Smart Pointers

CMSC 240

Many examples thanks to the text *C++ Primer Plus* by Stephen Prata

linked off our useful resources page

RAII

- Recall: Resource Acquisition Is Initilization
- A C++ programming idiom/mantra/philosophy/technique
- You'll see it in a lot of guides to programming C++, so you should know what it means

RAII

- The problem: Resources are sometimes required to be allocated from the heap
  - E.g., static variables, locks
- These resources have to be released at some point
  - If not, memory leak: a long running program with a memory leak will slowly run out of memory, which can kill performance

RAII

- You don’t have any long running programs?
  - Do you keep a web browser open?
  - Do you sometimes keep Microsoft Word or other text editing programs open while you are creating documents?
  - Do you keep your Outlook Mail program open for days at a time?
  - Then you have long running programs
    - And so do airlines, ISPs, etc.

RAII

- So, dynamically allocating memory is not a problem as long as you remember to deallocate that memory when you’re done with it.
- General advice: (Thanks Steven Prata (from C++ Primer Plus): “..a solution involving the phrase ‘just remember to’ is seldom the best solution.”

RAII

- But consider: memory allocated automatically (on the stack) is automatically deallocated when it goes out of scope
- Thought: Can we somehow give ownership of a resource allocated dynamically to an object that is allocated automatically
  - If so, the dynamic resource can be returned when the owning resource goes out of scope (in destructor call)

Standard Example

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    str = ps;
    return;
}
```

- Traditional memory leak: the memory dynamically allocated to `ps` is never released
  - This is wrong on several levels. Why?

Better (Correct) Example

```
class TestClass{

public:
    TestClass(){
        str = new char[1000];
    }
    ~TestClass(){}

  private:
    char* str;

};

int main() {

  TestClass myTestClassArray[1000];
  return 0;
}
```

But It's Not Just Carelessness

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

- Here the programmer remembers to include `delete`, but statement is never reached if exception is thrown
  - This also has issues. What?

But It's Not Just Carelessness

- Note: When `remodel()` terminates, no matter for what reason, its resources are released
  - So the memory occupied by `ps` is released
  - But NOT the memory it points to
- It would be nice if memory pointed to by `ps` was released as well
- If `ps` had a destructor, memory could be released there

Smart Pointers

- But alas, `ps` is just an ordinary pointer, not a class object, so it has no destructor
- If it were an object, then we could code a destructor and the memory would be freed on termination, for whatever reason, of `remodel()`
- This is the idea behind smart pointers
  - C++ 98: `auto_ptr` (deprecated)
  - Modern C++: `unique_ptr`, `shared_ptr`, `weak_ptr`

Smart Pointers

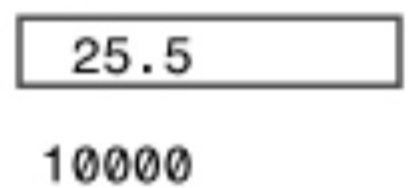
- Though `auto_ptr` has been deprecated, we will still cover it, because you may run into it (or, less likely, end up with an implementation of C++ that is older than C++11)
- Also, we won’t focus much on `weak_ptr`
- And note that all of these ptr classes are templated: you specify the data type pointed to

```
void demol()
{
     double * pd = new double; // #1
     *pd = 25.5;               // #2
     return;                   // #3
}
```

#1: Creates storage for pd and a double value, saves address:

#2: Copies value into dynamic memory:

#3: Discards pd, leaves value in dynamic memory:



Smart Pointers

```
void demo2()
{
      auto_ptr<double> ap(new double); // #1
      *ap = 25.5;                      // #2
      return;                          // #3
}
```

#1: Creates storage for ap and a `double` value, saves address:

#2: Copies value into dynamic memory:

#3: Discards ap, and ap's destructor frees dynamic memory.

Smart Pointers

- All smart pointers in the `memory` header

