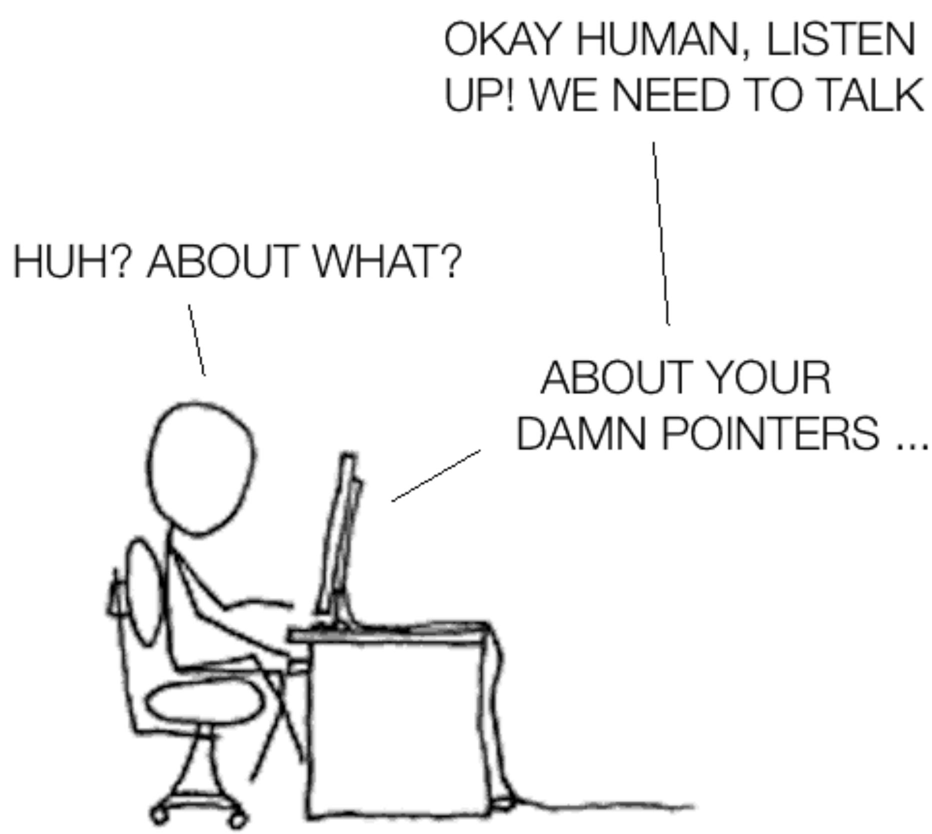
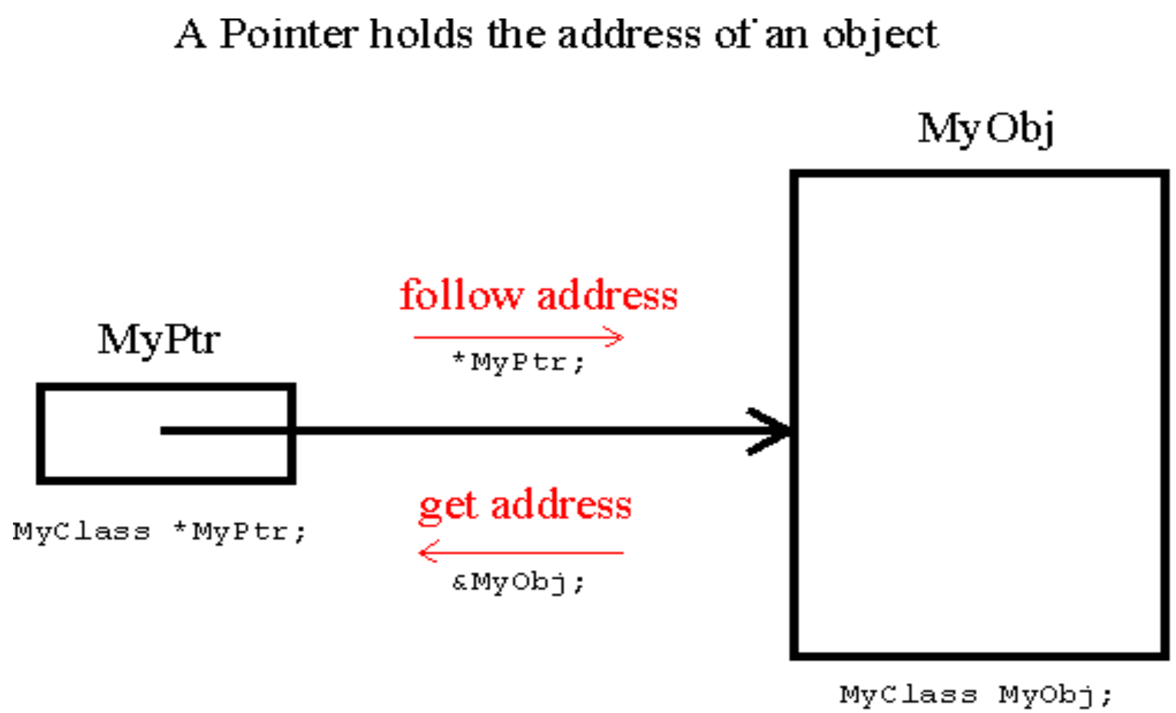
Modern C++ smart pointers in C++17, 20 and beyond

