# C++ Smart Pointers

CSE 333

**Instructor:** Alex Sanchez-Stern

**Teaching Assistants:**

Justin Tysdal

Sayuj Shahi

Nicholas Batchelder

Leanna Mi Nguyen

# Administrivia

- Exercise 13 is due **Monday (July 29th)**

- HW3 due **next Thursday (August 1st)**

# Lecture Outline

- **Abstract Classes**
- Smart Pointers
  - Intro and `toy_ptr`
  - `std::unique_ptr`
  - `std::shared_ptr` and `std::weak_ptr`

# Abstract Classes

- Sometimes we want to include a function in a class *just for overriding*
  - In Java, we would use an abstract method
  - In C++, we use a "pure virtual" function
    - Example: `virtual string noise() = 0;`
- A class containing *any* pure virtual methods is abstract
  - You can't create instances of an abstract class
  - Extend abstract classes and override methods to use them
- A class containing *only* pure virtual methods is the same as a Java interface **used to be**
  - Pure type specification without implementations

# Lecture Outline

- Abstract Classes
- **Smart Pointers**
  - **Intro and `toy_ptr`**
  - `std::unique_ptr`
  - `std::shared_ptr` and `std::weak_ptr`

- Reference: *C++ Primer*, Chapter 12.1

# Last Week…

- We learned about STL

- We noticed that STL was doing an enormous amount of copying

- A solution: store pointers in containers instead of objects
  - But who's responsible for deleting and when???

# C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to heap-allocated data
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - With correct use of smart pointers, you no longer have to remember when to `delete` heap memory! (*If* it's owned by a smart pointer)
    - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
      - When that is depends on what kind of smart pointer you use

# A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that frees the pointer
  - Overloaded `*` and `->` operators that access the pointer

# ToyPtr Class Template

ToyPtr.h

```
#ifndef TOYPTR_H_
#define TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  explicit ToyPtr(T *ptr) : ptr_(ptr) { }  // constructor
  ~ToyPtr() { delete ptr_; }               // destructor
  T &operator*()  { return *ptr_; }        // * operator
  T *operator->() { return ptr_;  }        // -> operator

 private:
  T *ptr_;                                 // the pointer
};

#endif  // TOYPTR_H_
```

This is weird! The overload for the -> operator behaves differently than others

# ToyPtr Example

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"

// simply struct to illustrate the "->" operator
typedef struct { int x = 1, y = 2; } Point;
std::ostream &operator<<(std::ostream &out, const Point &rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char **argv) {
  // Create a dumb pointer
  Point *leak = new Point;

  // Create a "smart" pointer
  ToyPtr<Point> notleak(new Point);

  std::cout << "     *leak: " << *leak << std::endl;
  std::cout << "   leak->x: " << leak->x << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  std::cout << "notleak->x: " << notleak->x << std::endl;

  return 0;
}
```

# ToyPtr Example

usetoy.cc

```
#include <iostream>
#include "ToyPtr.h"
```

```
==2554== Memcheck, a memory error detector
==2554== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==2554== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==2554== Command: ./usetoy
==2554==
     *leak: (1,2)
   leak->x: 1
  *notleak: (1,2)
notleak->x: 1
==2554==
==2554== HEAP SUMMARY:
==2554==     in use at exit: 8 bytes in 1 blocks
==2554==   total heap usage: 4 allocs, 3 frees, 74,768 bytes allocated
```

```
  std::cout << "     *leak: " << *leak << std::endl;
  std::cout << "   leak->x: " << leak->x << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  std::cout << "notleak->x: " << notleak->x << std::endl;

  return 0;
}
```

# What Makes This a Toy?

- Can't handle:
  - Arrays
  - Copying
  - Reassignment
  - Comparison
  - … plus many other subtleties…

- Luckily, others have built non-toy smart pointers for us!

# Lecture Outline

- C++ Inheritance
- **Smart Pointers**
  - Intro and `toy_ptr`
  - **`std::unique_ptr`**
  - `std::shared_ptr` and `std::weak_ptr`

- Reference: *C++ Primer*, Chapter 12.1

# `std::unique_ptr`

- A `unique_ptr<T>` ***takes ownership*** of a pointer
  - Template parameter is the type that the "owned" pointer references (i.e., the `T` in pointer type `T*`)
  - Part of C++'s standard library (C++11)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`'d or falls out of scope

# Using unique_ptr

unique1.cc

```
#include <iostream>  // for std::cout, std::endl
#include <memory>    // for std::unique_ptr
#include <cstdlib>   // for EXIT_SUCCESS

void Leaky() {
  int *x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
} // never used delete, therefore leak

void NotLeaky() {
  std::unique_ptr<int> x(new int(5));  // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
} // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS;
}
```

# Why are `unique_ptr`s useful?

- If you have many potential exits out of a function, it's easy to forget to call `delete` on all of them
  - `unique_ptr` will `delete` its pointer when it falls out of scope
  - Thus, a `unique_ptr` also helps with *exception safety*

