CS 3460

Introduction to Smart Pointers

Introduction

- Java has a GC to identify and return unused memory to the OS
- C++ raw pointers require the developer to free memory when no longer needed
- Smart pointers were added to the standard library in C++ 11
  - Provide a means for “automatic” memory management of dynamically allocated memory
  - There was an `std::auto_ptr` from C++ 98; removed in C++ 17

Overview

- Smart pointer is acquired
- Can be passed around (by-value), resulting in copies of the pointer being made
- When the last copy of the pointer goes out of scope, the memory is reclaimed, immediately
  - Developer doesn’t have to track this
  - It happens automatically in the destructor (of the pointer)
- This is not the same as the GC in Java.
  - GC is non-deterministic as far as an application is concerned
  - Smart pointers are deterministic; can trace a path for the full lifetime of a smart pointer

Two Types

- Header file `<memory>`
- Templated type; can point to any data type
- Shared Pointer
  - `std::shared_ptr`
  - Shared ownership by any number of shared pointers
- Unique Pointer
  - `std::unique_ptr`
  - Unique ownership, can not be copied, only *moved*

Introductory Example

```
std::shared_ptr<int> a = std::make_shared<int>(1);
std::shared_ptr<double> b = std::make_shared<double>(3.14159);

std::cout << "The value stored in a is " << *a << std::endl;
std::cout << "The value stored in b is " << *b << std::endl;
```

- Note how the pointer is typed, no use of * decorator
- Note how the allocation is performed
  - Type is specified in the <>
  - A value is set in the () - This is invoking a constructor
- Syntax for use of the pointer is the same as raw pointers
  - Note the same use of the dereference operator
  - Eventually we'll see the use of the -> operator

Introductory Example

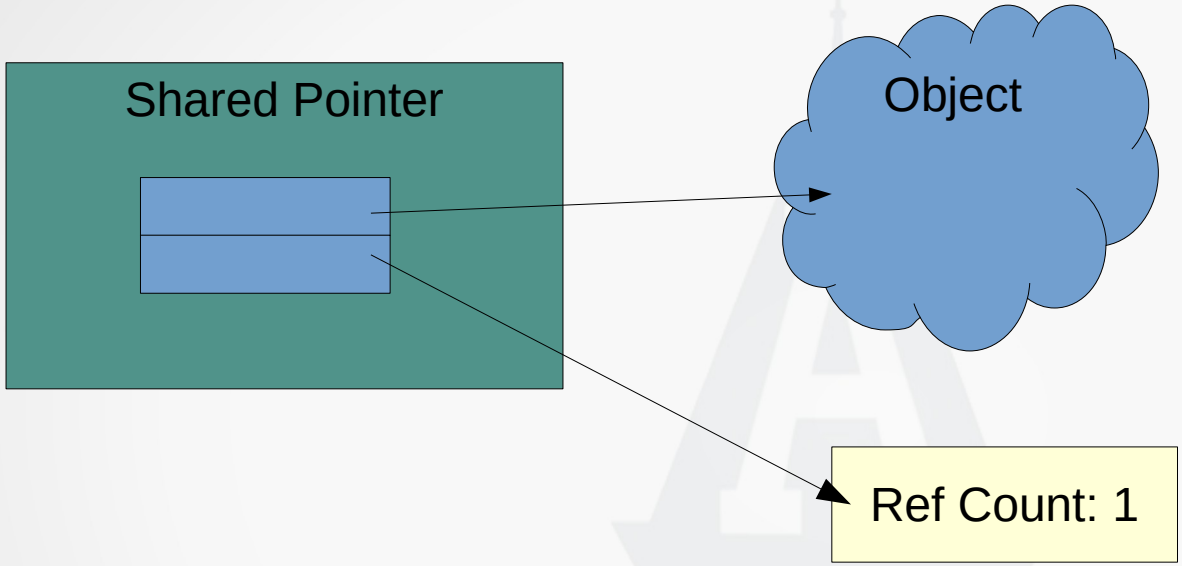
```
std::shared_ptr<int> a = std::make_shared<int>(1);
std::shared_ptr<double> b = std::make_shared<double>(3.14159);

std::cout << "The value stored in a is " << *a << std::endl;
std::cout << "The value stored in b is " << *b << std::endl;
```