(prefer scoped objects, do not heap allocate unnecessarily)

Pointers, dumb and smart



What is a (raw, dynamic) pointer?



The variable that stores the address of a memory block is what in C++ is called a pointer. What we are interested in is dynamic **allocation and deallocation using new and delete on the heap** using various **lifetime** and **ownership** semantics

Raw pointers and its problems Part I

Widget *w;

- You don’t know whether the raw pointer points to an object or to an array of objects.
- You don’t know the ownership, whether you need to destruct what it points to, i.e. if the pointer owns the thing it points to. So you have to destruct the object when you are done?
- If you know you need to call delete you don’t know how? Use delete or delete[] or something else.

Raw pointers and its problems Part II

- Assume you know that you need to use **delete**, so you know how to destruct it, it's difficult to ensure that you destruct exactly once along every path in your code (including exceptions). Missing a path leads to resource leaks, and doing it more than once leads to undefined behavior (or erase your hard drive… :D)

- There is no way to tell if the pointer is a dangling pointer or i.e points to a memory that no longer holds the object the pointer is pointing to?

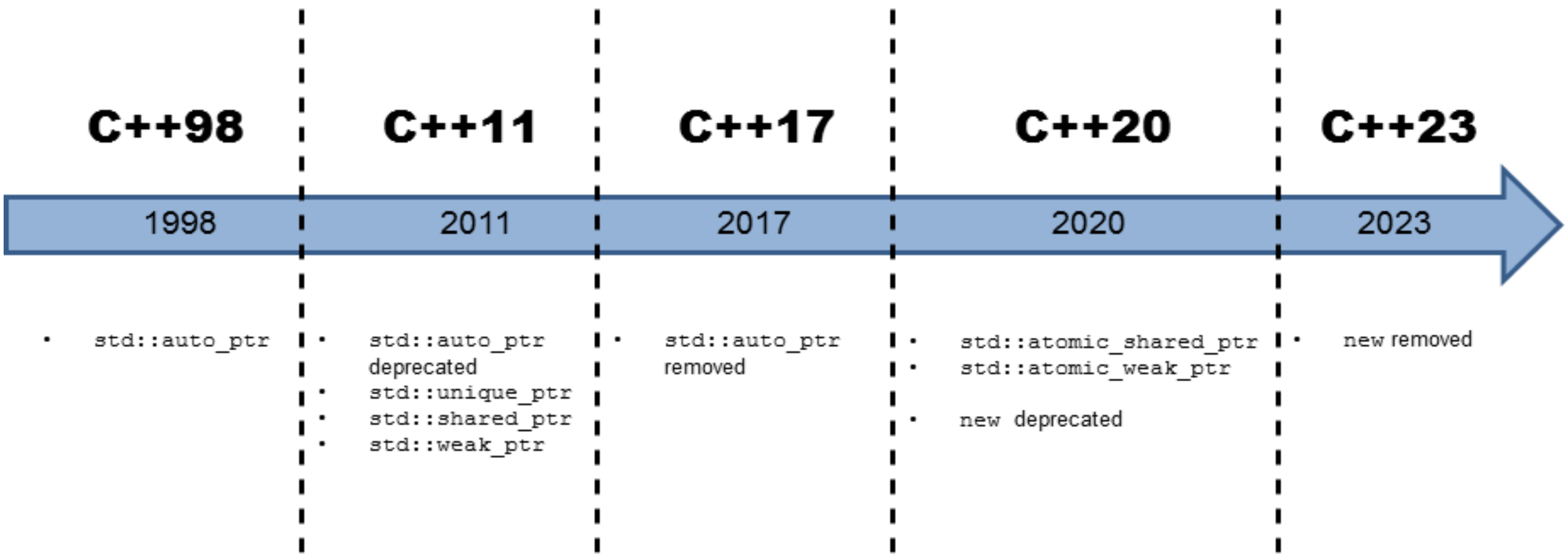
Raw pointers and its problems Part III

- Smart pointers could solve most of the issues most of the time. No holy grail. Smart pointers are wrappers around raw pointers that act much like as the raw pointers without the pitfalls.

- Since C++98 there are std::auto_ptr (deprecated and removed!), **std::unique_ptr, std::shared_ptr, std::weak_ptr** and more to come beyond c++17

- **std::auto_ptr** has serious flaws hence it was deprecated in c++11.

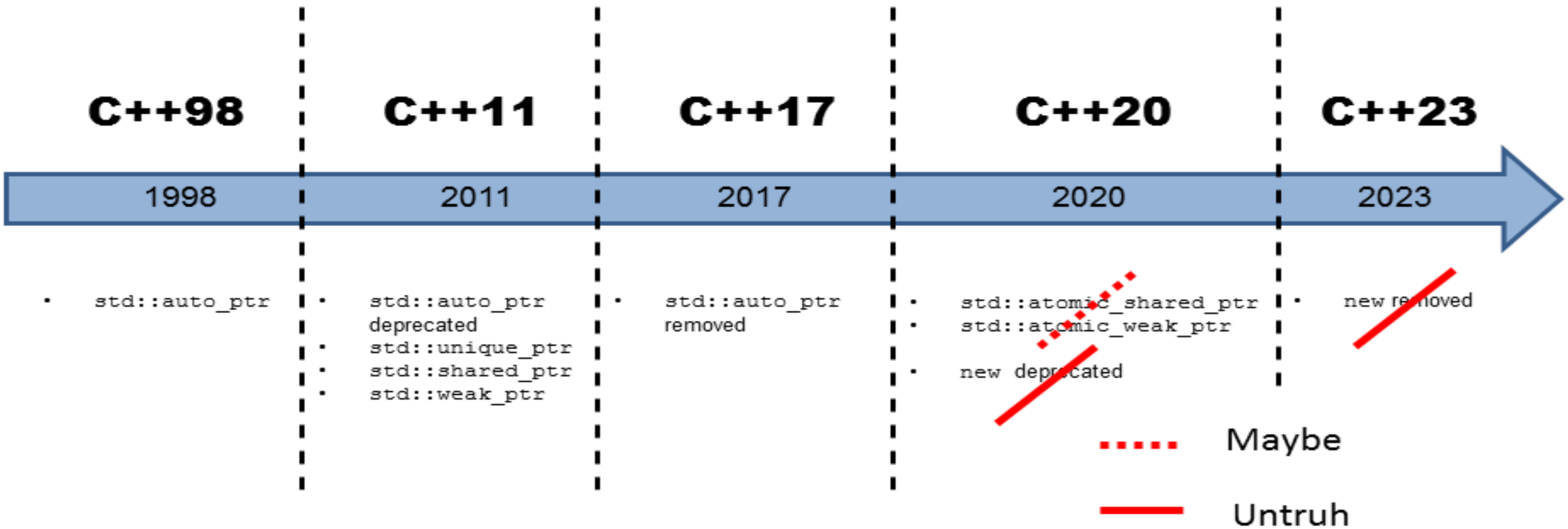
C++ (r)evolution of smart pointers



Source: http://www.modernescpp.com/index.php/no-new-new

The truth about “raw pointers removed from c++”