```
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  ...
  // lots of code, including several returns
  // lots of code, including potential exception throws
  ...
}
```

# unique_ptr Operations

unique2.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5))

  int val = *x;              // Return the va      pointed-to object
  int *ptr = x.get(); // Return a po        to pointed-to object

  // Access a field or function         pointed-to object
  unique_ptr<IntPair> ip(new I     air);
  ip->a = 100;

  // Deallocate current pointed-to object and store new pointer
  x.reset(new int(1));

  ptr = x.release();  // Release responsibility for freeing
  delete ptr;
  return EXIT_SUCCESS;
}
```

ptr is invalid after reset!

If we don't do this, the int in x will leak!

# Transferring Ownership

- Use `reset()` and `release()` to transfer ownership
  - `release` returns the pointer, sets wrapped pointer to `nullptr`
  - `reset` `delete`'s the current pointer and stores a new one

# Transferring Ownership

**z owns int(5), x and y own nothing**

unique3.cc

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());  // x abdicates ownership to y
  cout << "x: " << x.get() << endl; // prints "0"
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());
  return EXIT_SUCCESS;
}
```

# unique_ptrs Cannot Be Copied

- std::unique_ptr has disabled its copy constructor and assignment operator
  - You cannot copy a unique_ptr, helping maintain "uniqueness" or "ownership"

uniquefail.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));  // OK

  std::unique_ptr<int> y(x);           // fail – no copy ctor

  std::unique_ptr<int> z;              // OK – z is nullptr

  z = x;                               // fail – no assignment op

  return EXIT_SUCCESS;
}
```

# unique_ptrs Cannot Be Copied

- std::unique_ptr has disabled its copy constructor and assignment operator
  - You cannot copy a unique_ptr, helping maintain “uniqueness” or “ownership”

uniquefail.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));  // line 1

  std::unique_ptr<int> y(x);           // line 2

  std::unique_ptr<int> z;              // line 3

  z = x;                               // line 4
  return EXIT_SUCCESS;
}
```

# `unique_ptr` and STL

- `unique_ptr`s *can* be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and `unique_ptr`s cannot be copied…
- Move semantics to the rescue!
  - When supported, STL containers will *move* rather than *copy*
    - `unique_ptr`s support move semantics

# Aside: Copy Semantics

- Assigning values typically means making a copy
  - Sometimes this is what you want
    - *e.g.* assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - *e.g.* assigning a returned string goes through a temporary copy

copysemantics.cc

```
std::string ReturnFoo(void) {
  std::string x("foo");
  return x;          // this return might copy
}
int main(int argc, char **argv) {
  std::string a("hello");
  std::string b(a);  // copy a into b

  b = ReturnFoo();   // assign return value into b

  return EXIT_SUCCESS;
}
```

# Move Semantics (added in C++11)

- "Move semantics" move values from one object to another without copying
  - Useful for optimizing away temporary copies
  - A complex topic that uses things called *"rvalue references"*
    - Mostly beyond the scope of 333 this quarter

movesemantics.cc

```
std::string ReturnFoo(void) {
  std::string x("foo");
  // this return might copy
  return x;
}

int main(int argc, char **argv) {
  std::string a("hello");

  // moves a to b
  std::string b = std::move(a);
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;

  // moves the returned value into b
  b = std::move(ReturnFoo());
  std::cout << "b: " << b << std::endl;

  return EXIT_SUCCESS;
}
```

# Transferring Ownership via Move

- `unique_ptr` supports move semantics
  - Can "move" ownership from one `unique_ptr` to another
    - Behavior is equivalent to the "release-and-reset" combination

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y = std::move(x);  // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z = std::move(y);

  return EXIT_SUCCESS;
}
```

equivalent to: `unique_ptr<int> y(x.release())`

equivalent to: `z.reset(y.release())`

# unique_ptr and STL Example

uniquevec.cc

```
int main(int argc, char **argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  // z gets a copy of int value pointed to by vec[1]
  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  // won't compile!  Cannot copy unique_ptr
  std::unique_ptr<int> copied = vec[1];

  // Works! vec[1] now wraps a nullptr
  std::unique_ptr<int> moved = std::move(vec[1]);
  std::cout << "*moved: " << *moved << std::endl;
  std::cout << "vec[1].get(): " << vec[1].get() << std::endl;
  return EXIT_SUCCESS;
}
```