```
#include <memory>
void remodel(std::string & str)
{
    std::auto_ptr<std::string> ps (new std::string(str));
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    // delete ps;  NO LONGER NEEDED
    return;
}
```

Modern C++ Smart Pointers

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

class Report
{
private:
    std::string str;
public:
    Report(const std::string s) : str(s) { std::cout << "Object created!\n"; }
    ~Report() { std::cout << "Object deleted!\n"; }
    void comment() const { std::cout << str << "\n"; }
};

int main()
{
    {
        std::shared_ptr<Report> ps (new Report("using shared_ptr"));
        ps->comment();    // use -> to invoke a member function
    }
    {
        std::unique_ptr<Report> ps (new Report("using unique_ptr"));
        ps->comment();
    }
    return 0;
}
```

Note each smart ptr declared in a block so ptr expires when execution leaves the block

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

class Report
{
private:
    std::string str;
public:
    Report(const std::string s) : str(s) { std::cout << "Object created!\n"; }
    ~Report() { std::cout << "Object deleted!\n"; }
    void comment() const { std::cout << str << "\n"; }
};

int main()
{
    {
        std::shared_ptr<Report> ps (new Report("using shared_ptr"));
        ps->comment();   // use -> to invoke a member function
    }
    {
        std::unique_ptr<Report> ps (new Report("using unique_ptr"));
        ps->comment();
    }
    return 0;
}
```

```
(base) m1-mcs-dszajda:Chapter 16 dszajda$ ./smrtptrs
Object created!
using shared_ptr
Object deleted!
Object created!
using unique_ptr
Object deleted!
```

Guidelines For Smart Pointers

- In most cases, when one initializes a raw pointer (or other handle to a resource), pass the pointer to a smart pointer immediately
  - Microsoft docs: “In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.”

Thanks Microsoft, for this and the following code examples and guidelines
https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160

Guidelines For Smart Pointers

- Effectively, a smart pointer is a wrapper for a raw pointer
- Access the encapsulated pointer using the usual operators -> and *, which the smart pointer class overloads so that they return the encapsulated raw pointer

Guidelines For Smart Pointers

```
#include <memory>

class LargeObject {

public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}

void SmartPointerDemo()  {

    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);

} //pLarge is deleted automatically when function block goes out of scope.
```

Note usual pointer syntax

Essential Steps

1. Declare smart pointer as an automatic (local) variable
   - Do **NOT** use the `new` or `malloc` expression on the smart pointer itself (Why not?)
2. In the type parameter, specify the pointed-to type of the encapsulated pointer
3. Pass a raw pointer to a new-ed object in the smart pointer constructor
   - Some utility functions and smart pointer constructors do this for you

Essential Steps

4. Use the overloaded -> and * operators to access the object
5. Let the smart pointer delete the object

- And one other thing to avoid:

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);  // NO!
```

- What is the issue here?

Essential Steps

4. Use the overloaded `->` and `*` operators to access the object
5. Let the smart pointer delete the object

- And one other thing to avoid:

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);  // NO!
```

- When `pvac` expires, program applies `delete` operator to non-heap memory!

Performance

- Smart pointers are designed to be as efficient as possible in terms of both memory and performance
  - The only data member in unique_ptr is the encapsulated pointer (so memory required is exactly the same as for the raw pointer)
- The overloaded operators -> and * are not significantly slower than using raw pointers directly

Member Functions

- Smart pointers have their own member functions which are accessed via the usual "dot" notation
  - E.g., some smart pointers have a `reset()` method which releases the pointed to memory before the smart pointer goes out of scope

Member Functions

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...

}
```

Legacy Code

- Smart pointers provide methods that allow access to the encapsulated raw pointer
    - Which might be needed if one has to deal with legacy code that does not accept smart pointers
    - Use the `get()` method to access raw pointer
- So you can manage memory in your own code, but pass raw pointer if necessary

Legacy Code

```
void SmartPointerDemo4()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass raw pointer to a legacy API
    LegacyLargeObjectFunction(pLarge.get());
}
```

