COMP6771
Advanced C++ Programming

Week 5.2
Smart Pointers

In this lecture

Why?

- Managing unnamed / heap memory can be dangerous, as there is always the chance that the resource is not released / free'd properly. We need solutions to help with this.

What?

- Smart pointers
- ~~auto_ptr;~~ Unique pointer, shared pointer, Weak
- Partial construction

Recap: RAII - Making unnamed objects safe

Don't use the new / delete keyword in your own code

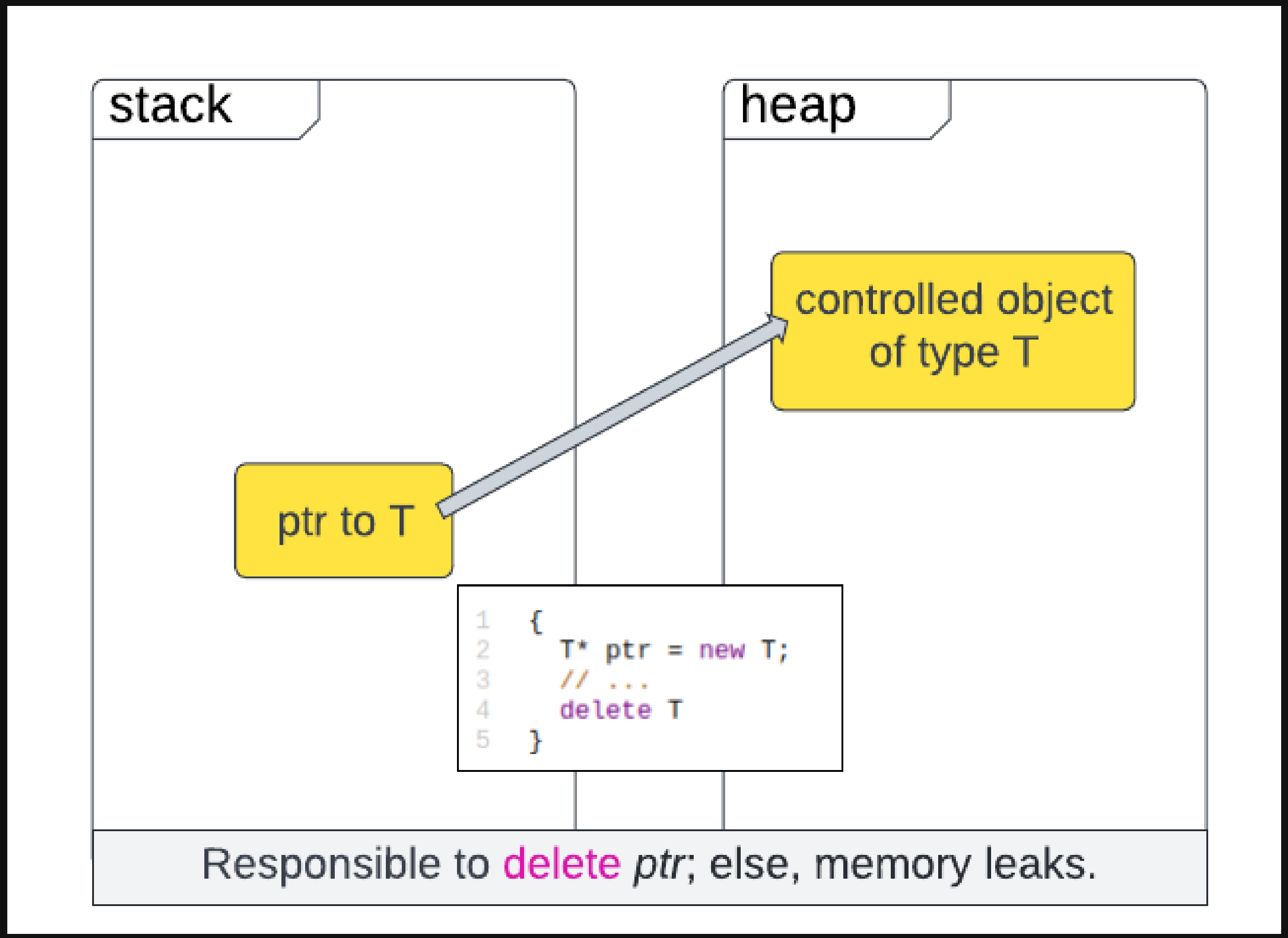
We are showing for demonstration purposes