- When the `a` and `b` variables go out of scope, the memory is automatically reclaimed

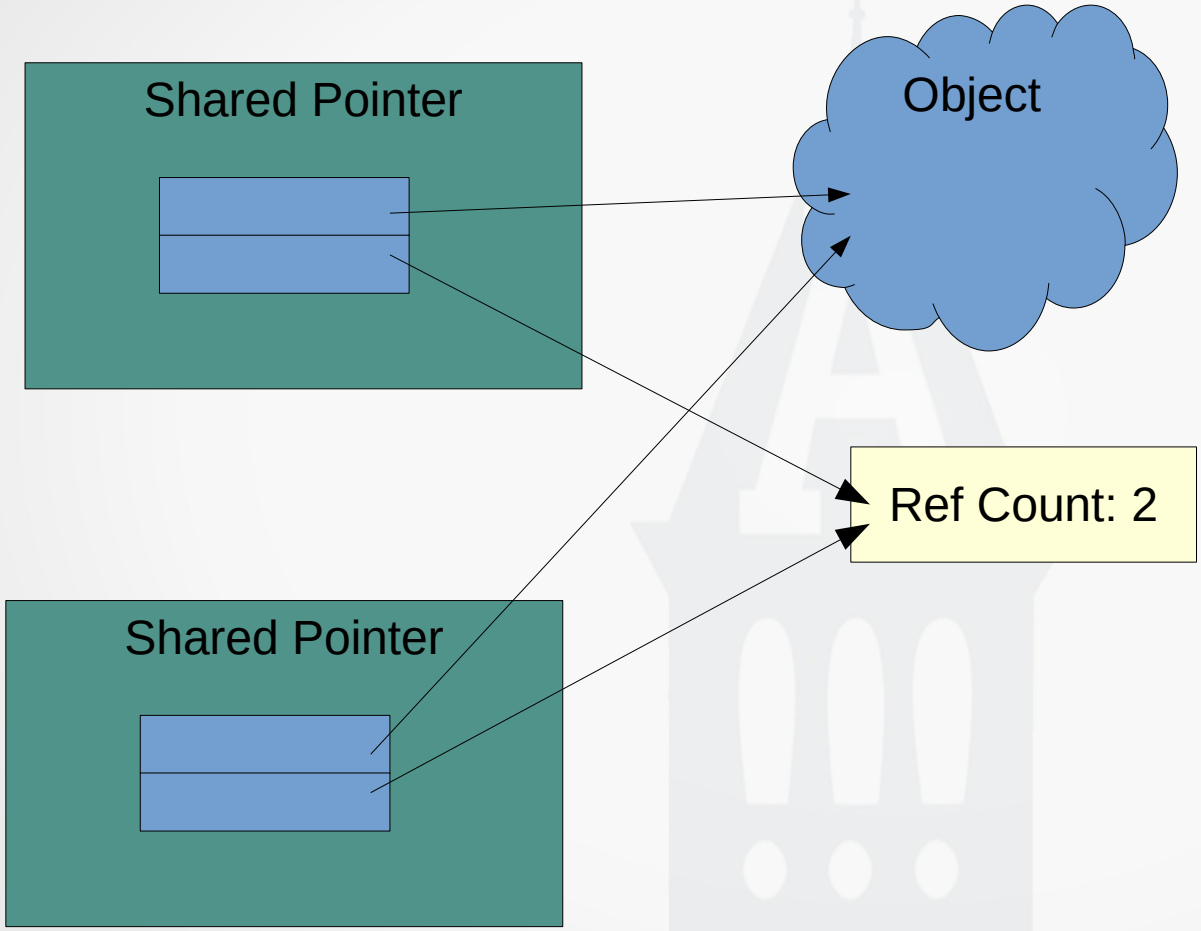
What is a Shared Pointer?