Smart Pointer Considerations

- Why are there four smart pointers (well three now) and why was `auto_ptr` deprecated?
- Well, let's start by considering assignment:

```
auto_ptr<string> ps (new string("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

- Can anyone see the issue here?

Smart Pointer Considerations

- Ways to avoid this issue:
  - Define the assignment so that it makes a deep copy, so that we end up with two distinct equivalent objects
  - Institute the concept of *ownership*, so that only one smart pointer can own an object. When that pointer is destructed, the object is deleted
    - `auto_ptr` and `unique_ptr` both do this, though `unique_ptr` is more restrictive

Smart Pointer Considerations

- Ways to avoid this issue:
  - *Reference counting*: create an even smarter pointer that keeps track of how many smart pointers point to an object.
    - Only when the final pointer expires is the destructor called to release the referenced object
    - This is what `shared_ptr` does
- Note these same strategies would apply to the copy constructor

Smart Pointer Considerations

- There are good uses for each
- Let's look at one example where `auto_ptr` is a problem
- Note: to compile following example, should NOT use the -std=c++17 flag!
    - Many modern C++ compilers will yell that they don't recognize `auto_ptr`

```cpp
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>

int main() {

    using namespace std;
    auto_ptr<string> films[5] =  {
        auto_ptr<string> (new string("Fowl Balls")),
        auto_ptr<string> (new string("Duck Walks")),
        auto_ptr<string> (new string("Chicken Runs")),
        auto_ptr<string> (new string("Turkey Errors")),
        auto_ptr<string> (new string("Goose Eggs"))
    };

    auto_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";

    return 0;
}
```

Note behavior is undefined, so you might get different output

```
(base) m1-mcs-dszajda:Chapter 16 dszajda$ ./fowl
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Segmentation fault: 11
```

```cpp
// fowl.cpp  -- auto_ptr a poor choice
#include <iostream>
#include <string>
#include <memory>

int main() {

    using namespace std;
    auto_ptr<string> films[5] =  {
        auto_ptr<string> (new string("Fowl Balls")),
        auto_ptr<string> (new string("Duck Walks")),
        auto_ptr<string> (new string("Chicken Runs")),
        auto_ptr<string> (new string("Turkey Errors")),
        auto_ptr<string> (new string("Goose Eggs"))
    };

    auto_ptr<string> pwin;
    pwin = films[2];   // films[2] loses ownership

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";

    return 0;
}
```

- The problem: When `films[2]` is assigned to `pwin`, ownership is transferred and `films[2]` no longer points to the object
  - `films[2]` becomes a null pointer

```cpp
// fowlsp.cpp  -- shared_ptr a good choice
#include <iostream>
#include <string>
#include <memory>

int main()
{
    using namespace std;
    shared_ptr<string> films[5] =
    {
        shared_ptr<string> (new string("Fowl Balls")),
        shared_ptr<string> (new string("Duck Walks")),
        shared_ptr<string> (new string("Chicken Runs")),
        shared_ptr<string> (new string("Turkey Errors")),
        shared_ptr<string> (new string("Goose Eggs"))
    };
    shared_ptr<string> pwin;
    pwin = films[2];   // films[2], pwin both point to "Chicken Runs"

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";

    return 0;
}
```

```
(base) m1-mcs-dszajda:Chapter 16 dszajda$ ./fowlsp
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Chicken Runs
Turkey Errors
Goose Eggs
The winner is Chicken Runs!
```

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

int main()
{
    using namespace std;
    unique_ptr<string> films[5] =
    {
        unique_ptr<string> (new string("Fowl Balls")),
        unique_ptr<string> (new string("Duck Walks")),
        unique_ptr<string> (new string("Chicken Runs")),
        unique_ptr<string> (new string("Turkey Errors")),
        unique_ptr<string> (new string("Goose Eggs"))
    };
    unique_ptr<string> pwin;
    pwin = films[2];   // films[2], pwin both point to "Chicken Runs"

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";

    return 0;
}
```

What about this?