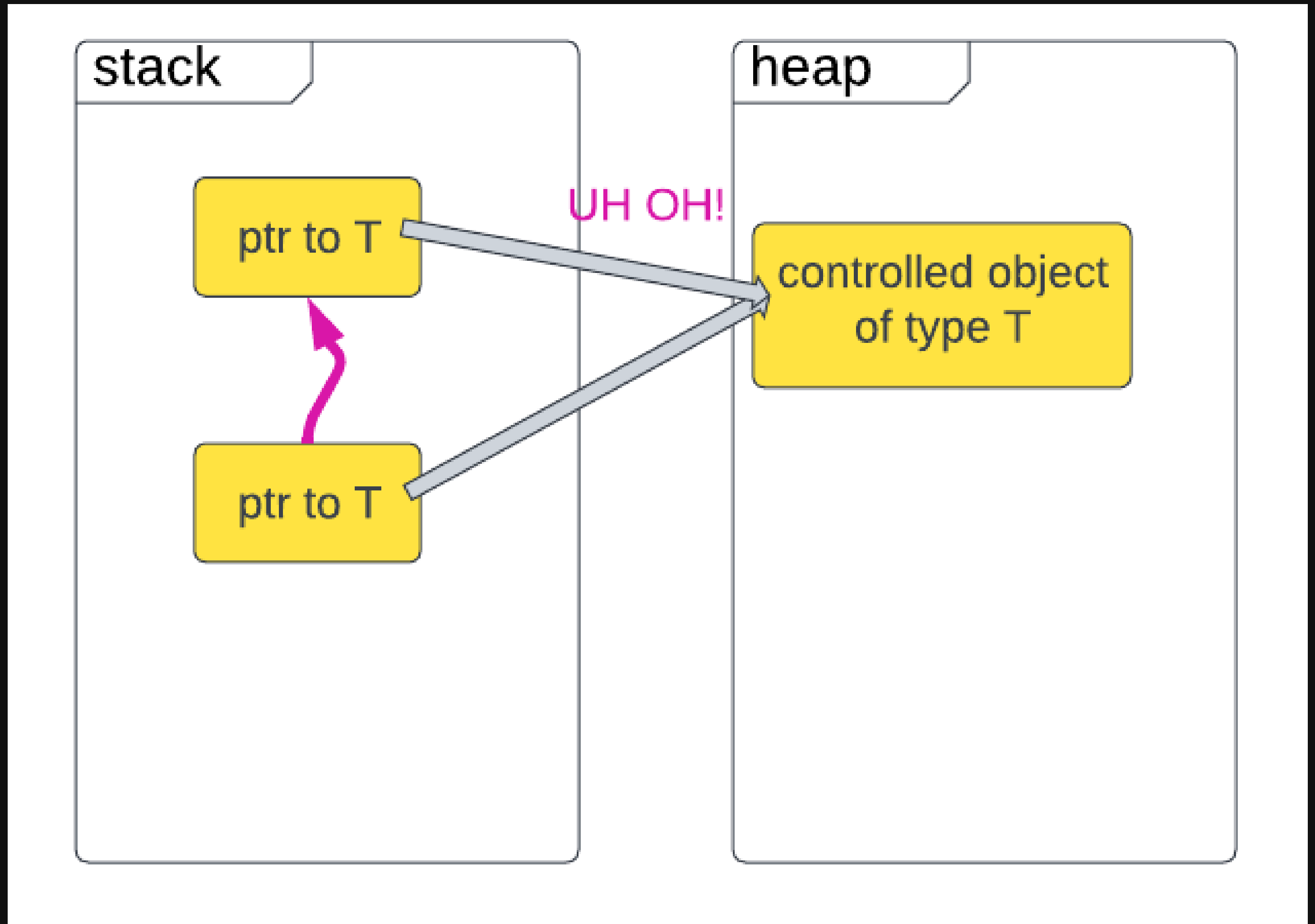
```cpp
// myintpointer.h

class MyIntPointer {
 public:
  // This is the constructor
  MyIntPointer(int* value);

  // This is the destructor
  ~MyIntPointer();

 int* GetValue();

 private:
  int* value_;
};
```

```cpp
// myintpointer.cpp
#include "myintpointer.h"

MyIntPointer::MyIntPointer(int* value): value_{value} {}

int* MyIntPointer::GetValue() {
  return value_
}

MyIntPointer::~MyIntPointer() {
  // Similar to C's free function.
  delete value_;
}
```

```cpp
void fn() {
  // Similar to C's malloc
  MyIntPointer p{new int{5}};
  // Copy the pointer;
  MyIntPointer q{p.GetValue()};
  // p and q are both now destructed.
  // What happens?
}
```

demo551-safepointer.cpp





A raw pointer (T*) is copyable. Provided a copy was made, then which of the two has ownership?

Smart Pointers

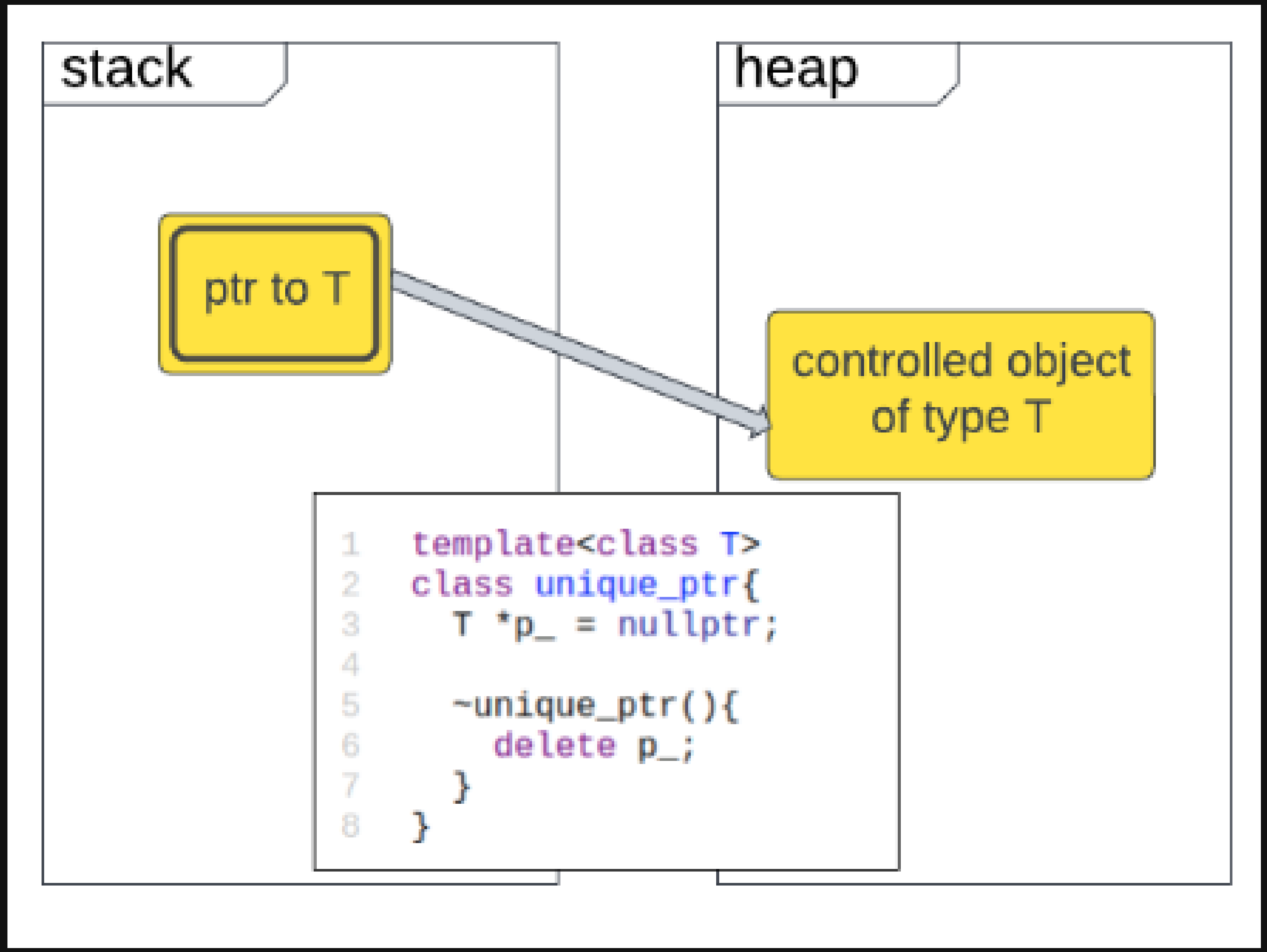
- auto_ptr vs unique_ptr
- manage the lifetime of its resources
- allocate/deallocate according to RAII (release resourse)
- support automatic memory management
- Ways of wrapping unnamed (i.e. raw pointer) heap objects in named stack objects so that object lifetimes can be managed much easier
- Introduced in C++11
- use std::unique_prt for exclusive ownership resource management.
- Usually two ways of approaching problems:
  - unique_ptr + raw pointers ("observers")
  - shared_ptr + weak_ptr/raw pointers