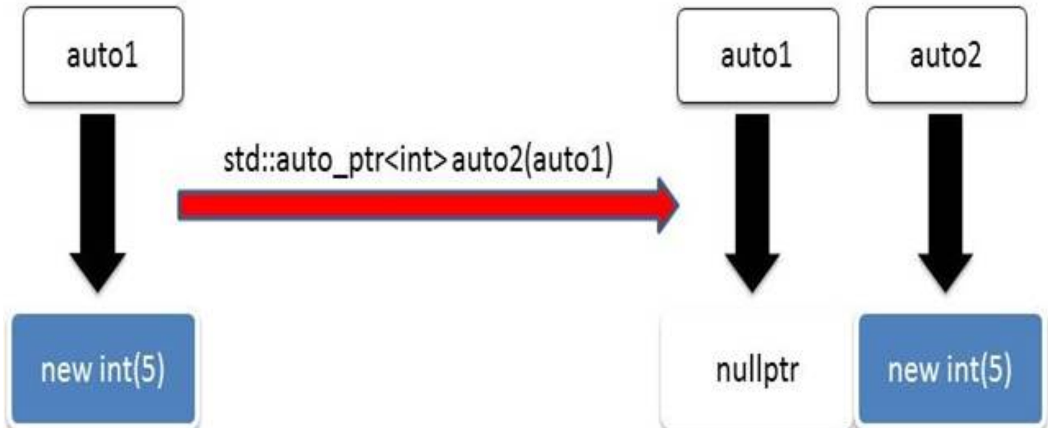
Pointers won’t be deprecated in c++20 or removed in c++23.
It was an April Fool’s joke.



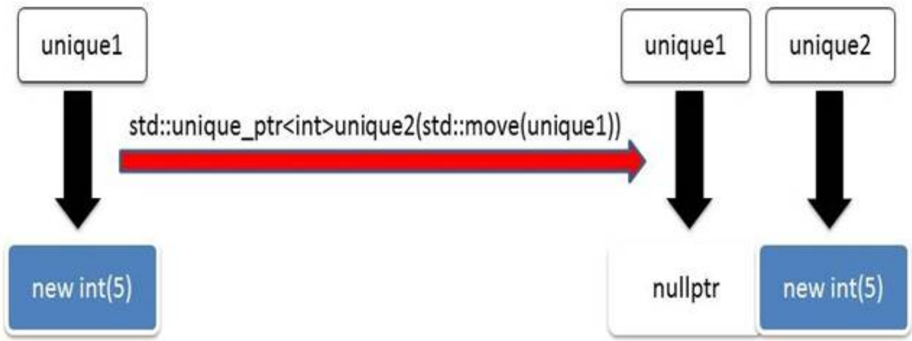
std::auto_ptr deprecated and removed

- **std::auto_ptr** has been deprecated in c++11 and has been removed in c++17

- **std::auto_ptr** main flaw is that what looks like a copy operation is actually a move operation, which lead to a series of serious bugs. Use **unique_ptr** instead.

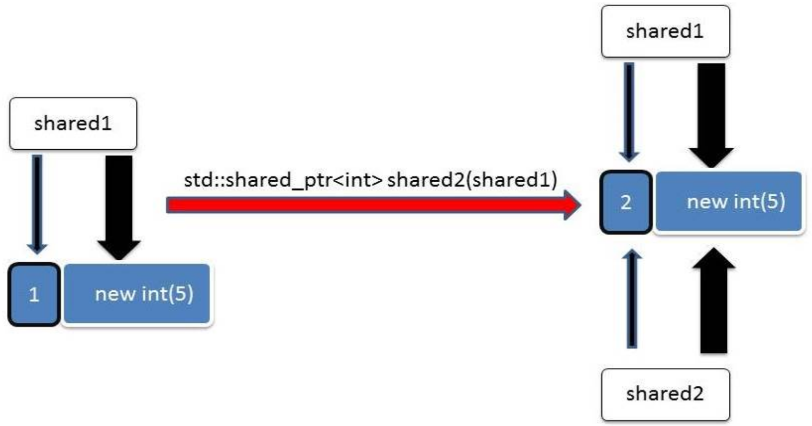


std::unique_ptr



- **Exclusive ownership**. Assignment, transfers (move) ownership.
- **Cannot copy** but move a unique_ptr
- Comes in two forms **unique_ptr<T> and unique_ptr<T[]>**
- Should be considered as the go to smart pointer, usually.
- Should be the same size as raw pointers and execute at the same speed. Will show some numbers later.
- Common use case is a factory function.