**No leaks!**

# `unique_ptr` and "<"

- A `unique_ptr` implements some comparison operators, including `operator<`
  - However, it doesn't invoke `operator<` on the pointed-to objects
    - Instead, it just promises a stable, strict ordering (probably based on the pointer address, not the pointed-to-value)
  - So to use **`sort()`** on `vector`s, you want to provide it with a comparison function

```
template <class Iter, class T>
  sort(Iter begin_it, Iter end_it,
       bool (*sort_function)(T, T));
```

# unique_ptr and STL Sorting

uniquevecsort.cc

```
using namespace std;
bool sortfunction(const unique_ptr<int> &x,
                  const unique_ptr<int> &y) { return *x < *y; }
void printfunction(unique_ptr<int> &x) { cout << *x << endl; }

int main(int argc, char **argv) {
  vector<unique_ptr<int>> vec;
  vec.push_back(unique_ptr<int>(new int(9)));
  vec.push_back(unique_ptr<int>(new int(5)));
  vec.push_back(unique_ptr<int>(new int(7)));

  // buggy: sorts based on the values of the ptrs
  sort(vec.begin(), vec.end());
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  // better: sorts based on the pointed-to values
  sort(vec.begin(), vec.end(), &sortfunction);
  cout << "Sorted:" << endl;
  for_each(vec.begin(), vec.end(), &printfunction);

  return EXIT_SUCCESS;
}
```

# unique_ptr and Arrays

- unique_ptr can store arrays as well
  - Will call delete[] on destruction

unique5.cc

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

# Lecture Outline

- C++ Inheritance
- **Smart Pointers**
  - Intro and `toy_ptr`
  - `std::unique_ptr`
  - **`std::shared_ptr` and `std::weak_ptr`**

- Reference: *C++ Primer*, Chapter 12.1

# std::shared_ptr

- shared_ptr is similar to unique_ptr but we allow shared data to have multiple owners
  - How? Reference counting!

# What is Reference Counting?

- Idea: associate a *reference count* with each object
  - Reference count holds number of references (pointers) to the object
  - Adjust reference count whenever pointers are changed:
    - Increase by 1 each time we have a new pointer to an object
    - Decrease by 1 each time a pointer to an object is removed
  - When reference counter decreased to 0, no more pointers to the object, so delete it (automatically)

# `std::shared_ptr`

- `shared_ptr` uses reference counting
  - The copy/assign operators are not disabled; instead they *increment* or *decrement* reference counts as needed
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!
  - Allows us to have multiple smart pointers to the same object and still get automatic cleanup
    - At the cost of maintaining reference counts at runtime

# shared_ptr Example

sharedexample.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count: 1

  // temporary inner scope with local y (!)
  {
    std::shared_ptr<int> y = x;          // ref count: 2
    std::cout << *y << std::endl;
  }                                      // exit scope, y deleted

  std::cout << *x << std::endl;         // ref count: 1

  return EXIT_SUCCESS;
}                                        // ref count: 0
```

# shared_ptrs and STL Containers

- Even simpler than unique_ptrs
  - Safe to store shared_ptrs in containers, since copy & assign maintain a shared reference count

sharedvec.cc

```
vector<std::shared_ptr<int> > vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1];  // works!
std::cout << "*copied: " << *copied << std::endl;

std::shared_ptr<int> moved = std::move(vec[1]);  // works!
std::cout << "*moved: " << *moved << std::endl;
std::cout << "vec[1].get(): " << vec[1].get() << std::endl;
```

# RefLang

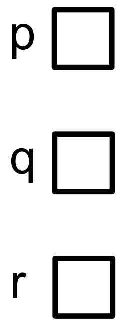
- Suppose for the moment that we have a new C++ -like language that uses reference counting for heap data
- As in C++, a struct is a type with public fields, so we can implement lists of integers using the following Node type