Type	**Shared ownership**	**Take ownership**
std::unique_ptr<T>	No	Yes
raw pointers	No	No
std::shared_ptr<T>	Yes	Yes
std::weak_ptr<T>	No	No

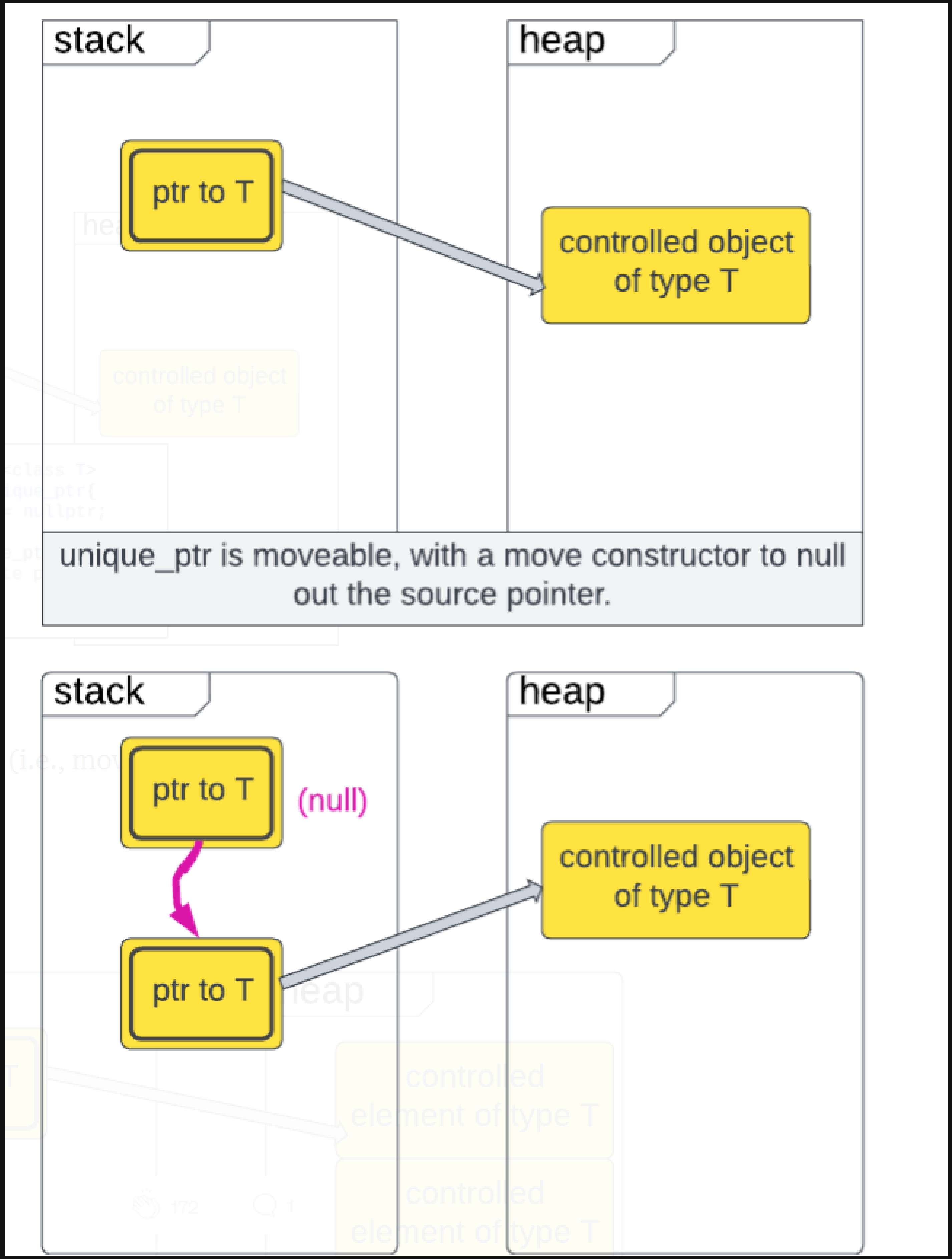
Unique pointer

- **std::unique_pointer<T>**
  - The unique pointer owns the object that handles DMA in restricted scope
  - When the unique pointer is destructed, the underlying object is too
  - Can be parameterized with deleter:**std::unique_pointer<T, deleter>**
  - **No additional/very tiny overhead compared to raw**
- **raw pointer (observer)**
  - Unique Ptr may have many observers
  - This is an appropriate use of raw pointers (or references) in C++
  - Once the original pointer is destructed, you must ensure you don't access the raw pointers (no checks exist)
  - These observers **do not** have ownership of the pointer

Also note the use of 'nullptr' in C++ instead of NULL




```
template<class T>
class unique_ptr{
  T *p_ = nullptr;

  ~unique_ptr(){
    delete p_;
  }
}
```



Unique pointer: Usage

```cpp
void my_func()
{
    int* valuePtr = new int(15);
    int x = 45;
    // ...
    if (x == 45)
        return;    // here we have a memory
    // ...
    delete valuePtr;
}

int main()
{
}
```

```cpp
#include <memory>

void my_func()
{
    std::unique_ptr<int> valuePtr(new int(15));
    int x = 45;
    // ...
    if (x == 45)
        return;    // no memory leak anymore!
    // ...
}

int main()
{
}
```

```cpp
std::unique_ptr<int> valuePtr(new int(47));

std::unique_ptr<int> valuePtr;
valuePtr.reset(new int(47));

//can be accessed just like when you would use a raw pointe
std::unique_ptr<std::string>  strPtr(new std::string);
strPtr->assign("Hello world");
```