std::shared_ptr



- Use for **shared ownership resource management**. If you copy or assign a std::shared_ptr the internal reference counter will be increased.
- std::shared_ptr is epxensive, twice the size of a raw pointer. Has 2 raw pointers, one to the resource, one to the reference count.
- Memory to the reference count must be dynamically allocated.
- Increment and decrement operations of the reference count must be atomic, generally atomic operations are slower.

Smart pointer core guidelines part I

- R.20: Use **unique_ptr** or **shared_ptr** to represent ownership
- R.21: Prefer **unique_ptr** over **shared_ptr** unless you need to share ownership
- R.22: Use **make_shared()** to make **shared_ptrs**
- R.23: Use **make_unique()** to make **unique_ptrs**
- R.24: Use **weak_ptr** to break cycles of **shared_ptrs**
- R.30: Take smart pointers as parameters only to explicitly express lifetime semantics
- R.31: If you have non-std smart pointers, follow the basic pattern from std

Smart pointer core guidelines part II

- R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget
- R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget
- R.34: Take a shared_ptr<widget> parameter to express that a function is part owner
- R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer
- R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object ???
- R.37: Do not pass a pointer or reference obtained from an aliased smart pointer

Ownership semantic, who is the owner? Rules to resource management.

- **Local objects.** The c++ runtime as the owner automatically manages the lifetime of these resources.
- **References.** I'm not the owner. I only borrowed a non empty resource that I can use.
- **Raw pointers.** I'm not the owner. I only borrowed a resource that can be empty. I must not delete.
- **std::unique_ptr.** I'm the exclusive owner the resource. I may explicitly release it.
- **std::shared_ptr.** I share the ownership with other shared_ptr. I may explicitly release my shared ownership.
- **std::weak_ptr.** I'm not the owner of the resource but I may become temporarily the shared owner the resource by using the method std::weak_ptr.lock()

Performance and memory overhead concerns

Performance analysis of smart pointers

```cpp
// all.cpp

#include <chrono>
#include <iostream>

static const long long numInt= 100000000;

int main(){

  auto start = std::chrono::system_clock::now();

  for ( long long i=0 ; i < numInt; ++i){
    int* tmp(new int(i));
    delete tmp;
    // std::shared_ptr<int> tmp(new int(i));
    // std::shared_ptr<int> tmp(std::make_shared<int>(i));
    // std::unique_ptr<int> tmp(new int(i));
    // std::unique_ptr<int> tmp(std::make_unique<int>(i));
  }

  std::chrono::duration<double> dur= std::chrono::system_clock::now() - start;
  std::cout << "time native: " << dur.count() << " seconds" << std::endl;

}
```

Compiler	Optimization	new	std::shared_ptr	std::make_shared	std::unique_ptr	std::make_unique
GCC	no	3.03	13.48	30.47	8.74	9.09
GCC	yes	3.03	6.42	3.24	3.07	3.04
cl.exe	no	8.79	25.17	18.75	11.94	13.00
cl.exe	yes	7.42	17.29	9.40	7.58	7.68

Source: http://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer

Memory overhead

- **std::unique_ptr** needs by default no additional memory. Is as large as the underlying raw pointer.

- **std::shared_ptr** share a resource. They internally use a reference counter. The reference counter is usually another pointer pointing to the controller.