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

int main()
{
    using namespace std;
    unique_ptr<string> films[5] =
    {
        unique_ptr<string> (new string("Fowl Balls")),
        unique_ptr<string> (new string("Duck Walks")),
        unique_ptr<string> (new string("Chicken Runs")),
        unique_ptr<string> (new string("Turkey Errors")),
        unique_ptr<string> (new string("Goose Eggs"))
    };
    unique_ptr<string> pwin;
    pwin = films[2];   // films[2], pwin both point to "Chicken Runs"

    cout << "The nominees for best avian baseball film are\n";
    for (int i = 0; i < 5; i++)
        cout << *films[i] << endl;
    cout << "The winner is " << *pwin << "!\n";

    return 0;
}
```

```
(base) m1-mcs-dszajda:Chapter 16 dszajda$ make fowlup
g++ -std=gnu++2a -DDEBUG -g -Wall -c fowlup.cpp
fowlup.cpp:18:10: error: object of type 'std::__1::unique_ptr<std::__1::basic_string<char>,
      std::__1::default_delete<std::__1::basic_string<char> > >' cannot be assigned because its copy assignment operator is implicitly
      deleted
    pwin = films[2];   // films[2], pwin both point to "Chicken Runs"
         ^
/Library/Developer/CommandLineTools/usr/bin/../include/c++/v1/memory:2493:3:       copy assignment operator is implicitly deleted because
      'unique_ptr<std::__1::basic_string<char>, std::__1::default_delete<std::__1::basic_string<char> > >' has a user-declared move
      constructor
  unique_ptr(unique_ptr&& __u) noexcept
  ^
1 error generated.
make: *** [fowlup.o] Error 1
```

Why `unique_ptr` is Better than `auto_ptr`

- Based on the examples, it would seem we need to look into differences between these two
- Consider:

```
auto_ptr<string> p1(new string("auto");   //#1
auto_ptr<string> p2;                       //#2
p2 = p1;                                   //#3
```

  - Good: `p1` stripped of ownership, so no double free
  - Bad: If `p1` is subsequently used

Why `unique_ptr` is Better than `auto_ptr`

- Based on the examples, it would seem we need to look into differences between these two
- Now consider this:

```
unique_ptr<string> p3(new string("auto");   //#4
unique_ptr<string> p4;                      //#5
p4 = p3;                                    //#6
```

  - Compiler won't allow statement #6, so no worry about using `p3` after assignment
  - Result: compile-time error vs. program crash

Why `unique_ptr` is Better than `auto_ptr`

- Consider another example

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp(new string(s));
    return temp;
}
```

This is some code in `main()`

```
unique_ptr<string> ps;
ps = demo("Uniquely special");
```

Why `unique_ptr` is Better than `auto_ptr`

- `demo()` returns a temporary `unique_ptr`, whose ownership is taken over by `ps`
  - The returned `unique_ptr` is then destroyed
  - But it's OK because `ps` now owns the string
  - And because temp is destroyed, no chance of it being misused to access invalid data (so compiler allows it!)

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp(new string(s));
    return temp;
}
     unique_ptr<string> ps;
     ps = demo("Uniquely special");
```

Why `unique_ptr` is Better than `auto_ptr`

- Question: is what is assigned to `ps` an lvalue or an rvalue?

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp(new string(s));
    return temp;
}
     unique_ptr<string> ps;
     ps = demo("Uniquely special");
```

Why `unique_ptr` is Better than `auto_ptr`

- So #1 is not allowed (`pu1` stays around and could cause damage) while #2 is allowed because the temporary `unique_ptr` built in the constructor is destroyed when ownership of the string is passed to `pu3`