- It is an object; a C++ class
- Internal to the class
  - A raw pointer to the dynamically allocated memory
  - A raw pointer to a dynamically allocated reference count
    - Initially set to 1
- When a shared pointer is copied, ref count is incremented
- When a shared pointer goes out of scope
  - ref count is decremented
  - if ref count goes to 0, raw pointers are cleaned up

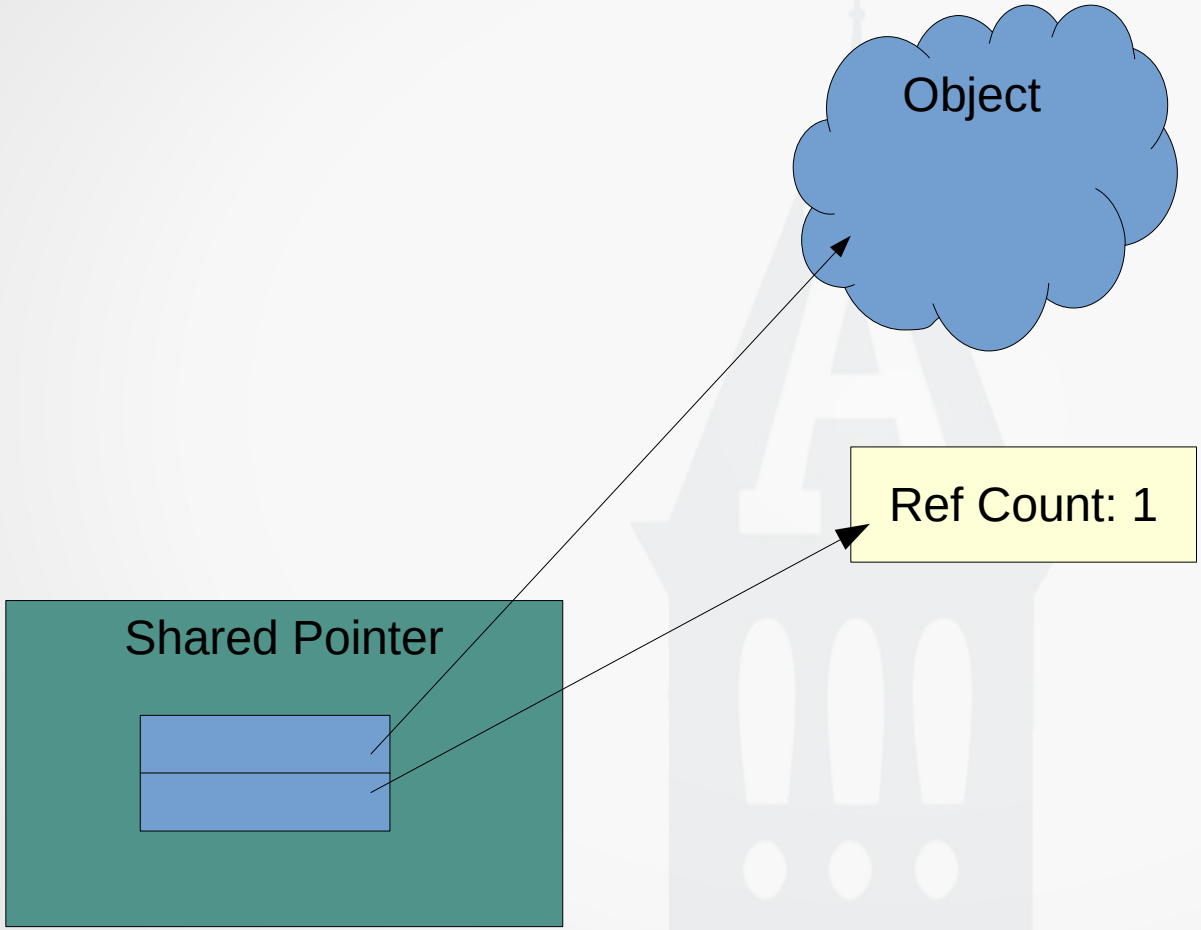
What is a Shared Pointer?



What is a Shared Pointer?



What is a Shared Pointer?