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

int main() {
  auto up1 = std::unique_ptr<int>{new int};
  auto up2 = up1; // no copy constructor
  std::unique_ptr<int> up3;
  up3 = up2; // no copy assignment

  up3.reset(up1.release()); // OK
  auto up4 = std::move(up3); // OK
  std::cout << up4.get() << "\n";
  std::cout << *up4 << "\n";
  std::cout << *up1 << "\n";
}
```

demo552-unique1.cpp

Observer Ptr: Usage

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

int main() {
  auto up1 = std::unique_ptr<int>{new int{0}};
  *up1 = 5;
  std::cout << *up1 << "\n";
  auto op1 = up1.get();
  *op1 = 6;
  std::cout << *op1 << "\n";
  up1.reset();
  std::cout << *op1 << "\n";
}
```

demo553-observer.cpp

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

int main()
{
    std::unique_ptr<int> valuePtr(new int(15));
    std::unique_ptr<int> valuePtrNow(std::move(valuePtr));

    std::cout << "valuePtrNow = " << *valuePtrNow << '\n';
    std::cout << "Has valuePtr an associated object? "
              << std::boolalpha
              << static_cast<bool>(valuePtr) << '\n';
}
```

Can we remove "new" completely?

Unique Ptr Operators

This method avoids the need for "new". It has other benefits that we will explore.

`make_unique` is safe for creating temporaries, whereas with explicit use of new you have to remember the rule about not using unnamed temporaries.

make_unique prevents the unspecified-evaluation-order leak triggered by expressions like

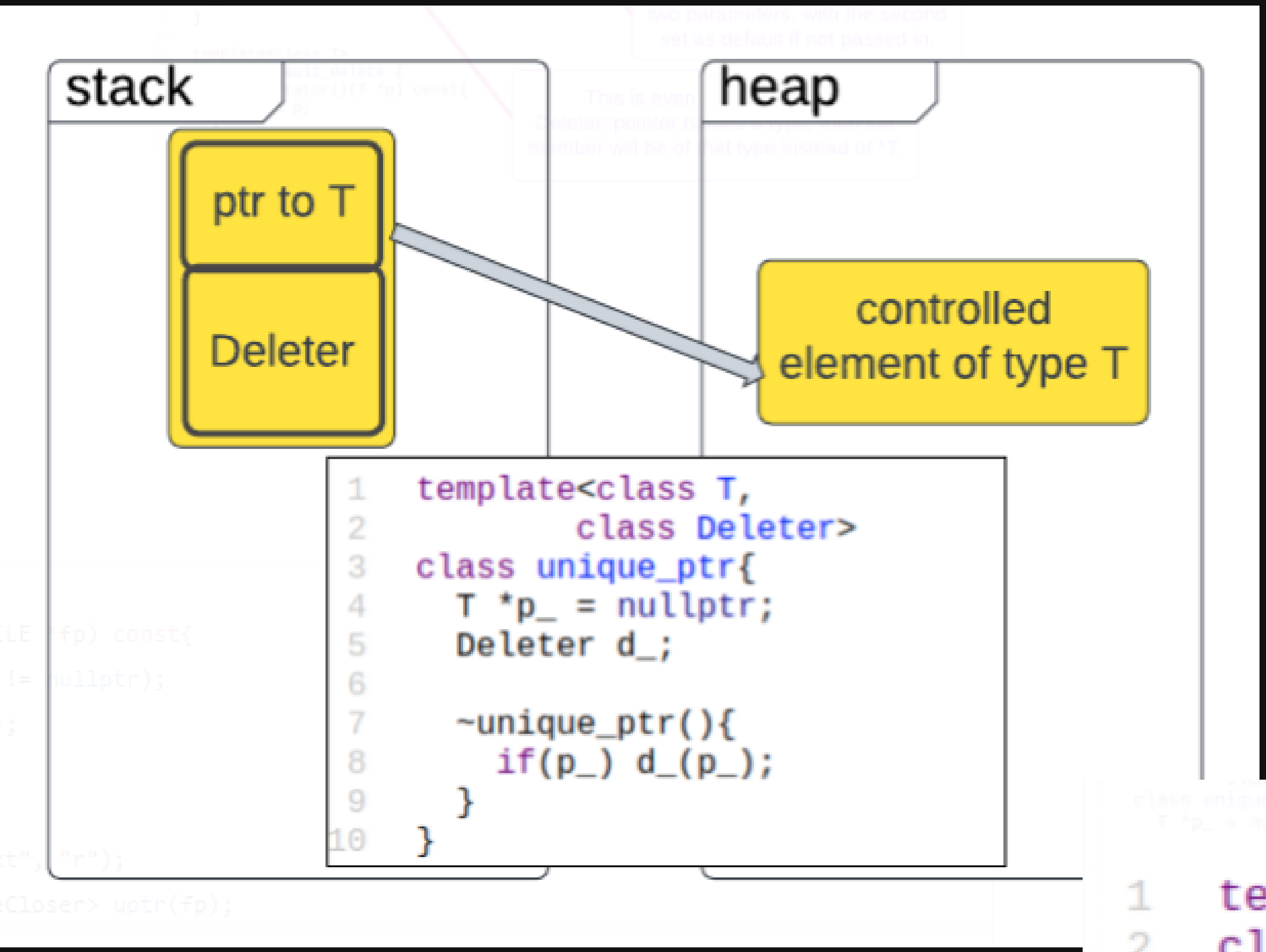
```
foo(unique_ptr<T>(new T()), unique_ptr<U>(new U())); // unsafe*

foo(make_unique<T>(), make_unique<U>()); // exception safe  // however no impact on efficiecy
```

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

auto main() -> int {
        // 1 - Worst - you can accidentally own the resource multiple
        // times, or easily forget to own it.
         auto* silly_string = new std::string{"Hi"};
         auto up1 = std::unique_ptr<std::string>(silly_string);
         auto up11 = std::unique_ptr<std::string>(silly_string);

        // 2 - Not good - requires actual thinking about whether there's a leak.
        auto up2 = std::unique_ptr<std::string>(new std::string("Hello"));

        // 3 - Good - no thinking required.
        auto up3 = std::make_unique<std::string>("Hello");

        std::cout << *up2 << "\n";
        std::cout << *up3 << "\n";
        // std::cout << *(up3.get()) << "\n";
        // std::cout << up3->size();
}
```

demo554-unique2.cpp

- https://stackoverflow.com/questions/37514509/advantages-of-using-stdmake-unique-over-new-operator
- https://stackoverflow.com/questions/20895648/difference-in-make-shared-and-normal-shared-ptr-in-c

Text




```
template<class T, class Deleter = std::default_delete<T>>
class unique_ptr{
  T *p_ = nullptr;
  Deleter d_;

  ~unique_ptr(){
    if(p_) d_(p_);
  }
}

template<class T>
struct default_delete {
  void operator()(T *p) const{
    delete p;
  }
}
```

unique_ptr is always a template of two parameters, with the second set as default if not passed in.

