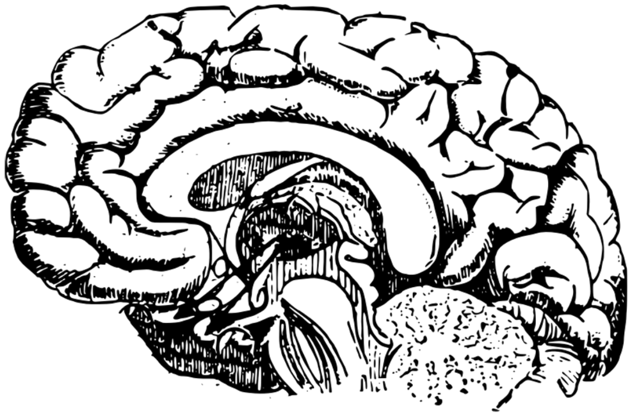
Recitation 9: Smart Pointers in C++



Goals

- C++ memory management.
- We all learned C++ in CSCI 1300/2270.
- Today, we will learn an important aspect of memory management in C++.
- Just *skimming the surface*, we will provide more references.
    - You have to go look them up on your own.
    - Use smart pointers in your next C++ project to learn more.

“Raw” Pointers

```
char * pointer_name = & variable_name;

int * my_int = new int(5);
```

“Raw” Pointers require manual management of memory

```
int * my_int = new int(5);
delete my_int;

int * my_int_arr = new int[10];
delete[] my_int_arr;
```

Problems with Raw Pointers: Manual memory management

```
Class A {

    int * ptr;

    A(): ptr(new int(30)) {} // Initialize ptr

    A(A const& a): ptr(a.ptr){} // copy constructor

    ~A(){ delete(ptr) } // delete the pointer.

}
```

See a problem?

Problems with Raw Pointers (Answer)

```
Class A {

    int * ptr;

    A(): ptr(new int(30)) {} // Initialize ptr

    A(A const& a): ptr(a.ptr){} // copy constructor

    ~A(){ delete(ptr) } // delete the pointer.

}
```

```
A * a = new A();
A * b = new A(*a); // Copy constructor
delete(b); // b -> ptr is deleted but so is a-> ptr
delete(a); // a -> ptr is deleted but double delete.
```

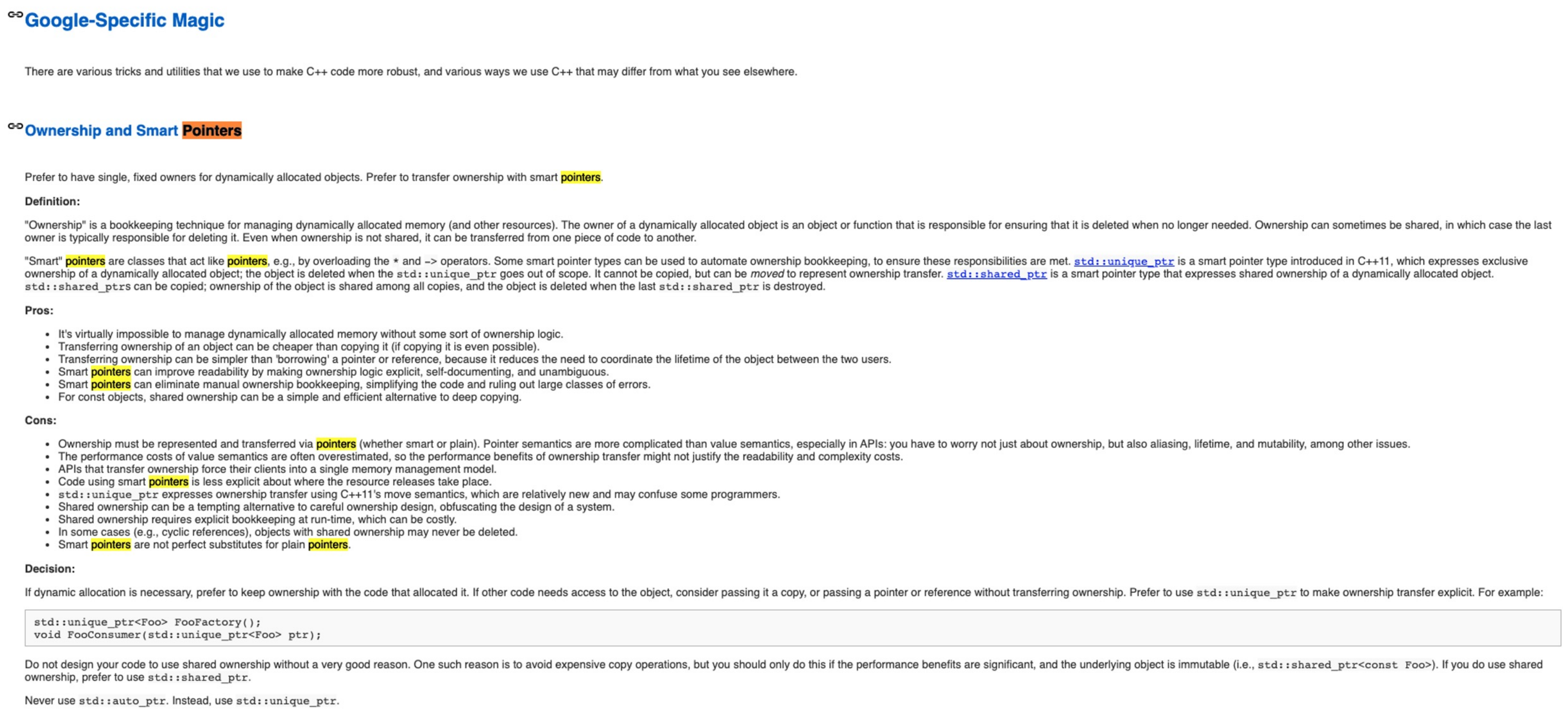
Problems with Raw Pointers # 2: Manual memory management

```
Class A {

    int * ptr;

    A(): ptr(new int(30)) {} // Initialize ptr

    A(A const& a): ptr(a.ptr){} // copy constructor

    // Look ma: no delete!!

}
```