Smart Pointers & Raw Arrays

```
std::shared_ptr<int[]> primes = std::make_shared<int[]>(4);
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;

for (int index = 0; index < 4; index++)
{
    std::cout << primes[index] << std::endl;
}
```

- Note the type is `int[]`, and the parameter is the size
- Can't use an initializer list with `std::make_unique`
- Still don't recommend you do arrays this way, just that you can
  - Use `std::array` or `std::vector`

Smart Pointers & Raw Arrays

```
std::shared_ptr<int[]> primes = std::make_shared<int[]>(4);
primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;

for (int index = 0; index < 4; index++)
{
    std::cout << primes[index] << std::endl;
}
```

- Surprisingly…
  - it wasn't included as part of C++ 11/14/17
  - Finally arrived in C++ 20

Resource Management

- Shared pointer constructor has an overloaded constructor that accepts two parameters
  - raw pointer to dynamically allocated memory
  - a *deleter* function
    - It accepts a raw pointer to the shared pointer type
    - This function can do anything you want, including cleaning up the memory

```
void cleanupArray(int* p)
{
    delete[] p;
}

. . .

std::shared_ptr<int[]> primes(new int[4]{2, 3, 5, 7}, cleanupArray);
```

(preventing) The Copy Penalty

- Small performance penalty each time a shared pointer is copied
  - Most commonly as a parameter to a function
- Solutions
  - Pass it as a const reference
  - Obtain and pass the raw pointer
    - `.get()` - obtains the raw pointer
    - If ownership isn't an issue, this is a good idea

Code Demo – Preventing The Copy Penalty

const shared pointer reference
const shared pointer reference to const data
raw pointer

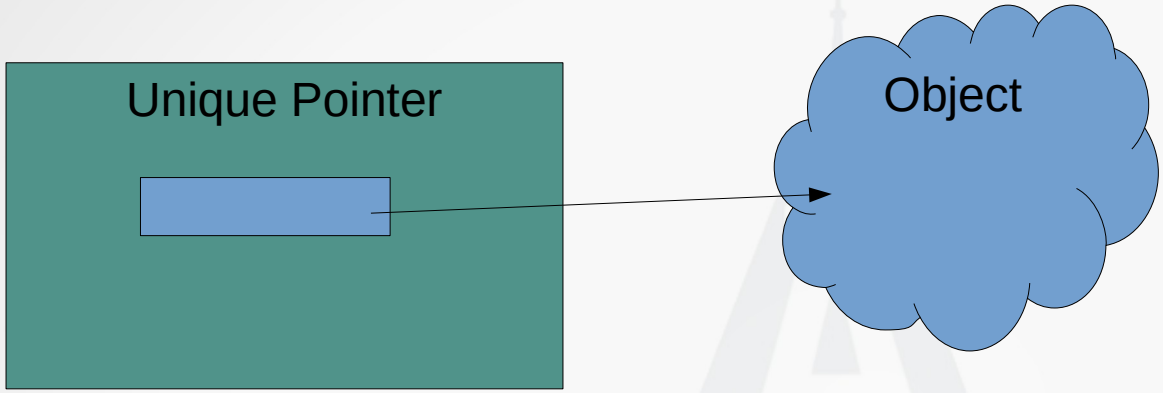
Unique Smart Pointers

- Sometimes it is desirable to allow only one pointer having ownership of an object; and guarantee that (by compiler)
  - Think about concurrency
  - Can’t do this with raw pointers...can copy
  - Can’t do this with shared pointers...can copy
- Enter `std::unique_ptr`
  - Mostly like `std::shared_ptr`, but can’t be copied
  - Can only be *moved*: `std::move(...)`

What is a Unique Pointer?

- It is an object; a C++ class
- Internal to the class
  - A raw pointer to the dynamically allocated memory
  - (no reference count)
- Can not copy, compiler guarantee!
  - you'll learn how to do this very soon
- When a unique pointer goes out of scope
  - raw pointer is cleaned up