```
struct Node {
  int payload;   // node payload
  Node * next;   // next Node or nullptr
};
```

- The reference counts are handled behind the scenes by the memory manager code – they are not accessible to the programmer

# Example 1

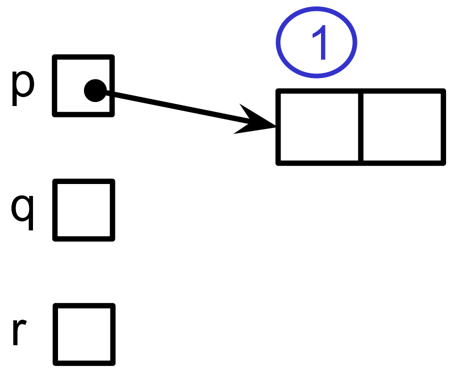
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

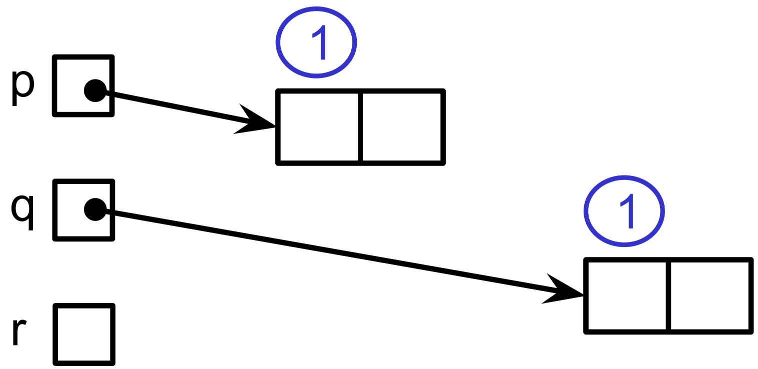
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

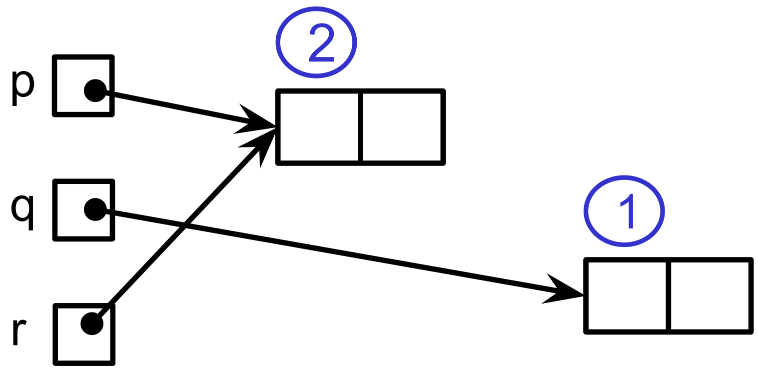
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