See a problem?

Interested in being a professional C++ programmer???

Read a company's style guide: https://google.github.io/styleguide/cppguide.html



Google-Specific Magic

There are various tricks and utilities that we use to make C++ code more robust, and various ways we use C++ that may differ from what you see elsewhere.

Ownership and Smart Pointers

Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

Definition:

"Ownership" is a bookkeeping technique for managing dynamically allocated memory (and other resources). The owner of a dynamically allocated object is an object or function that is responsible for ensuring that it is deleted when no longer needed. Ownership can sometimes be shared, in which case the last owner is typically responsible for deleting it. Even when ownership is not shared, it can be transferred from one piece of code to another.

"Smart" pointers are classes that act like pointers, e.g., by overloading the `*` and `->` operators. Some smart pointer types can be used to automate ownership bookkeeping, to ensure these responsibilities are met. `std::unique_ptr` is a smart pointer type introduced in C++11, which expresses exclusive ownership of a dynamically allocated object; the object is deleted when the `std::unique_ptr` goes out of scope. It cannot be copied, but can be *moved* to represent ownership transfer. `std::shared_ptr` is a smart pointer type that expresses shared ownership of a dynamically allocated object. `std::shared_ptr`s can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed.

Pros:

- It's virtually impossible to manage dynamically allocated memory without some sort of ownership logic.
- Transferring ownership of an object can be cheaper than copying it (if copying it is even possible).
- Transferring ownership can be simpler than 'borrowing' a pointer or reference, because it reduces the need to coordinate the lifetime of the object between the two users.
- Smart pointers can improve readability by making ownership logic explicit, self-documenting, and unambiguous.
- Smart pointers can eliminate manual ownership bookkeeping, simplifying the code and ruling out large classes of errors.
- For const objects, shared ownership can be a simple and efficient alternative to deep copying.

Cons:

- Ownership must be represented and transferred via pointers (whether smart or plain). Pointer semantics are more complicated than value semantics, especially in APIs: you have to worry not just about ownership, but also aliasing, lifetime, and mutability, among other issues.
- The performance costs of value semantics are often overestimated, so the performance benefits of ownership transfer might not justify the readability and complexity costs.
- APIs that transfer ownership force their clients into a single memory management model.
- Code using smart pointers is less explicit about where the resource releases take place.
- `std::unique_ptr` expresses ownership transfer using C++11's move semantics, which are relatively new and may confuse some programmers.
- Shared ownership can be a tempting alternative to careful ownership design, obfuscating the design of a system.
- Shared ownership requires explicit bookkeeping at run-time, which can be costly.
- In some cases (e.g., cyclic references), objects with shared ownership may never be deleted.
- Smart pointers are not perfect substitutes for plain pointers.