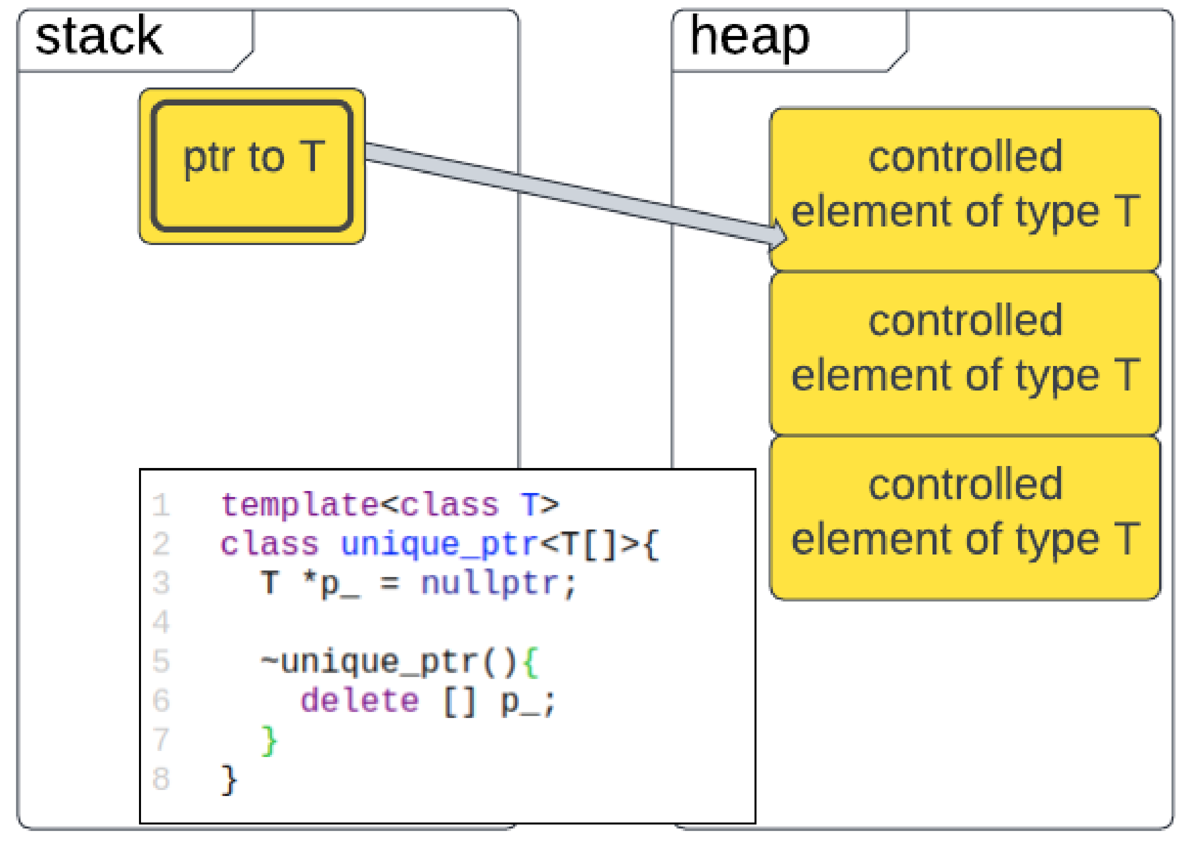
This is even customizable: if Deleter::pointer names a type, then the member will be of that type instead of *T.

Unique_ptr Array

can be specialized for array **std::unique_pointer<T []>**

unique_ptr disposes of the controlled object by calling deleter .what what about u*nique_ptr* to array of objects?

```
auto pArr = std::unique_ptr<MyClass[]>(new MyClass[10]);

```

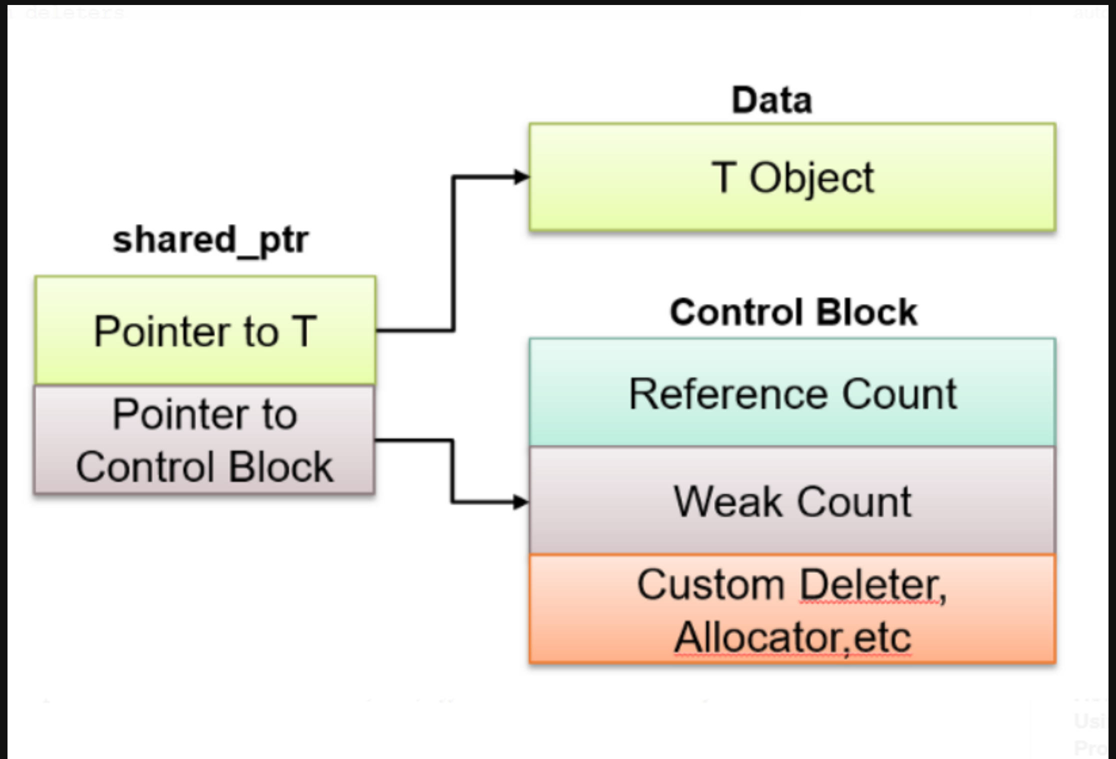


There is a specialization for array types.

