COMP6771
Advanced C++ Programming

Week 5.2
Smart Pointers

Recap: RAII - Making unnamed objects safe

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
  MyIntPointer(int* value);

  // This is the destructor
  ~MyIntPointer();

 int* GetValue();

 private:
  int* value_;
};
```

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

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

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

MyIntPointer::~MyIntPointer() {
  // 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?
}
```

demo551-safepointer.cpp

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/raw pointers

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
- **raw pointer (observer)**
  - 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

Also note the use of 'nullptr' in C++ instead of NULL

Unique pointer: Usage

```
#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";
}
```

demo552-unique1.cpp

Observer Ptr: Usage

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

int main() {
  std::unique_ptr<int> up1(new int{0});
  *up1 = 5;
  std::cout << *up1 << "\n";
  int* op1 = up1.get();
  *op1 = 6;
  std::cout << *op1 << "\n";
  up1.reset();
  std::cout << *op1 << "\n";
}
```

demo553-observer.cpp

Can we remove "new" completely?

Unique Ptr Operators

This method avoids the need for "new". It has other benefits that we will explore.

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

int main() {
  // 1 - Worst - you can accidentally own the resource multiple
  // times, or easily forget to own it.
  int *i = new int;
  auto up1 = std::make_unique<std::string>(i);
  auto up11 = std::make_unique<std::string>(i);

  // 2 - Not good - requires actual thinking about whether there's a leak.
  std::unique_ptr<std::string> up2{new std::string{"Hello"}};

  // 3 - Good - no thinking required.
  std::unique_ptr<std::string> up3 = std::make_unique<std::string>("Hello");

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

demo554-unique2.cpp

- 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() {
  std::shared_ptr<int> x(new int{5});
  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";
}
```

demo555-shared.cpp

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';
  }
}
```

demo556-weak.cpp

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:
    - An object has multiple owners, **and you don't know which one will stay around the longest**
    - You need temporary ownership (outside scope of this course)
    - This is very rare

Smart pointer 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)

"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

Stack unwinding

- Stack unwinding is the process of exiting the stack frames until we find an exception handler for the function
- This calls any destructors on the way out
  - Any resources not managed by destructors won't get freed up
  - If an exception is thrown during stack unwinding, std::terminate is called

Not safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
  g();
  // Never executed.
  delete ptr;
}
```

Not safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
  g();
  auto uni = std::unique_ptr<int>(ptr);
}
```

Safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = std::make_unique<int>(5);
  g();
}
```

Exceptions & Destructors

- During stack unwinding, std::terminate() will be called if an exception leaves a destructor
- The resources may not be released properly if an exception leaves a destructor
- All exceptions that occur inside a destructor should be handled inside the destructor
- Destructors usually don't throw, and need to explicitly opt in to throwing
  - STL types don't do that

Partial construction

- What happens if an exception is thrown halfway through a constructor?
  - The C++ standard: "An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects"
  - A destructor is not called for an object that was partially constructed
  - Except for an exception thrown in a constructor that delegates (why?)

Spot the bug

```cpp
#include <exception>

class my_int {
public:
   my_int(int const i) : i_{i} {
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class unsafe_class {
public:
   unsafe_class(int a, int b)
   : a_{new my_int{a}}
   , b_{new my_int{b}}
   {}

  ~unsafe_class() {
    delete a_;
    delete b_;
  }
private:
   my_int* a_;
   my_int* b_;
};

int main() {
  auto a = unsafe_class(1, 2);
}
```

demo557-bad.cpp

Partial construction: Solution

- Option 1: Try / catch in the constructor
  - Very messy, but works (if you get it right...)
  - Doesn't work with initialiser lists (needs to be in the body)
- Option 2:
  - An object managing a resource should initialise the resource last
    - The resource is only initialised when the whole object is
    - Consequence: An object can only manage one resource
    - If you want to manage multiple resources, instead manage several wrappers , which each manage one resource

```cpp
#include <exception>
#include <memory>

class my_int {
public:
   my_int(int const i)
   : i_{i} {
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class safe_class {
public:
   safe_class(int a, int b)
   : a_(std::make_unique<my_int>(a))
   , b_(std::make_unique<my_int>(b))
   {}
private:
   std::unique_ptr<my_int> a_;
   std::unique_ptr<my_int> b_;
};

int main() {
  auto a = safe_class(1, 2);
}
```

demo558-partial1.cpp

Feedback