Decision:

If dynamic allocation is necessary, prefer to keep ownership with the code that allocated it. If other code needs access to the object, consider passing it a copy, or passing a pointer or reference without transferring ownership. Prefer to use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Do not design your code to use shared ownership without a very good reason. One such reason is to avoid expensive copy operations, but you should only do this if the performance benefits are significant, and the underlying object is immutable (i.e., `std::shared_ptr<const Foo>`). If you do use shared ownership, prefer to use `std::shared_ptr`.

Never use `std::auto_ptr`. Instead, use `std::unique_ptr`.

Solution # 1 : Use simply destructible objects

```
Class A {

    std::vector<int> vec; // Use a STL vector instead of array

    A(): vec(30, 0) {} // Initialize ptr to size 30 with 0s

    A(A const& a): vec(a.vec){} // copy constructor

    // No destructor needed: vec is automatically destructed

}
```

Full copy (deep copy) - Expensive!

Solution # 2: Smart Pointers

- std::unique_ptr
- Std::shared_ptr
- std::weak_ptr

Smart Pointer : Example 1

```
  A * raw_ptr = new A()

 // make it a shared

  std::shared_ptr<A> pt

  ptr -> call_function(

  (*ptr).call_function();

//No need to delete raw_ptr or ptr
```

Creating raw_ptr is not a good idea.

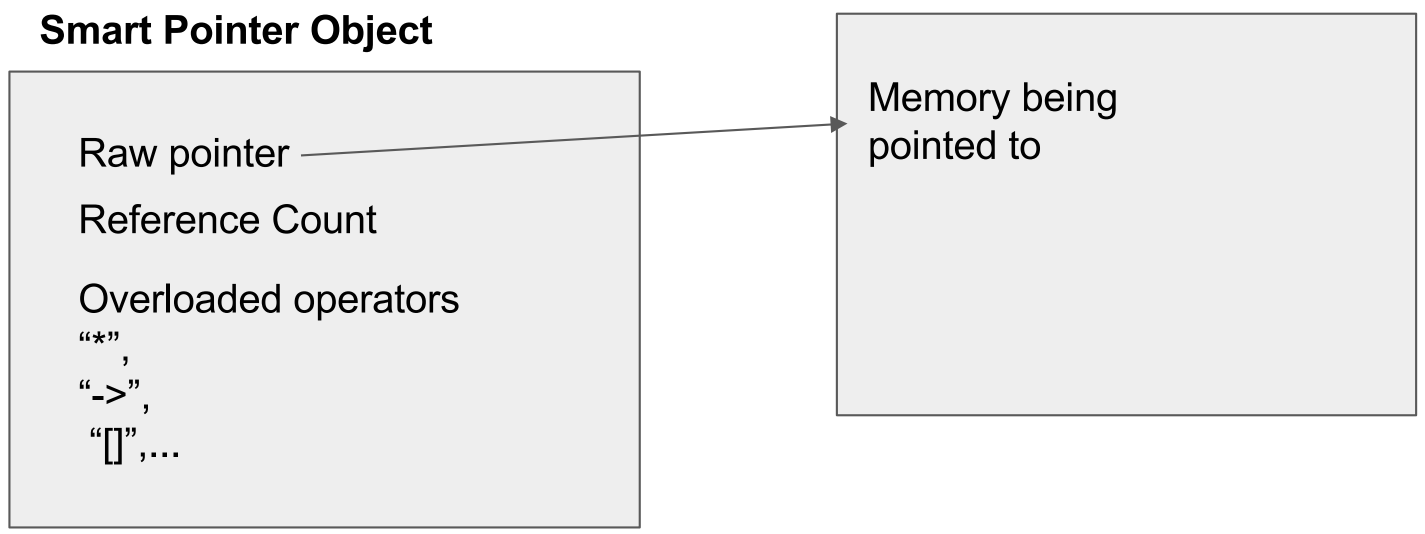
Use std::make_shared to directly call the constructor and make a shared pointer

```
std::shared_ptr<A> ptr = std::make_shared<A>();
```

Smart Pointer: Example 2

```
class A {

    shared_ptr<int> ptr; // instead of int *, use a shared_ptr<int>

    A(): {

        this-> ptr = make_shared<int>(30); // calls new int(30) internally!

    }

    A(A const& a): ptr(a.ptr){} // copy constructor

    // No destructor needed: ptr is automatically destructed

}
```

What is a smart pointer?