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

int main()
{
    const int size = 10;
    std::unique_ptr<int[]> fact(new int[size]);

    for (int i = 0; i < size; ++i) {
        fact[i] = (i == 0) ? 1 : i * fact[i-1];
    }

    for (int i = 0; i < size; ++i) {
        std::cout << i << "! = " << fact[i] << '\n';
    }
}
```

Shared pointer

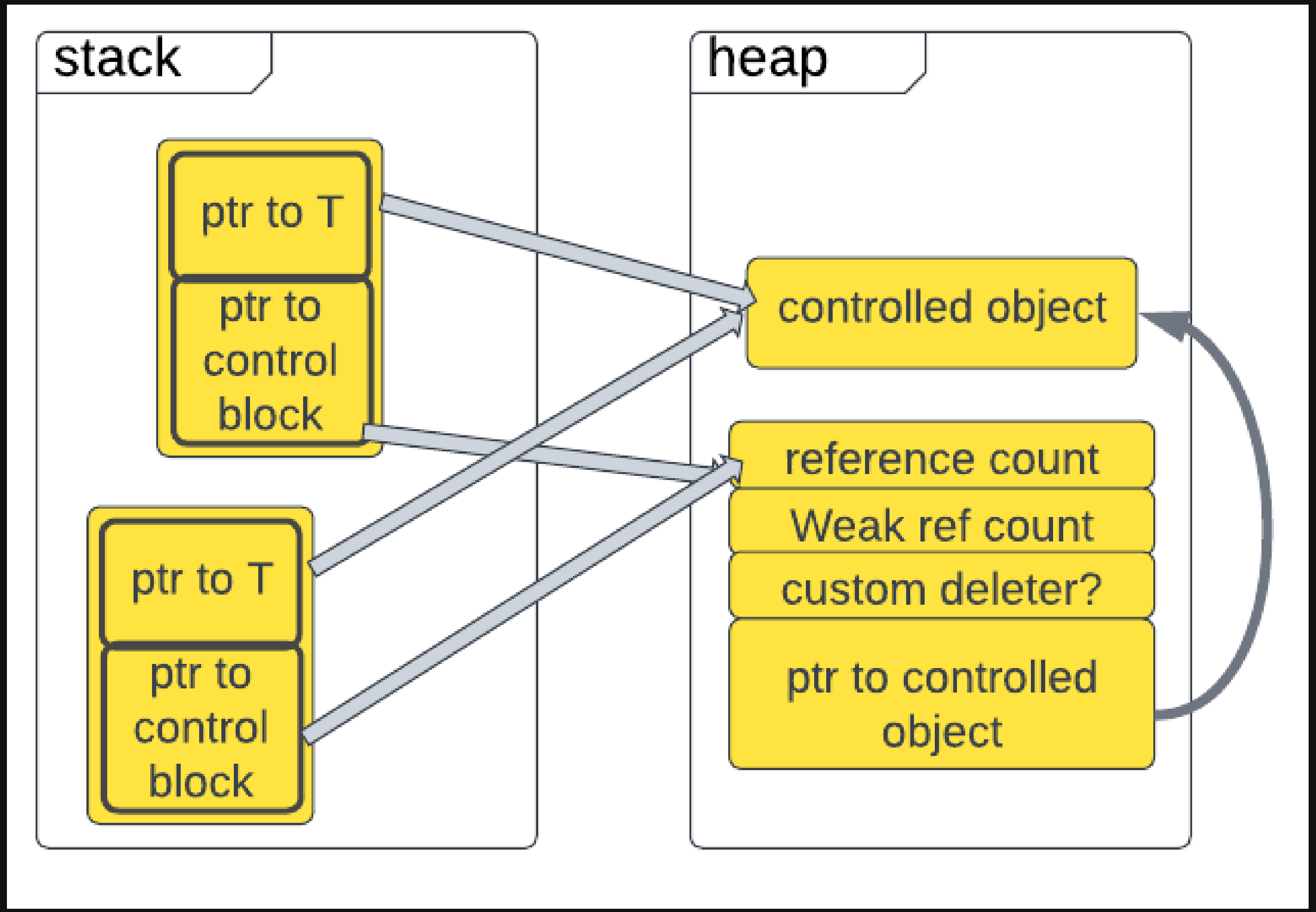
- **std::shared_pointer<T>**
- Several shared pointers share ownership of the object
  - A reference counted pointer
  - When a shared pointer is destructed, **if it is the only shared pointer left** pointing at the object, then the **object is destroyed**
  - May also have many observers
    - Just because the pointer has shared ownership doesn't mean the observers should get ownership too - don't mindlessly copy it
- **std::weak_ptr<T>**
  - Weak pointers are used with share pointers when:
    - You don't want to add to the reference count
    - You want to be able to check if the underlying data is still valid before using it.

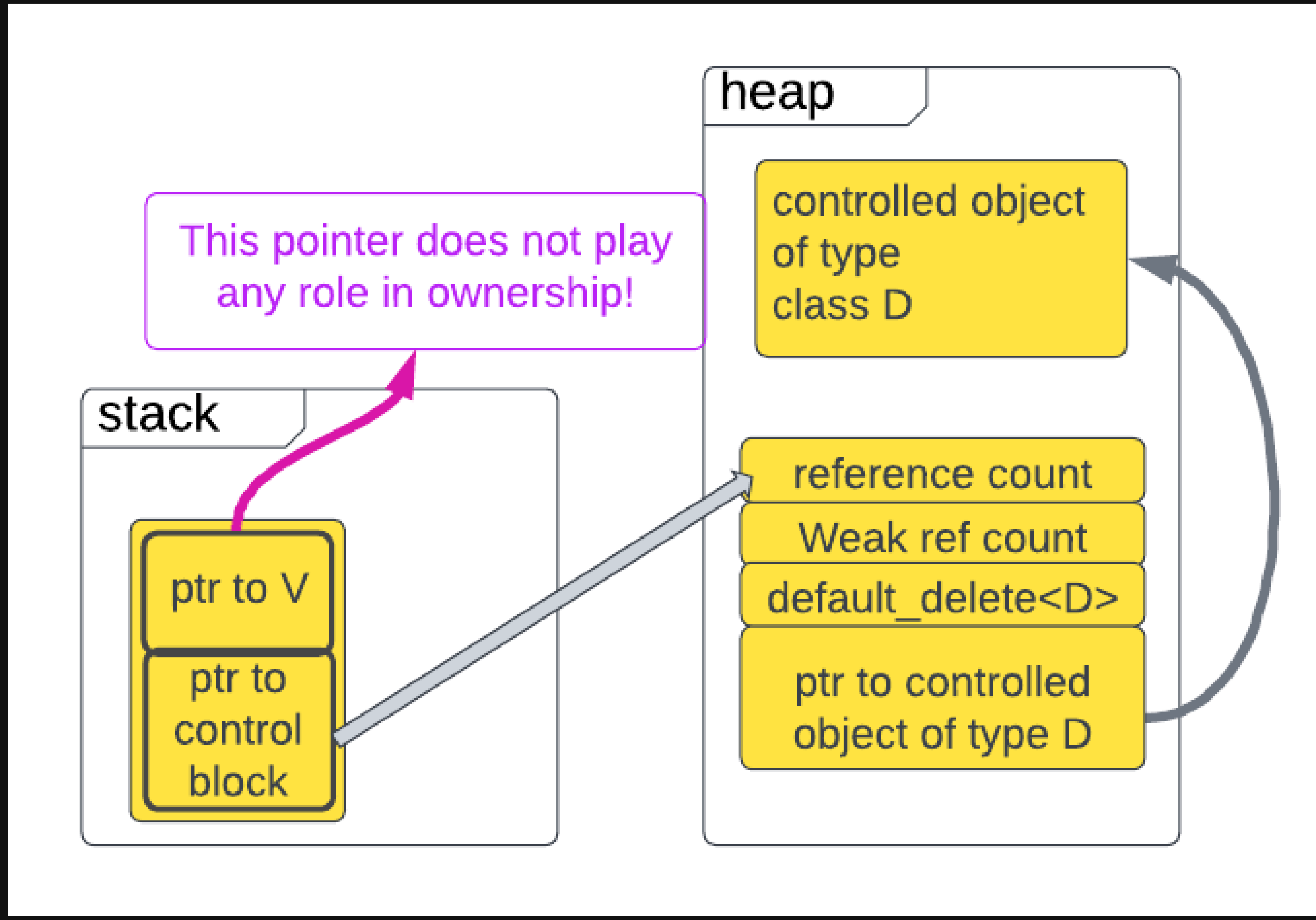


`shared_ptr`, unlike `unique_ptr`, places a layer of indirection between the physical heap-allocated object and the notion of ownership.

`shared_ptr` instances are essentially participating in ref-counted *ownership* of the **control block**.

The control block itself is the sole arbiter of what it means to “delete the controlled object.”





Shared pointer: Usage

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

auto main() -> int {
        auto x = std::make_shared<int>(5);
        std::cout << "use count: " << x.use_count() << "\n";
        std::cout << "value: " << *x << "\n";
        x.reset(); // Memory still exists, due to y.
        std::cout << "use count: " << y.use_count() << "\n";
        std::cout << "value: " << *y << "\n";
        y.reset(); // Deletes the memory, since
        // no one else owns the memory
        std::cout << "use count: " << x.use_count() << "\n";
        std::cout << "value: " << *y << "\n";
}
```