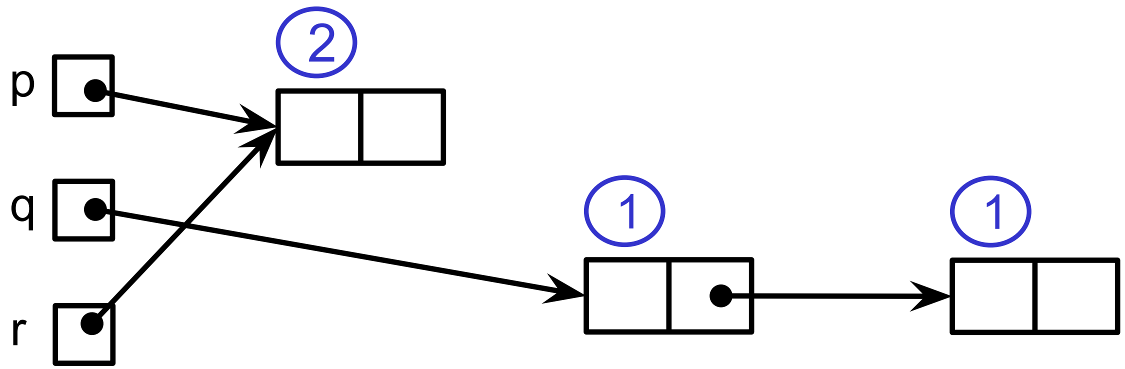
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

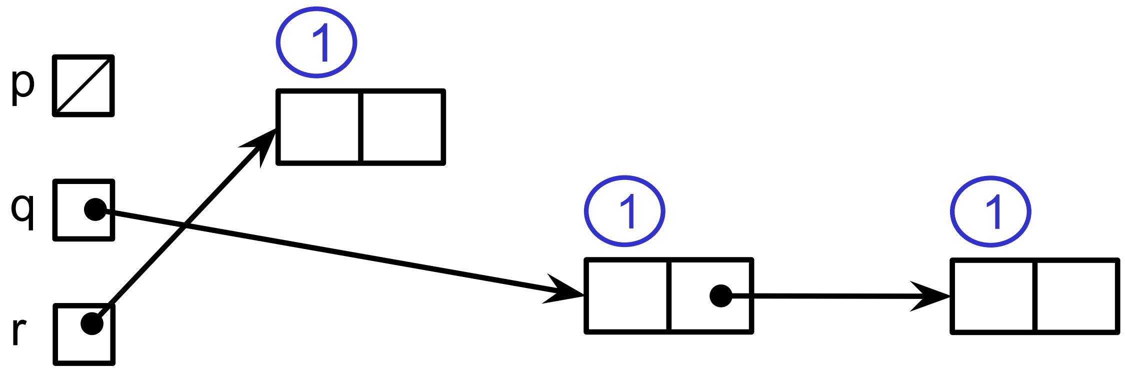
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

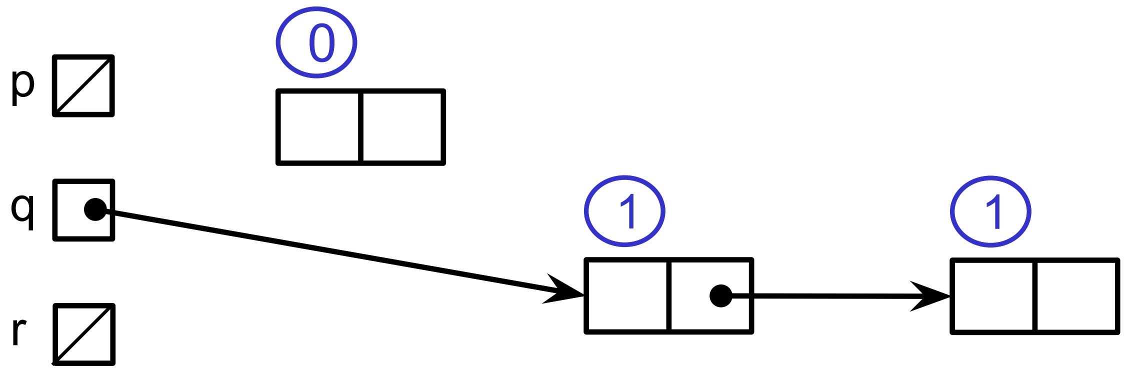
- Let’s execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

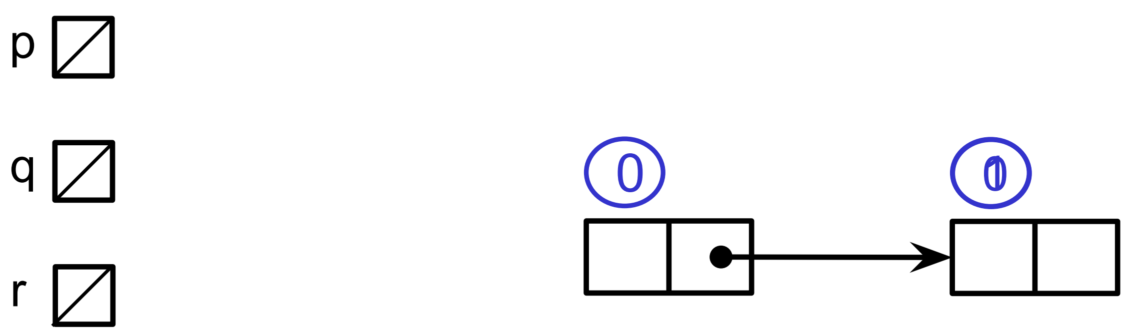
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

# Example 1

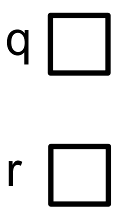
- Let's execute the following code. Heap data is shown using rectangles; associated reference counts with ovals




