COMP6771 Week 5.1

Smart Pointers

Object lifetimes

To create safe object lifetimes in C++, we always attach the lifetime of one object to that of something else

- Named objects:
  - A variable in a function is tied to its scope
  - A data member is tied to the lifetime of the class instance
  - An element in a std::vector is tied to the lifetime of the vector
- Unnamed objects:
  - A heap object should be tied to the lifetime of whatever object created it
  - Examples of bad programming practice
    - An owning raw pointer is tied to nothing
    - A C-style array is tied to nothing
- **Strongly recommend** watching the first 44 minutes of Herb Sutter's cppcon talk "Leak freedom in C++... By Default"

Creating a safe* pointer type

Don't use the new / delete keyword in your own code

We are showing for demonstration purposes

```cpp
// myintpointer.h

class MyIntPointer {
 public:
  // This is the constructor
  MyIntPionter(int* value);

  // This is the destructor
  ~MyIntPointer() noexcept;

 int* GetValue();

 private:
  int* value_;
};
```

```cpp
// myintpointer.cpp
#include "myintpointer.h"

MyIntPointer::MyIntPointer(int* value): value_{value} {}

int* MyIntPointer::GetValue() {
  return value_
}

MyIntPointer::~MyIntPointer() noexcept {
  // Similar to C's free function.
  delete value_;
}
```

```cpp
void fn() {
  // Similar to C's malloc
  MyIntPointer p{new int{5}};
  // Copy the pointer;
  MyIntPointer q{p.GetValue()};
  // p and q are both now destructed.
  // What happens?
}
```

Smart Pointers

- Ways of wrapping unnamed (i.e. raw pointer) heap objects in named stack objects so that object lifetimes can be managed much easier
- Introduced in C++11
- Usually two ways of approaching problems:
  - unique_ptr + raw pointers ("observers")
  - shared_ptr + weak_ptr/observer_ptr

Type	**Shared ownership**	**Take ownership**
std::unique_ptr<T>	No	Yes
raw pointers	No	No
std::shared_ptr<T>	Yes	Yes
std::weak_ptr<T>	No	No

Unique pointer

- std::unique_pointer<T>
  - The unique pointer owns the object
  - When the unique pointer is destructed, the underlying object is too
- std::experimental::observer_ptr<T>
  - Unique Ptr may have many observers
  - This is an appropriate use of raw pointers (or references) in C++
  - Once the original pointer is destructed, you must ensure you don't access the raw pointers (no checks exist)
  - These observers **do not** have ownership of the pointer

Unique pointer: Usage

```cpp
#include <memory>
#include <iostream>

int main() {
  std::unique_ptr<int> up1{new int};
  std::unique_ptr<int> up2 = up1; // no copy constructor
  std::unique_ptr<int> up3;
  up3 = up2; // no copy assignment

  up3.reset(up1.release()); // OK
  std::unique_ptr<int> up4 = std::move(up3); // OK
  std::cout << up4.get() << "\n";
  std::cout << *up4 << "\n";
  std::cout << *up1 << "\n";
}
```

Can we remove "new" completely?

Observer Ptr: Usage

```cpp
#include <memory>
#include <experimental/memory>
#include <iostream>

int main() {
  int *i = new int;
  std::unique_ptr<int> up1{i};
  *up1 = 5;
  std::cout << *up1 << "\n";
  std::experimental::observer_ptr<int> op1{i};
  *op1 = 6;
  std::cout << *op1 << "\n";
  up1.reset();
  std::cout << *op1 << "\n";
}
```

Unique Ptr Operators

```
#include <memory>
#include <experimental/memory>
#include <iostream>

int main() {
  // 1 - Worst
  int *i = new int;
  std::unique_ptr<std::string> up1{i};

  // 2 - Not good
  std::unique_ptr<std::string> up2{new std::string{"Hello"}};

  // 3 - Good
  std::unique_ptr<std::string> up3 = make_unique<std::string>("Hello");

  std::cout << *up3 << "\n";
  std::cout << *(up3.get()) << "\n";
  std::cout << up3->size();
}
```

- https://stackoverflow.com/questions/37514509/advantages-of-using-stdmake-unique-over-new-operator
- https://stackoverflow.com/questions/20895648/difference-in-make-shared-and-normal-shared-ptr-in-c

Shared pointer

- std::shared_pointer<T>
- Several shared pointers share ownership of the object
  - A reference counted pointer
  - When a shared pointer is destructed, **if it is the only shared pointer left** pointing at the object, then the **object is destroyed**
  - May also have many observers
    - Just because the pointer has shared ownership doesn't mean the observers should get ownership too - don't mindlessly copy it
- std::weak_ptr<T>
  - Weak pointers are used with share pointers when:
    - You don't want to add to the reference count
    - You want to be able to check if the underlying data is still valid before using it.

Shared pointer: Usage

```cpp
#include <memory>
#include <iostream>

int main() {
  int* i = new int;
  *i = 5;
  std::shared_ptr<int> x{i};
  std::shared_ptr<int> y = x; // Both now own the memory
  std::cout << "use count: " << x.use_count() << "\n";
  std::cout << "value: " << *x << "\n";
  x.reset(); // Memory still exists, due to y.
  std::cout << "use count: " << y.use_count() << "\n";
  std::cout << "value: " << *y << "\n";
  y.reset(); // Deletes the memory, since
  // no one else owns the memory
  std::cout << "use count: " << x.use_count() << "\n";
  std::cout << "value: " << *y << "\n";
}
```

Can we remove "new" completely?

Weak Pointer: Usage

```cpp
#include <memory>
#include <iostream>

int main() {
  std::shared_ptr<int> x = std::make_shared<int>(1);
  std::weak_ptr<int> wp = x; // x owns the memory
  {
    std::shared_ptr<int> y = wp.lock(); // x and y own the memory
    if (y) {
      // Do something with y
      std::cout << "Attempt 1: " << *y << '\n';
    }
  } // y is destroyed. Memory is owned by x
  x.reset(); // Memory is deleted
  std::shared_ptr<int> z = wp.lock(); // Memory gone; get null ptr
  if (z) {
    // will not execute this
    std::cout << "Attempt 2: " << *z << '\n';
  }
}
```

When to use which type

- **Unique pointer** vs **shared pointer**
  - You almost always want a unique pointer over a shared pointer
  - Use a shared pointer if either:
    - You have several pointers, **and you don't know which one will stay around the longest**
    - You need temporary ownership (outside scope of this course)

When to use which type

- **Let's look at an example:**
  - //lectures/week5/reader.cpp

Shared or unique pointer?

- Computing examples
  - Linked list
  - Doubly linked list
  - Tree
  - DAG (mutable and non-mutable)
  - Graph (mutable and non-mutable)
  - Twitter feed with multiple sections (eg. my posts, popular posts)
- Real-world examples
  - The screen in this lecture theatre
  - The lights in this room
  - A hotel keycard
  - Lockers in a school

“Leak freedom in C++” poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don’t use owning raw *’s == don’t use explicit *delete*

Don’t create ownership cycles across modules by owning “upward” (violates layering)

Use *weak_ptr* to break cycles