demo555-shared.cpp

Can we remove "new" completely?

Weak Pointer: Usage

Own no resourse: orrows from shared ptr

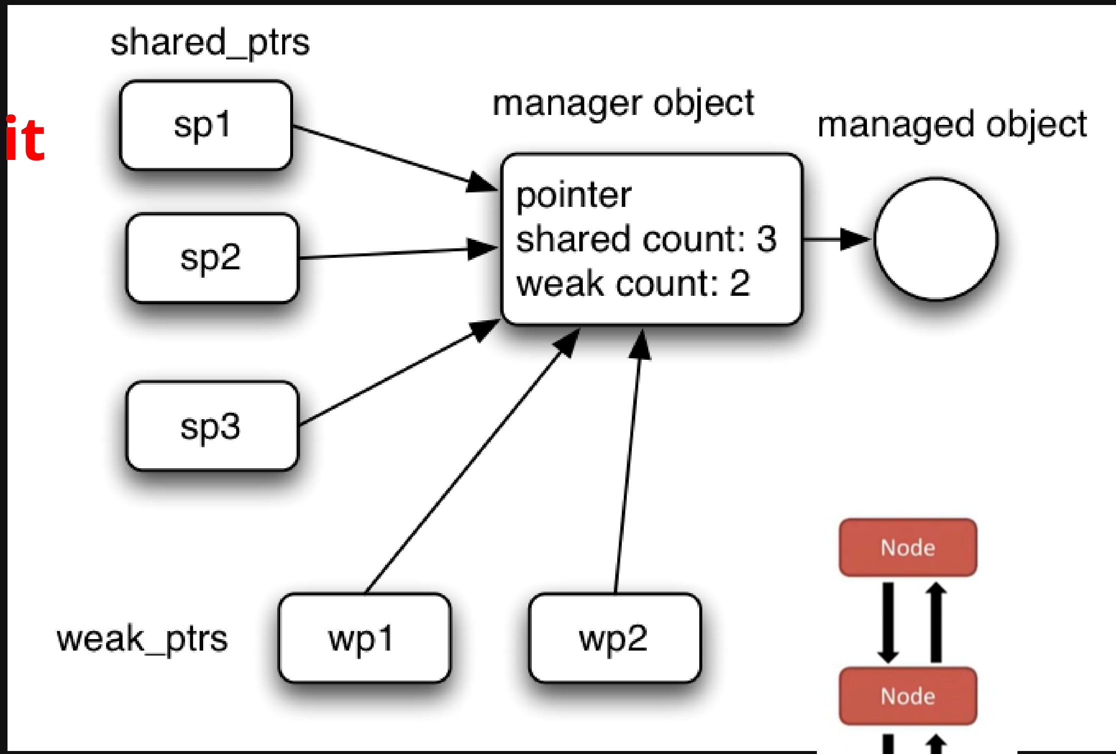
break circular dependency

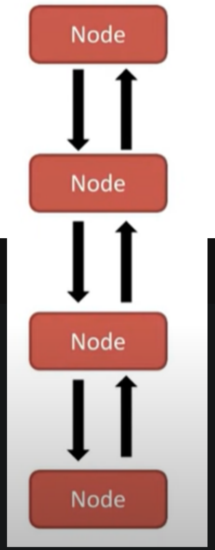
We have to convert it into Shared_ptr to use it

wp.lock();

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

auto main() -> int {
        auto x = std::make_shared<int>(1); //no ownership
//ref to objected managed by shared pointer
        auto wp = std::weak_ptr<int>(x); // x owns the memory
//wp.use_count(); wp.expired();
        auto y = wp.lock();
        if (y != nullptr) { // x and y own the memory
                // Do something with y
                std::cout << "Attempt 1: " << *y << '\n';
        }
}
```

demo556-weak.cpp





Text

```
struct Person;

struct Team{
    shared_ptr<Person> goalKeeper;
    ~Team(){cout<<"Team destructed.";}
};
struct Person{
    shared_ptr<Team> team;
    ~Person(){cout<<"Person destructed.";}
};

int main(){


    auto Barca = make_shared<Team>();
    auto Valdes = make_shared<Person>();

    Barca->goalKeeper = Valdes;
    Valdes->team = Barca;

    return 0;

}
```

If Barca goes out of scope, it is not deleted since the managed object is still pointed by valdee.team. When Valdes goes out of scope, its managed object is not deleted either as it is pointed by Barca.goalkeeper.