Atomic smart pointers or smart pointers in concurrency

- C++20 will have std::atomic_shared_ptr and std::atomic_weak_ptr
- std::shared_ptr is half thread safe. The access to the control block is thread-safe but to the resource it is not.

```
std::shared_ptr<int> ptr = std::make_shared<int>(2018);

for (auto i= 0; i<10; i++){
   std::thread([&ptr]{                                 (1)
     ptr = std::make_shared<int>(2019);                 (2)
   }).detach();
}
```

(2) Is a race condition on the resource and the code is undefined behavior

Errors and gotchas to watch out for

Errors and gotchas #1

Too many shared pointers

- Overuse of shared_ptr can be a problem.
- **Subtle bugs.** For example the resource is shared out through a pointer and some other part of the code inadvertently modifies the resource.
- **Unnecessary resource usage.** Even if the pointer does not modify the shared resource, it might use the memory far longer than necessary even after the original shared_ptr goes out of scope. Creating a shared_ptr is more resource intensive than creating a unique_ptr.

Advice − By default use unique_ptr

Errors and gotchas #2

Resources shared by shared_ptr are not thread safe

std::shared_ptr allows to share the resource using multiple shared pointers and those can be used from multiple threads. It's a common mistake to think that wrapping an object into a shred_ptr is inherently thread safe. You still need to use synchronization primitives around the resource to which the pointer points to. The control block is thread safe while the resource is not. (C++20 will have atomic smart pointers)

Advice − think about ownership, if you do not plan to share the resource use **unique_ptr**. Period.

Errors and gotchas #3

Using auto_ptr

The **auto_ptr** has been deprecated, do not use it. The transfer of ownership executed by the copy constructor when the pointer is passed by value can cause crashes when the original pointer gets dereferenced again. When you copy you essentially move leading to unexpected results.

Advice – **unique_ptr** does what auto_ptr was intended to do, but is much better. unique_ptr has move semantics. Use unique_ptr instead.

Errors and gotchas #4
Not using make_shared to initialize shared_ptr!

Performance problem! Basically when you create an object using **new**, and then create a **shared_ptr**, there are **two dynamic allocations** that happen. One for the object itself from the new and one for the managed object created by the shared_ptr constructor.

(Note: if you do need to allocate an array, alternatively you can do:

```
auto buffer = std::make_shared<std::array<char, 64>>();
```

)

Advice − use make_shared to create shared pointers, also use make_unique to create unique pointers

Errors and gotchas #5

Failing to assign a raw pointer to a shared_ptr as soon as it is created

A raw pointer should be assigned to a shared_ptr as soon as it is created. The raw pointer should never be used again.

```
int main()
{
      Parrot* georgeParrot = new Parrot("George");
      shared_ptr pParrot(georgeParrot);
      cout << pParrot.use_count() << endl; // ref-count is 1
      shared_ptr pParrot2(georgeParrot);
      cout << pParrot2.use_count() << endl; // ref-count is 1
      return 0;
}
```

The above code will cause **ACCESS VIOLATION** and crash the program. When the first shared_ptr goes out of scope georgeParrot is destructed. Then when the second goes out of scope it is destructed the second time.

Advice – Use *std::make_shared,* if it is not possible then create the shared pointer and the object on one line:

```
shared_ptr pParrott(new georgeParrot(“George”));
```

Errors and gotchas #6
Deleting the raw pointer used by the shared_ptr

You can get a handle to the raw pointer of a **shared_ptr** using the **shared_ptr.get()** call. This is best avoided. Consider the following code:

```
void StartJob()
{
     shared_ptr pParrot(new Parrot("George"));
     Parrot* georgeParrot = pParrot.get(); // returns the raw pointer
     delete georgeParrot;  // georgeParrot is gone
}
```

The result of the above code is **ACCESS VIOLATION**. Why? Once you get the raw pointer from the shared pointer you delete it. Once the function ends and the shared pointer goes out of scope, it attempts to destruct the shared pointer.

Advice – Do not do this

Errors and gotchas #7
Not using a custom deleter when using an array of pointers with a shared_ptr (pre c++17)

c++17 and after shared_ptr can be used to manage a dynamically allocated array.

```
std::shared_ptr<int[]> sp(new int[10])
```

Pre c++17 shared_ptr could not be used to manage a dynamic array. By default shared_ptr will call delete on the managed object when no more references remain to it. When you allocate using new[] you need to call delete[]. **So you must supply a custom deleter before c++17.** For example:

```
std::shared_ptr<int> sp(new int[10], array_deleter<int>());
```

Alternatively you can just go with std::array<>

Errors and gotchas #8
Not avoiding cyclic references when using shared pointers

The problem. In many situations when a class has a shared_ptr member variable you can get into cyclical references. When two shared_ptrs are holding references to each other they will never be destructed, because the reference count will never go to 0, hence we have a memory leak. Double-linked structures end up with circular references.

```
template<class T>
struct Node {
	T value;
	shared_ptr<Node<T>> parent;
	shared_ptr<Node<T>> left;
	shared_ptr<Node<T>> right;
};
```

If we remove a Node, there's a cyclic reference to it. It will never be deleted because the reference count will be never zero.