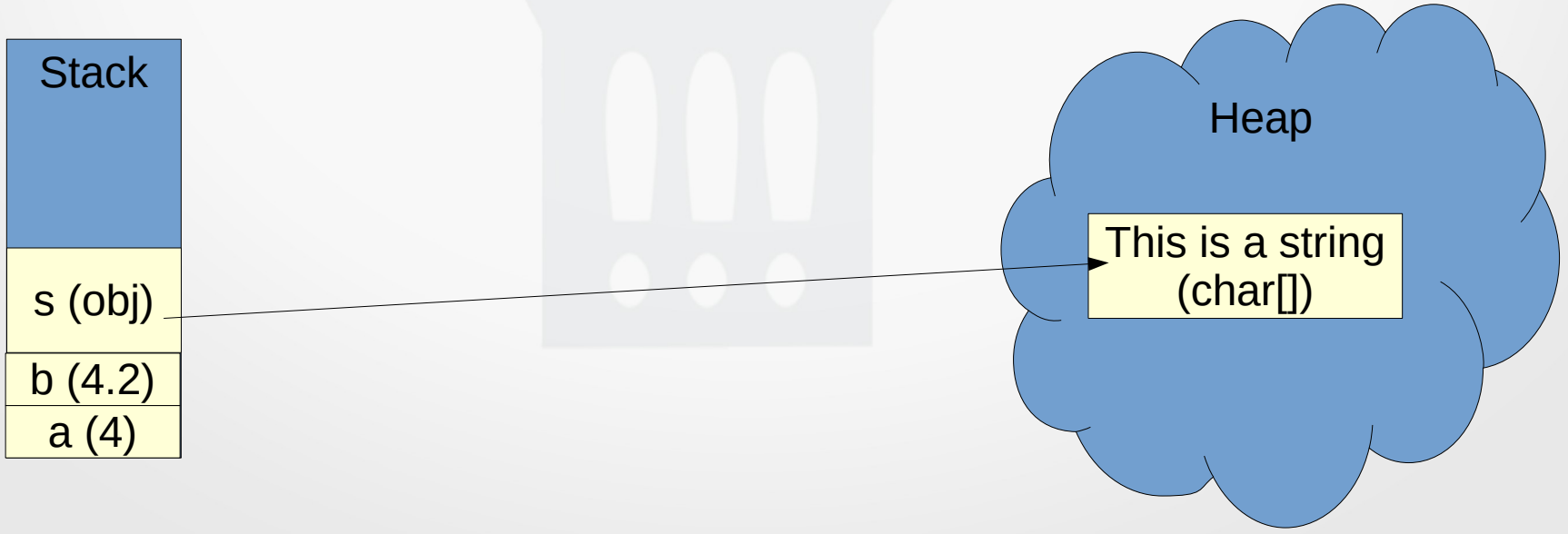
What is a Unique Pointer



Code Demo – Unique Pointers

Classes & Dynamic Memory

- In Java, all objects are heap allocated
- In C++, an object may be heap or stack allocated
  - Could even have a stack allocated object that also has heap allocated memory.



We'll use this class for demonstration

```
class Rectangle
{
  public:
    Rectangle(double width, double height) :
        m_width(width),
        m_height(height)
    {
    }

    double getArea() { return m_width * m_height; }
    double getPerimeter() { return m_width * 2 + m_height * 2; }
    double getWidth() { return m_width; }
    double getHeight() { return m_height; }

  private:
    double m_width;
    double m_height;
};
```

Member Access Syntax

- Let's start with the following…

```
Rectangle r1(2, 4);
std::cout << "(width, height) = ("
    << r1.getWidth() << ", "
    << r1.getHeight() << ")"
    << std::endl;
```

- `r1` is what I'll call a ***value***; it isn't a pointer, it is a value
  - C++ actually calls this a *copyable*
  - In this case it is on the stack, but a value can also be in the heap
- The dot `.` operator is used for member access

Member Access Syntax

- The next example...

```
Rectangle* r2 = new Rectangle(4, 6);
std::cout << "(width, height) = ("
     << r2->getWidth() << ", "
     << r2->getHeight() << ")"
     << std::endl;
delete r2;
```

- `r2` is pointer
  - In this case the `Rectangle` is heap allocated, but remember you can still obtain pointers to heap allocated objects
- The pointer `->` operator is used for member access

Member Access Syntax

- The final example

```
std::shared_ptr<Rectangle> r3 = std::make_shared<Rectangle>(6, 8);
std::cout << "(width, height) = ("
    << r3->getWidth() << ", "
    << r3->getHeight() << ")"
    << std::endl;
```

- `r3` is a smart pointer
- The pointer `->` operator is used for member access
  - exactly the same as a raw pointer