```
Node * p = new Node();
Node * q = new Node();
Node * r = p;
q->next = new Node();
p = nullptr;
r = nullptr;
q = nullptr;
```

Poll Everywhere pollev.com/uwcse333

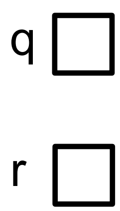
- What is the box-and-arrow diagram for this slightly-different snippet, when it finishes execution?




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

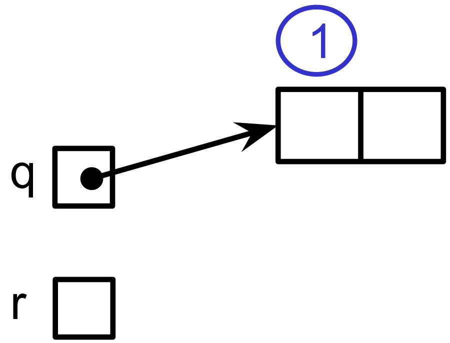
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

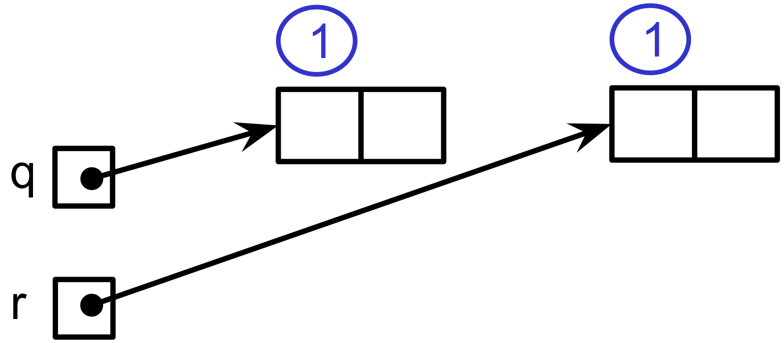
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

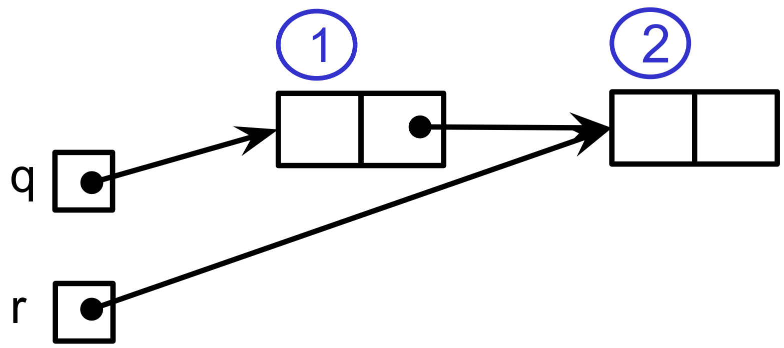
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

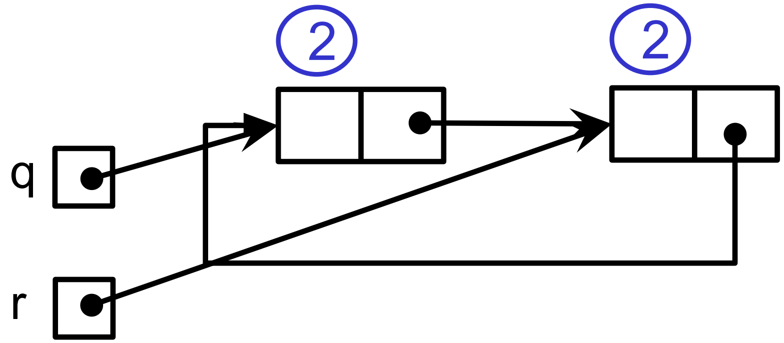
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

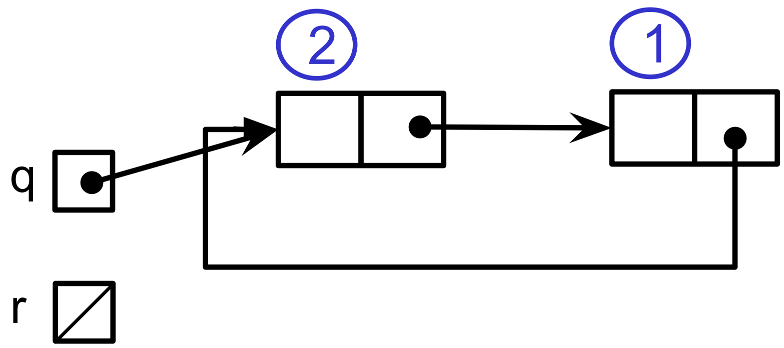
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

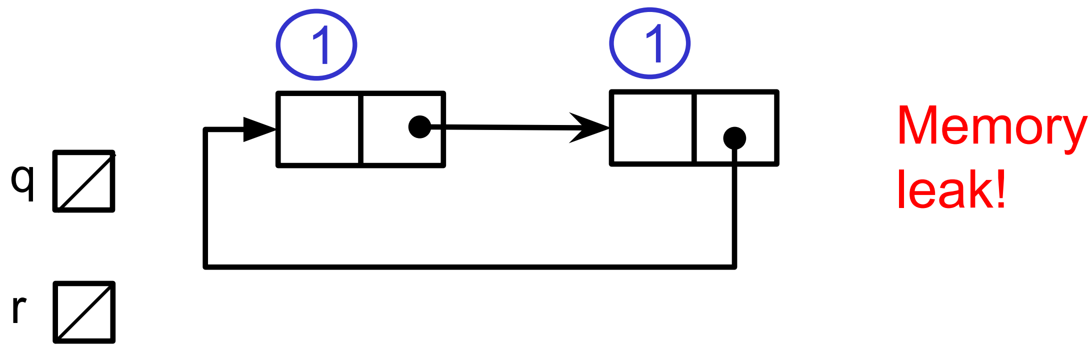
- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Example 2

- Similar to the previous code, but slightly different