The solution. To solve this, you should use std::weak_ptr<T>:

```
template<class T>
struct Node {
	T value;
	weak_ptr<Node<T>> parent;
	shared_ptr<Node<T>> left;
	shared_ptr<Node<T>> right;
};
```

Now things will work correctly and removing a node will not leave stuck references to the parent node. Side note: It makes walking the tree slightly more complicated. You need to lock the parent Node and you have a reasonable guarantee that it won't disappear while you are working on it.

Errors and gotchas #9
Not deleting a raw pointer returned by unique_ptr.release()

The problem: The release method does not destruct the object managed by the unique_ptr, but the unique_ptr object is released from the responsibility of deleting the object. Someone else must delete this object manually.

The following code will result in a memory leak.

```
int main()
{
    unique_ptr<Parrot> myParrot = make_unique<Parrot>("George Parrot");
    Parrot* rawPtr = myParrot.release();
    return 0;
}
```

The solution: Anytime you call Release() on a unique_ptr, remember to delete the raw pointer. Use **reset()** if you’re intent is to change the object managed by unique_ptr but you do not want to release it.

Errors and gotchas #10
Not using an expiry check when calling weak_ptr.lock()

- Before using weak_ptr you need to acquire the weak_ptr by calling a **lock()** method.
- It essentially **upgrades the weak_ptr to a shared_ptr** that you can use. But if the shared_ptr that the weak_ptr points to is no longer valid, the weak_ptr is empty. Calling any method will cause an **ACCESS VIOLATION**.
- **Before calling lock() on the weak_ptr you should call expired()** to know whether it is still valid or not empty.

```
// Example
if (!pMatrix->agentSmith.expired()) {
     cout << pMatrix->agentSmith.lock()->talk() << endl;
}
```

Passing smart pointers to functions – part I

- R.32. Take a **unique_ptr<Widget>** parameter to express that a function assumes exclusive ownership of a Widget.
- R.33. Take a **unique_ptr<Widget>&** parameter to express that a function reseats the Widget.
- R.34. Take a **shared_ptr<Widget>** to express that a function is part owner.
- R.35. Take a **shared_ptr<Widget>&** parameter to express that a function might reseat the shared pointer.
- R.36. Take a **const shared_ptr<Widget>&** parameter to express that it might retain a reference count to the object ???
- R.37. Do not pass a pointer or reference obtained from an aliased smart pointer.

Passing smart pointers to functions – R.32

R.32. Take a **unique_ptr<Widget>** parameter to express that a function assumes exclusive ownership of a Widget.

Below call (2) breaks because you cannot copy a std::unique_ptr only move.

```
void sink(unique_ptr<Matrix> uniqPtr){
    // do something with uniqPtr
}

int main(){
    auto uniqPtr = make_unique<Matrix>(1998);

    sink(std::move(uniqPtr));      // (1)
    sink(uniqPtr);                 // (2) ERROR
}
```

Passing smart pointers to functions – R.33

- R.33. Take a **unique_ptr<Widget>&** parameter to express that a function reseats the Widget.

```
void sink(std::unique_ptr<Matrix>& uniqPtr){
       uniquePtr.reset(new Matrix(2001)); // (0)
//
}

int main(){
    auto uniqPtr = std::make_unique<Matrix>(1998);

    sink(std::move(uniqPtr));      // (1) ERROR
    sink(uniqPtr);                 // (2)
}
```

Call (1) fails because you cannot bind an rvalue to a non-const lvalue reference. Call (0) will reseat, or create a new Matrix and destruct the old one.