Smart pointers wrap around the “raw pointer”.
Takes care of sharing and will automatically call delete when it goes out of scope.
User does not work with raw pointer once a smart pointer is created.

Smart Pointers Come in Many Types

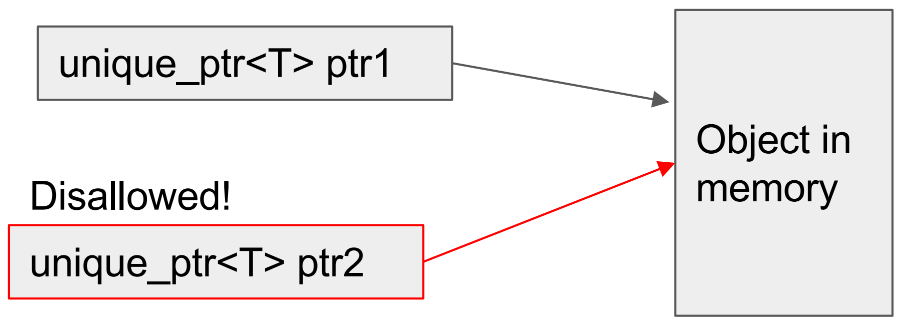
- Shared smart pointer: “`shared_ptr<T>`”
  - Use it when you have many pointers to the same memory location.
- Unique smart pointer: “`unique_ptr<T>`”
  - Use it when you guarantee that only one pointer will point to memory at a time.
  - This supports an important programming paradigm of single ownership.
- Weak pointer: “`weak_ptr<T>`”
  - weak_ptr will not delete the underlying memory when they go out of scope.
  - Its relation with shared pointers is similar to the relation between peek() and pop() operations for Stack, if we imagine stack elements as memory cells.

Smart Pointers

They can enforce ownership, eg., with unique_ptr.

Unique Pointers

Only one reference is allowed at any time.



```
int * raw_ptr = new …
std::unique_ptr<int> ptr1 = std::make_unique(raw_ptr);
std::unique_ptr<int> ptr2 = ptr1; // NOT ALLOWED TO COPY a unique pointer.
// use std::move to move a unique pointer from one onto another
std::unique_ptr<int> ptr2 = std::move(ptr1); // Right way -- ptr1 is now invalid.
```

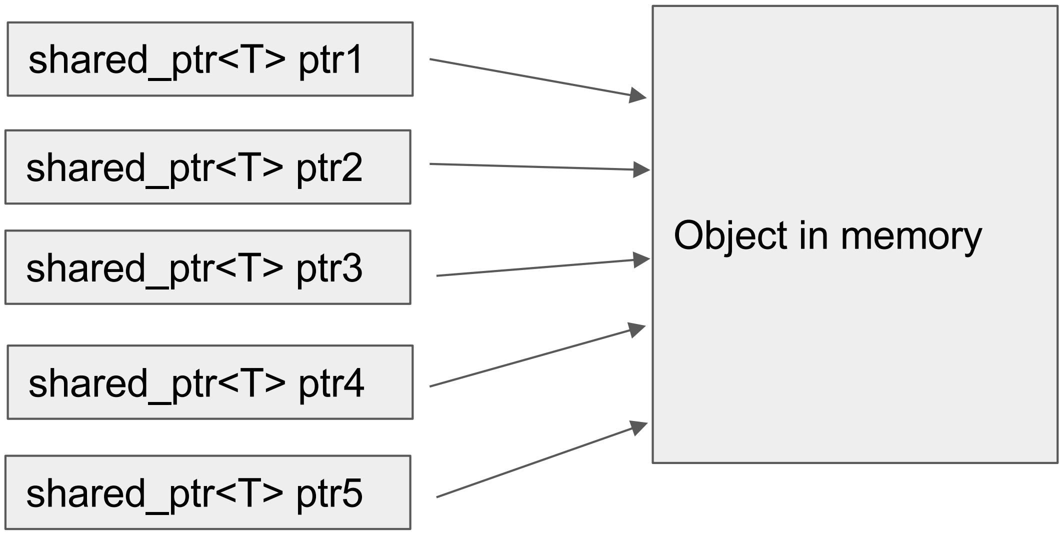
Smart Pointers

Smart pointers manage memory with reference counting.