```
Node * q = new Node();
Node * r = new Node();
q->next = r;
r->next = q;
r = nullptr;
q = nullptr;
```

# Cycle of `shared_ptrs`

- `shared_ptr`s are deleted when their reference count drops to 0

- Linked data structures with cycles don't play nicely with that …

# Cycle of shared_ptrs

strongcycle.cc

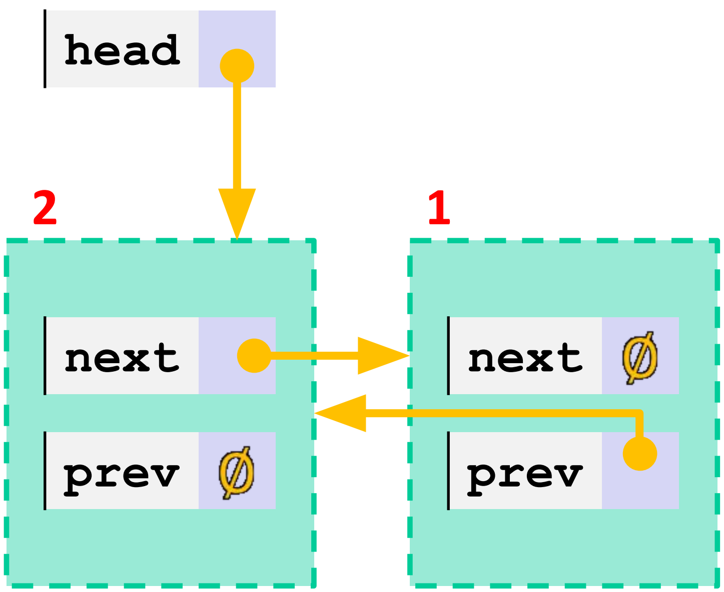
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- What happens when we `delete head`?

# Cycle of shared_ptrs

strongcycle.cc

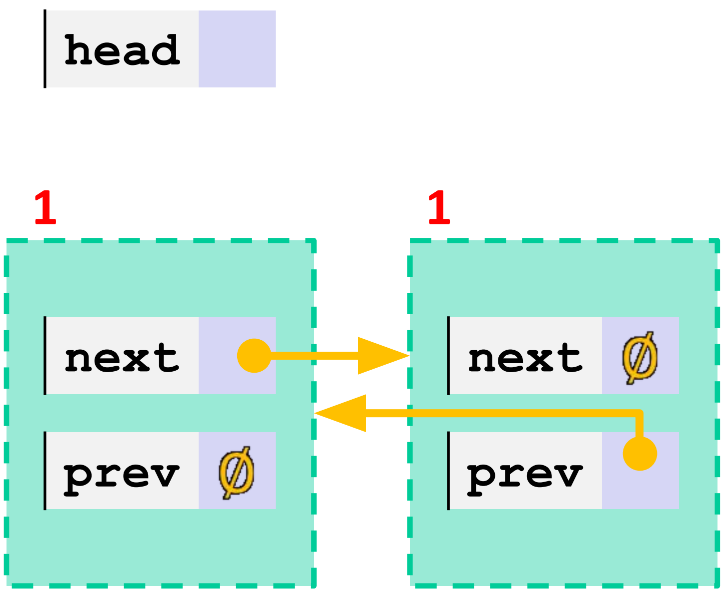
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- What happens when we `delete head`? Nodes unreachable but not deleted because ref counts > 0

# `std::weak_ptr`

- `weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count
  - Can *only* "point to" an object that is managed by a `shared_ptr`
  - Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
    - Object referenced may have been `delete`'d
  - Can't actually dereference unless you check if the object still exists
    - Then you can "get" its associated `shared_ptr`

- Can be used to fix our cycle problem!

# Breaking the Cycle with weak_ptr

weakcycle.cc

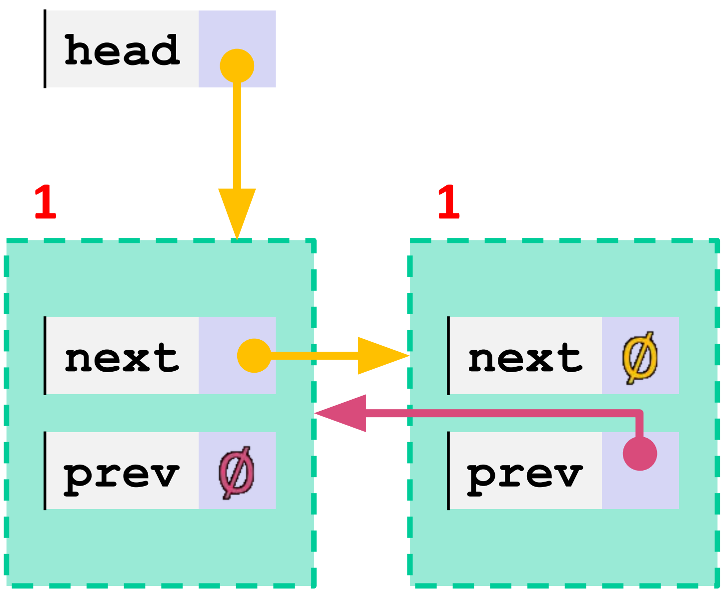
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- Now what happens when we `delete` `head`?

# Breaking the Cycle with `weak_ptr`

weakcycle.cc

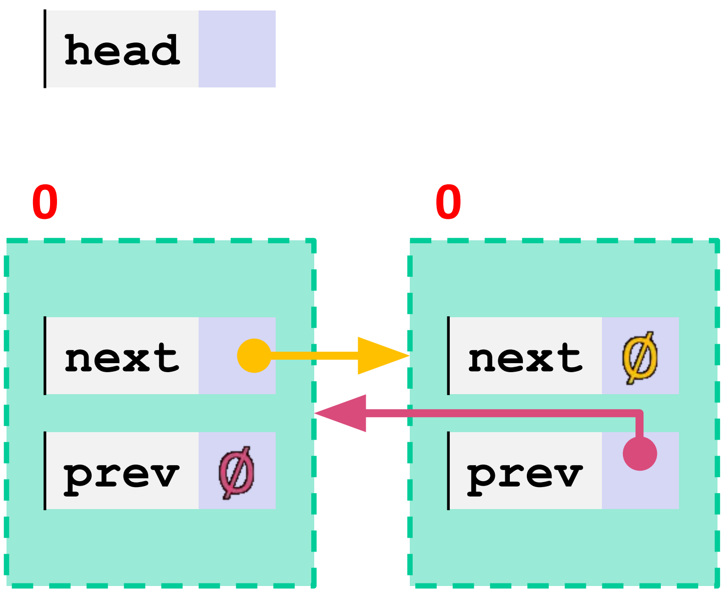
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```




- Now what happens when we `delete head`? Ref counts go to 0 and nodes deleted!

# Using a `weak_ptr`

- `lock()`: returns an "upgraded" `weak_ptr` to a `shared_ptr`
  - First checks if the data still exists, if not returns null
  - Otherwise, creates a `shared_ptr` pointing to the same data as `this`

# Using a weak_ptr

usingweak.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr, std::weak_ptr

int main(int argc, char **argv) {
  std::weak_ptr<int> w;
  {  // temporary inner scope with local x
    std::shared_ptr<int> x;
    {  // temporary inner-inner scope with local y
      std::shared_ptr<int> y(new int(10));
      w = y;            // weak ref; ref count for "10" node is same
      x = w.lock();     // get "promoted" shared_ptr, ref cnt = 2
      std::cout << *x << std::endl;
    } // y deleted; ref count now 1
    std::cout << *x << std::endl;
  } // x deleted; ref count now 0; mem freed

  std::shared_ptr<int> a = w.lock();  // nullptr
  std::cout << a << std::endl;        // output is 0 (null)

  return EXIT_SUCCESS;
}
```