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                      //#1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!");    //#2 allowed
```

Recall: Container Classes

- I know you coded quite a few in CS 221, and some in this class
  - dynamic arrays (vector),
  - queues (queue),
  - stacks (stack),
  - heaps (priority_queue),
  - linked lists (list),
  - trees (set),
  - associative arrays (map)...

Why `unique_ptr` is Better than `auto_ptr`

- The selective behavior is one reason that `unique_ptr` is better than `auto_ptr`
- Another: `auto_ptr` is banned (by recommendation, not necessarily enforcement by compiler) for use in Container classes
  - If some container algorithm tries to do something along the lines of #1 in the last example to the contents of a container containing `unique_ptr` objects, you get a compiler-time error.
  - If you do something like #2 with `unique_ptr`, compiler is fine with it
  - If you do something like #1 with `auto_ptr` in a container class, you can get undefined behavior and hard to diagnose crashes

Why `unique_ptr` is Better than `auto_ptr`

- Another: `auto_ptr` is banned (by recommendation, not necessarily enforcement by compiler) for use in Container classes
  - If some container algorithm tries to do something along the lines of #1 in the last example to the contents of a container containing `unique_ptr` objects, you get a compiler-time error.
- What if you really want to do something like #1?
  - After all, it's really only bad if you do something unsafe with the abandoned pointer.
  - So what if you need to do something like #1 (think about how one sometimes creates a temp object to store an element in an ArrayList to swap entries or the like)?

Why `unique_ptr` is Better than `auto_ptr`

- What if you really want to do something like #1?
  - After all, it's really only bad if you do something unsafe with the abandoned pointer.
  - So what if you need to do something like #1 (think about how one sometimes creates a temp object to store an element in an ArrayList to swap entries or the like)?
  - `std::move()` helps us there (recall from move semantics)

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                    //#1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!");   //#2 allowed
```

Why `unique_ptr` is Better than `auto_ptr`

```cpp
#include <iostream>
#include <string>
#include <memory>
using namespace std;

unique_ptr<string> demo(const char * s) {
    unique_ptr<string> temp(new string(s));
    return temp;
}

int main() {
    unique_ptr<string> ps1, ps2;
    ps1 = demo("Uniquely special");
    ps2 = move(ps1);                    // enable assignment
    ps1 = demo(" and more");
    cout << *ps2 << *ps1 << endl;

    return 0;
}
```

Why `unique_ptr` is Better than `auto_ptr`

- How is `unique_ptr` able to discriminate between safe and unsafe uses? It uses move constructors and rvalue references
  - Aspects of C++ that did not exist when `auto_ptr` was designed
- If a program attempts to assign one `unique_ptr` to another, the compiler allows it if the source object is a temporary rvalue and disallows it if the source object has some duration"

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                    //#1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!");   //#2 allowed
```

Why `unique_ptr` is Better than `auto_ptr`

- One final advantage: `unique_ptr` has a variant that can be used with arrays. `auto_ptr` does not.
- Recall that `new` has to be paired with `delete` and `new[]` with `delete[]`
  - `auto_ptr` has no version that handles the latter
  - `unique_ptr` does

```
std::unique_ptr< double[]>pda(new double(5));  // will use delete []
```

Why `unique_ptr` is Better than `auto_ptr`

- One final advantage: `unique_ptr` has a variant that can be used with arrays. `auto_ptr` does not.
- Recall that `new` has to be paired with `delete` and `new[]` with `delete[]`
  - `auto_ptr` has no version that handles the latter
  - `unique_ptr` does
- `auto_ptr` and `shared_ptr` should only be used for memory allocated with `new`, not for memory allocated with `new[]`

Selecting a Smart Pointer

- If your program uses more than one pointer to an object, use `shared_ptr`
  - E.g., you might have an array of pointers and use auxiliary pointers to identify specific elements, like the largest or smallest
  - Or two kind of objects that both have pointers to a third common object
- Or if you have an STL container of smart pointer objects
  - Many STL algorithms include copy or assignment operations that work with `shared_ptr`, but not with `unique_ptr` (compile-time error) or `auto_ptr` (undefined behavior)

Selecting a Smart Pointer

- If your program does not need multiple pointers to the same object, then unique_ptr is usually the choice.
  - Good choice for return type for function that returns a pointer to memory allocated by new
- Can store unique_ptr in a container object as long as you don't use methods that copy or assign one unique_ptr to another
  - E.g., sort()

weak_ptr

- A special-case smart pointer used in conjunction with `shared_ptr`
- A `weak_ptr` provides access to an object owned by one or more `shared_ptr`, but does not participate in reference counting
- Useful when you want to observe an object, but don't require it to stay alive
- Also required in some cases to break circular references between `shared_ptr` instances

Example thanks to LearnCpp.com: https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>

class Person {
	std::string m_name;
	std::shared_ptr<Person> m_partner; // initially created empty

public:

	Person(const std::string &name): m_name(name) {
		std::cout << m_name << " created\n";
	}
	~Person() {
		std::cout << m_name << " destroyed\n";
	}

	static bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {
		if (!p1 || !p2)
			return false;

		p1->m_partner = p2;
		p2->m_partner = p1;

		std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

		return true;
	}
};

int main() {
	auto lucy { std::make_shared<Person>("Lucy") }; // create a Person named "Lucy"
	auto ricky { std::make_shared<Person>("Ricky") }; // create a Person named "Ricky"

	Person::partnerUp(lucy, ricky); // Make "Lucy" point to "Ricky" and vice-versa

	return 0;
}
```