When a shared_ptr goes out of scope, the reference count for the object it points to is decremented. Once that count reaches zero, the object is deleted.

Shared Pointers

Shared pointers to same object share
a reference count.



When last reference goes out of scope, the object is automatically deleted from memory.

Shared Pointers

```
class A {
   A (int x, char y, …)
   …
}

//Not advisable
A * raw_ptr = new A( 20, ‘c’,...);
shared_ptr<A> a_shared (raw_ptr);

//Advisable
shared_ptr<A> a_shared = make_shared(20, ‘c’, …);
```

Problems with shared pointers

Sometimes, programmers may have an unexpected shared pointer reference to a large object.

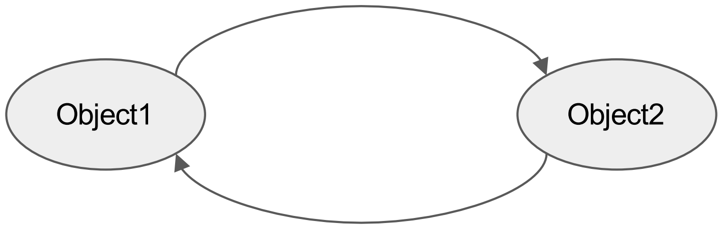
- Eg., a “statistics” object has a shared_ptr reference to a large file contents object.
- The object remains in the memory although we could have freed it up and the statistics object is not really necessary/reading that object.

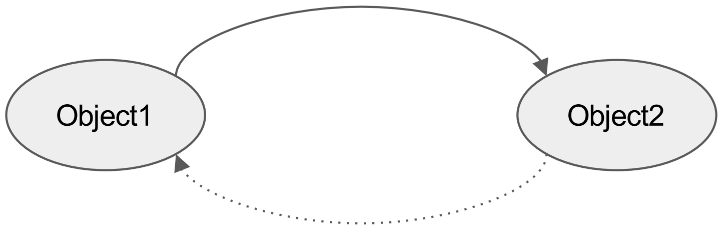
Programmer discipline is needed in tracking which objects hold shared_ptr references to other objects.

Cyclic references are a problem and strictly discouraged.

Problem with Shared_ptr: Cyclic References

Shared pointers may create cyclic references





Weak pointers break cyclic references

```
std::weak_ptr<A> partner;

Object2 -> partner = &Object1;

Const std::weak_ptr<A> get_partner() const { return partner;}
                                                              OR ??
Const std::shared_ptr<A> get_partner() const { return partner.lock();}
```

Weak Pointers

When a weak_ptr goes out of scope, nothing happens. If the object pointed to by a weak_ptr has not been deleted, it is possible to promote it to a shared_pointer using std::weak_ptr<T>::lock.