```
struct Person;

struct Team{
    shared_ptr<Person> goalKeeper;
    ~Team(){cout<<"Team destructed.";}
};
struct Person{
    weak_ptr<Team> team; // This line is changed.
    ~Person(){cout<<"Person destructed.";}
};

int main(){


    auto Barca = make_shared<Team>();
    auto Valdes = make_shared<Person>();

    Barca->goalKeeper = Valdes;
    Valdes->team = Barca;

    return 0;

```

When to use which type

- **Unique pointer** vs **shared pointer**
  - You almost always want a unique pointer over a shared pointer
  - Use a shared pointer if either:
    - An object has multiple owners, **and you don't know which one will stay around the longest**
    - You need temporary ownership (outside scope of this course)
    - This is very rare

Pointer	Time	Available Since
`new`	2.93 s	C++98
`std::unique_ptr`	2.96 s	C++11
`std::make_unique`	2.84 s	C++14
`std::shared_ptr`	6.00 s	C++11
`std::make_shared`	3.40 s	C++11

Smart pointer examples

- Linked list
- Doubly linked list
- Tree
- DAG (mutable and non-mutable)
- Graph (mutable and non-mutable)
- Twitter feed with multiple sections (eg. my posts, popular posts)

“Leak freedom in C++” poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don’t use owning raw *’s == don’t use explicit *delete*

Don’t create ownership cycles across modules by owning “upward” (violates layering)
Use *weak_ptr* to break cycles

Use Smart Pointers Efficiently but still use raw pointer and references ? they are not bad

Best practice: smart pointers, and minimize raw pointers or say big **NO to raw**

Raw pointer should be your default parameters and return types

sometime trade-off smart vs raw

- argument passing; but references can't be null, so are preferable
- A points to B, B points to A, or A->B->C->A
-

raw vs smart --->Premature Pessimizzation

if an entity must take a certain kind of ownership of the object, ***always*** use smart pointers - the one that gives you the kind of ownership you need.

If there is no notion of ownership, ***you may ignore*** use smart pointers but.

```
void PrintObject(shared_ptr<const Object> po) //bad
{
    if(po)
      po->Print();
    else
      log_error();
}

void PrintObject(const Object* po) //good
{
    if(po)
      po->Print();
    else
      log_error();
}
```

Stack unwinding

- Stack unwinding is the process of exiting the stack frames until we find an exception handler for the function
- This calls any destructors on the way out
  - Any resources not managed by destructors won't get freed up
  - If an exception is thrown during stack unwinding, std::terminate is called

Not safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
  g();
  // Never executed.
  delete ptr;
}
```

Not safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
    auto uni = std::unique_ptr<int>(ptr
  g();

}
```

Safe

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = std::make_unique<int>(5);
  g();
}
```

Exceptions & Destructors

- During stack unwinding, std::terminate() will be called if an exception leaves a destructor
- The resources may not be released properly if an exception leaves a destructor
- All exceptions that occur inside a destructor should be handled inside the destructor
- Destructors usually don't throw, and need to explicitly opt in to throwing
  - STL types don't do that

Partial construction

- comparatively rare in the wild
- challenge for language designers wanting to provide guarantees around invariants, immutability and concurrency-safety, and non-nullability.
- What happens if an exception is thrown halfway through a constructor?
  - The C++ standard: "An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects"
  - A destructor is not called for an object that was partially constructed i.e. root/derived
  - Except for an exception thrown in a constructor that delegates (why?)
  - two common
  - 'this' is leaked out of a constructor to some code that assumes the object has been initialized. [dont do that]
  - A failure partway through an object's construction leads to its destructor or finalizer running against a partially-constructed object. [tread with care]

Spot the bug

```cpp
#include <exception>

class my_int {
public:
   my_int(int const i) : i_{i} {
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class unsafe_class {
public:
   unsafe_class(int a, int b)
   : a_{new my_int{a}}
   , b_{new my_int{b}}
   {}

  ~unsafe_class() {
    delete a_;
    delete b_;
  }
private:
   my_int* a_;
   my_int* b_;
};

int main() {
  auto a = unsafe_class(1, 2);
}
```

demo557-bad.cpp

Partial construction: Solution

- **Safe approach: dont make it available until constructed fully**
- Option 1: Try / catch in the constructor
  - Very messy, but works (if you get it right...)
  - Doesn't work with initialiser lists (needs to be in the body)
- Option 2:
  - An object managing a resource should initialise the resource last
    - The resource is only initialised when the whole object is
    - Consequence: An object can only manage one resource
    - If you want to manage multiple resources, instead manage several wrappers , which each manage one resource

```cpp
#include <exception>
#include <memory>

class my_int {
public:
   my_int(int const i)
   : i_{i} {
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class safe_class {
public:
   safe_class(int a, int b)
   : a_(std::make_unique<my_int>(a))
   , b_(std::make_unique<my_int>(b))
   {}
private:
   std::unique_ptr<my_int> a_;
   std::unique_ptr<my_int> b_;
};

int main() {
  auto a = safe_class(1, 2);
}
```

demo558-partial1.cpp

make_shared and make_unique

- make_shared and make_unique wrap raw new, just as ~shared_ptr and ~unique_ptr wrap raw delete.
- Never touch raw pointers with hands, and then never need to worry about leaking them.
- make_shared can be performance optimization.

Function Signature	Ownership Semantic
func(value)	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of func
func(pointer*)	▪ Borrows the resource ▪ The resource could be empty ▪ Must not delete the resource
func(reference&)	▪ Borrows the resource ▪ The resource could not be empty ▪ Must not delete the resource
func(std::unique_ptr)	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of func
func(shared_ptr)	▪ Is a shared owner of the resource ▪ May delete the resource at the end of func

Feedback