std::make_shared

- From C++ reference:

function template

std::make_shared

<memory>

```
template <class T, class... Args>
  shared_ptr<T> make_shared (Args&&... args);
```

Make shared_ptr

Allocates and constructs an object of type `T` passing *args* to its constructor, and returns an object of type `shared_ptr<T>` that owns and stores a pointer to it (with a use count of `1`).

So when declared, `lucy` is a `shared_ptr` to a Person named "Lucy" and `ricky` is a `shared_ptr` to a Person named "Ricky". Both have use count of 1.

```
int main()
{
  auto lucy { std::make_shared<Person>("Lucy") }; // create a Person named "Lucy"
  auto ricky { std::make_shared<Person>("Ricky") }; // create a Person named "Ricky"

  partnerUp(lucy, ricky); // Make "Lucy" point to "Ricky" and vice-versa

  return 0;
}
```

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr
#include <string>

class Person {
        std::string m_name;
        std::shared_ptr<Person> m_partner; // initially created empty

public:

        Person(const std::string &name): m_name(name) {
                std::cout << m_name << " created\n";
        }
        ~Person() {
                std::cout << m_name << " destroyed\n";
        }

        static bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {
                if (!p1 || !p2)
                        return false;

                p1->m_partner = p2;
                p2->m_partner = p1;

                std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

                return true;
        }
};

int main() {
        auto lucy { std::make_shared<Person>("Lucy") }; // create a Person named "Lucy"
        auto ricky { std::make_shared<Person>("Ricky") }; // create a Person named "Ricky"