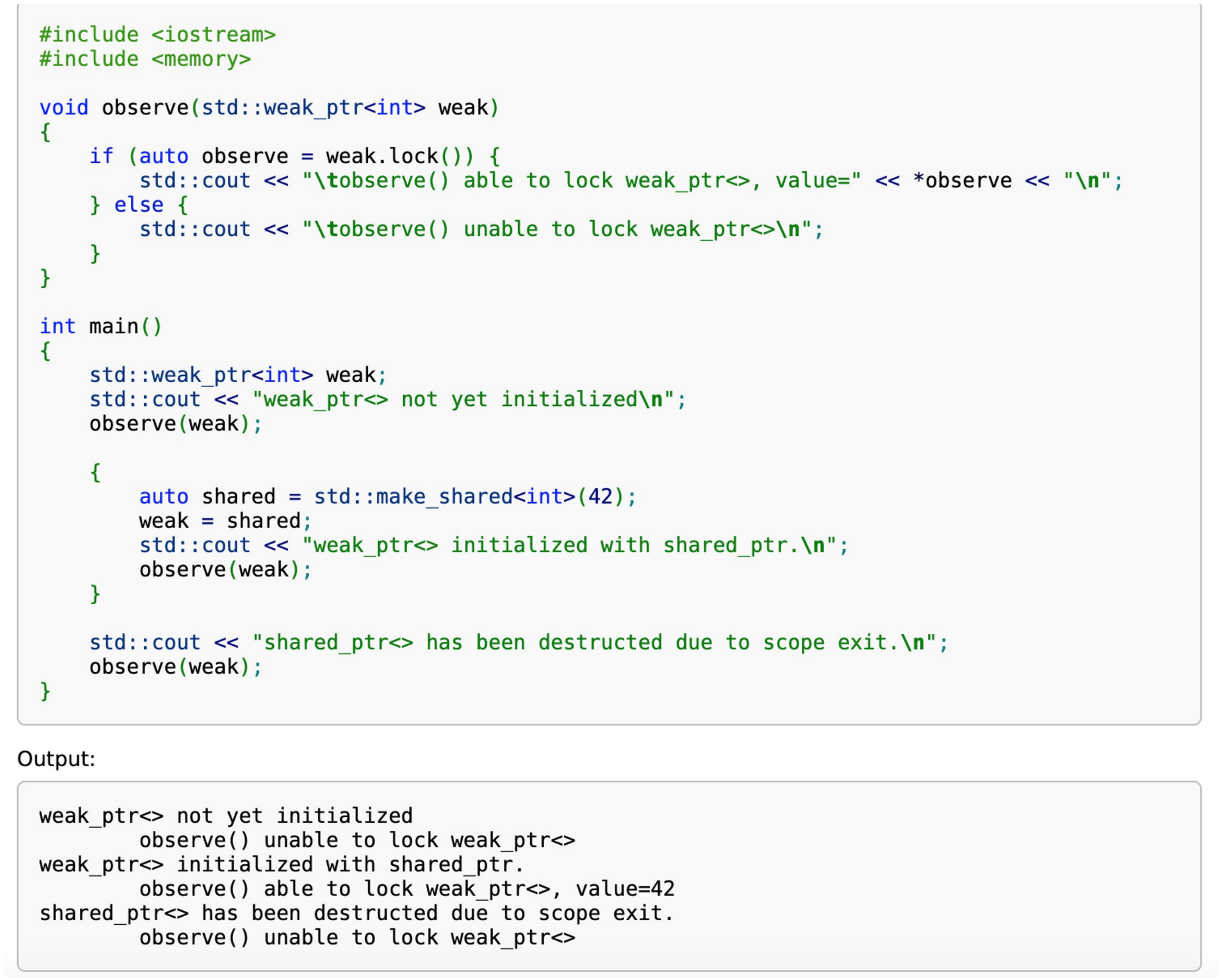
Weak Pointers

```
auto sp = std::make_shared<int>(42)
auto up = std::make_unique<int>(42)

auto up_vec =
std::make_unique<std::vector<int>>();
```

Why no make_weak?

Example

https://en.cppreference.com/w/cpp/memory/weak_ptr/lock



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

void observe(std::weak_ptr<int> weak)
{
    if (auto observe = weak.lock()) {
        std::cout << "\tobserve() able to lock weak_ptr<>, value=" << *observe << "\n";
    } else {
        std::cout << "\tobserve() unable to lock weak_ptr<>\n";
    }
}

int main()
{
    std::weak_ptr<int> weak;
    std::cout << "weak_ptr<> not yet initialized\n";
    observe(weak);

    {
        auto shared = std::make_shared<int>(42);
        weak = shared;
        std::cout << "weak_ptr<> initialized with shared_ptr.\n";
        observe(weak);
    }

    std::cout << "shared_ptr<> has been destructed due to scope exit.\n";
    observe(weak);
}
```

Output:

```
weak_ptr<> not yet initialized
        observe() unable to lock weak_ptr<>
weak_ptr<> initialized with shared_ptr.
        observe() able to lock weak_ptr<>, value=42
shared_ptr<> has been destructed due to scope exit.
        observe() unable to lock weak_ptr<>
```

Further Readings

- https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp
- https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization