators), but weak pointers must be “locked” (i.e., converted to a shared pointer) before they can be used to access the target object