# Using a `weak_ptr`

- `lock()`: returns an "upgraded" `weak_ptr` to a `shared_ptr`
  - First checks if the data still exists, if not returns null
  - Otherwise, creates a `shared_ptr` pointing to the same data as `this`

- `use_count()`: gets reference count
- `expired()`: returns `(use_count() == 0)`

# Caveat: `shared_ptrs` Must Share Nicely

- A warning: `shared_ptr` reference counting works as long as the shared references to the same object result from making copies of existing `shared_ptr` values

# shared_ptr Caveat

sharedbug.cc

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count: 1
  std::shared_ptr<int> y(x);            // ref count: 2

  int *p = new int(10);
  std::shared_ptr<int> xbug(p);   // ref count: 1
  std::shared_ptr<int> ybug(p);   // separate ref count: 1

  return EXIT_SUCCESS;
}                               // x and y ref count: 0 – ok delete
                                // xbug and ybug ref counts both 0
                                 //   both try to delete p
                                 //   -- double-delete error!
```

# Caveat: shared_ptrs Must Share Nicely

- If we create multiple shared_ptrs using the same raw pointer, the shared_ptrs will have separate reference counts.
  - Causes double deletes!
  - Good practice: allocate with new and create shared_ptr in the same line.

```
std::shared_ptr<int> x(new int(10));
```

**Good**

```
int *p = new int(10);
std::shared_ptr<int> x(p);
```

**Bad**

# Reference Counting Perspective

- Reference counting works great! But…
  - Extra overhead on every pointer copy or delete
  - Not general enough for the language to do it automatically
    - Cannot reclaim linked objects with circular references

# Summary

- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved
  - `get()` returns a copy of the pointer, but is dangerous to use; better to use `release()` instead
  - `reset()` `delete`s old pointer value and stores a new one
- A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
  - `delete`s an object once its reference count reaches zero
- A `weak_ptr` works with a shared object but doesn't affect the reference count
  - Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does

# Don't Forget!

- Exercise 13 is due **Monday (July 29th)**

- HW3 due **next Thursday (August 1st)**