Passing smart pointers to functions – R.34 and R.35

- R.34. Take a **shared_ptr<Widget>** to express that a function is part owner.
- R.35. Take a **shared_ptr<Widget>&** parameter to express that a function might reseat the shared pointer.

Three function signatures

- **`void share(std::shared_ptr<Matrix> shaMat);`**
  Shared owner, increase the ref count at the beginning and decrease the ref count at the end. The Matrix will stay alive as long as I use it.

- **`void reseat(std::shard_ptr<Matrix>& shadMat);`**
  Not a shared owner as the ref count stays the same. Resource can be reseated, or not. No guarantee that Matrix will stay alive during execution of the function.

- **`void mayShare(const std::shared_ptr<Matrix>& shaMat);`**
  Only borrow the resource. Can't extend the lifetime nor reseat the resource. It would be better to use a reference like (Matrix&)

R.37. Do not pass a pointer or reference obtained from an aliased smart pointer.

```
void oldFunc(Matrix* mat){
  // do something with mat
}

void shared(std::shared_ptr<Matrix>& shaPtr){       // (2) remove reference solution 1
   auto keepAlive = shaPtr; // solution 2
   oldFunc(*shaPtr);                                 // (3)
   // do something with shaPtr
 }

auto globShared = std::make_shared<Matrix>(2009);   // (1) smart pointer
shared(globShared);
```

globShared (1) is a shared pointer. The function shared takes it by reference therefore (2) so it does not increase the ref count. Problem is at (3) oldFunc accepts a pointer and oldFunc has no guarantee that the Matrix will stay alive during the execution, oldFunc only borrows the Matrix.

The solution is simple, either pass the std::shared_ptr by copy to the function shared or make a copy of shaPtr in the function shared

returning smart pointers from functions

- The rule is extremely simple. Return by value whether it is a std::unique_ptr or std::shared_ptr.

  The rest will be taken care of RVO (possibly)

smart pointers and concurrency

- Atomic smart pointers in c++20.
- Currently the shared pointers are prone to data races and thus, leading to undefined behavior. std::shared_ptr and std::weak_ptr guarantee that the incrementing and decrementing of the reference counter is an atomic operation and the resource is deleted only once, but no guarantee for atomic resource access, which leads to subtle bugs.
- std::atomic_shared_ptr<T> and std::atomic_weak_ptr<T> will solve these issues.

shared pointers and arrays

- Before c++17 only **unique_ptr** was able to handle arrays out of the box (without the need to define a custom deleter). In c++ 17 it's possible with shared_ptr.

```
std::shared_ptr<int []> ptr(new int[10]);
```

- **make_shared** doesn't support arrays in c++17. But this will be fixed in c++20. (or just use std::array ☺)

- It is also best avoided to use raw arrays, it is better to use standard containers.

smart pointers and vector

- **std::vector<std::unique_ptr<T>>** unique_ptr has no copy constructor, can only move.

```
vec.push_back(std::make_unique<Widget>());
vec.push_back(std::move(pWidget));
```

- **std::vector<std::shared_ptr<T>>** shared_ptr does have a copy constructor but if you want to avoid 2 copies or want to avoid the increase of reference counts, move them.

```
vec.push_back(std::make_shared<Widget>());
vec.push_back(std::move(pSharedWidget));
```

std::shared_ptr from this

If you derive your class from std::enable_shared_from_this using the CRP or Curriously Recurring Pattern you can create an object that returns a shared_ptr from this. Also why? The “obvious” technique of just returning shared_ptr<T>(this) is broken, because this winds up creating multiple distinct shared_ptr objects with separate reference counts.

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

class ShareMe: public std::enable_shared_from_this<ShareMe>{
public:
	std::shared_ptr<ShareMe> getShared(){
		return shared_from_this();
	}
};

int main(){
	shared_ptr<ShareMe> shareMe(new ShareMe);
	shared_ptr<ShareMe> shareMe1 = shareMe->getShared();

	{
		auto shareMe2(shareMe1);
		std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl; // 3
	}
	std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl; // 2

	shareMe1.reset();
	std::cout << "shareMe.use_count(): "  << shareMe.use_count() << std::endl; // 1
}
```

Thank you