        Person::partnerUp(lucy, ricky); // Make "Lucy" point to "Ricky" and vice-versa

        return 0;
}
```

```
(base) m1-mcs-dszajda:lecture_code_examples dszajda$ ./CircularReference
Lucy created
Ricky created
Lucy is now partnered with Ricky
```

Note two `Person` objects created dynamically but neither deleted!

So What Happened?

- We know that when declared, both `lucy` and `ricky` are pointers to the corresponding person objects
- When `partnerUp()` is called, the `m_partner` pointer for `lucy` points to `ricky`, and vice versa
  - So now `lucy` and `ricky.m_partner` both point to `lucy`
  - Same with `ricky` and `lucy.m_partner`
- This is OK. It's what `shared_ptr` is for (multiple pointers pointing to same object)

So What Happened?

- Fact: destructors are called in LIFO order at the end of a block
  - There is a good reason for this. See https://stackoverflow.com/questions/17238771/order-of-the-destructor-calls-at-the-end-of-block-program

So What Happened?

- So, at end of `main()`, destructor for `ricky` is called first. At that point, destructor for `ricky` checks if there are any other `shared_ptr` objects that co-own the `Person` "Ricky". There are (`lucy`'s `m_partner`), so destructor does not deallocate `Person` Ricky, because that would leave `Person` Lucy with a dangling pointer.
- At this point, there is one pointer to `Person` Ricky, and two to `Person` Lucy

So What Happened?

- Next the destructor for `lucy` is called. It does the same thing, seeing that there is another `shared_ptr` object that co-owns `Person` Lucy, so the destructor does not deallocate `Person` Lucy, because that would leave `Person` Ricky with a dangling pointer.
- The program then ends, but neither `Person` Ricky nor `Person` Lucy has been deallocated!

Circular References

- Our example had a *circular reference*: a series of references where each object references the next and the last object references the first
  - For previous example: Person Lucy refers to Person Ricky, which in turn references Lucy
  - Ex. Three objects A, B, C with A -> B -> C -> A
- Practical effect: Each object keeps the next object alive, with the last object keeping the first object alive
  - I'll let you work out why

`weak_ptr`

- This is where `weak_ptr` comes into play. It can observe and access the same objects as a `shared_ptr`, but it isn’t included in the reference count, so it does not prevent the objects from being deallocated

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person
{
        std::string m_name;
        std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

  Person(const std::string &name): m_name(name) {

    std::cout << m_name << " created\n";
  }
  ~Person() {
    std::cout << m_name << " destroyed\n";
  }

  static bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {

    if (!p1 || !p2)
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

    return true;
  }
};

int main() {
        auto lucy { std::make_shared<Person>("Lucy") };
        auto ricky { std::make_shared<Person>("Ricky") };

        Person::partnerUp(lucy, ricky);

        return 0;
}
```

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person
{
	std::string m_name;
	std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

  Person(const std::string &name): m_name(name) {

    std::cout << m_name << " created\n";
  }
  ~Person() {
    std::cout << m_name << " destroyed\n";
  }

  static bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {

    if (!p1 || !p2)
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

    return true;
  }
};

int main() {
	auto lucy { std::make_shared<Person>("Lucy") };
	auto ricky { std::make_shared<Person>("Ricky") };

	Person::partnerUp(lucy, ricky);

	return 0;
}
```

```
(base) m1-mcs-dszajda:lecture_code_examples dszajda$ ./weak_ptr
Lucy created
Ricky created
Lucy is now partnered with Ricky
Ricky destroyed
Lucy destroyed
```

`weak_ptr`

- Downside: you can't use `weak_ptr` directly
  - You need to convert it to a `shared_ptr` to use `->` and `*`
- This is done with the `lock()` function

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person {
  std::string m_name;
  std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

  Person(const std::string &name) : m_name(name) {
    std::cout << m_name << " created\n";
  }
  ~Person() {
    std::cout << m_name << " destroyed\n";
  }

  friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {
    if (!p1 || !p2)
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

    return true;
  }

  // use lock() to convert weak_ptr to shared_ptr
  const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); }
  const std::string& getName() const { return m_name; }
};

int main() {

  auto lucy { std::make_shared<Person>("Lucy") };
  auto ricky { std::make_shared<Person>("Ricky") };

  partnerUp(lucy, ricky);

  auto partner = ricky->getPartner(); // get shared_ptr to Ricky's partner
  std::cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';

  return 0;
}
```

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person {
  std::string m_name;
  std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

  Person(const std::string &name): m_name(name) {
    std::cout << m_name << " created\n";
  }
  ~Person() {
    std::cout << m_name << " destroyed\n";
  }

  friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2) {
    if (!p1 || !p2)
      return false;

    p1->m_partner = p2;
    p2->m_partner = p1;

    std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";

    return true;
  }
};

int main() {

  auto lucy { std::make_shared<Person>("Lucy") };
  auto ricky { std::make_shared<Person>("Ricky") };

  partnerUp(lucy, ricky);

  return 0;
}
```

```
(base) m1-mcs-dszajda:lecture_code_examples dszajda$ ./weak_to_shared
Lucy created
Ricky created
Lucy is now partnered with Ricky
Ricky's partner is: Lucy
Ricky destroyed
Lucy destroyed
```