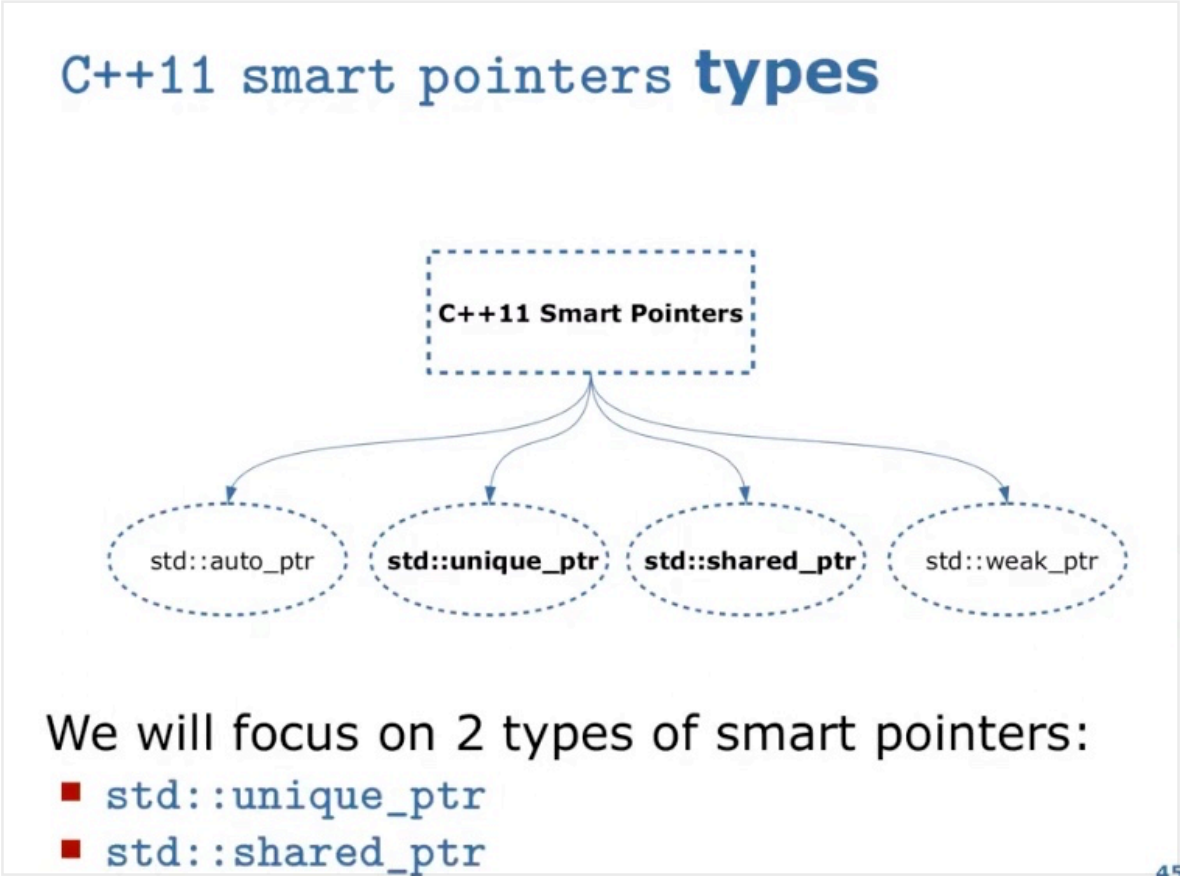
Smart Pointers

Raw pointers are hard to love

- Its declaration doesn't indicate whether it points to a single object or to an array.
- Its declaration reveals nothing about whether you should destroy what it points to when you are done using it, if the pointers owns the thing it points to.
- It may not be possible to determine whether to use the single-form "delete" or the array form "delete []". If you use wrong form, results are undefined.
- There is typically no way to tell if the pointer dangles. Dangling pointer arise when object are destroyed while pointers still point to them.

Smart Pointers

- Smart pointers wrap a raw pointer into a class and manage its lifetime (RAII).
- Smart pointers are all about ownership.
- Always use smart pointers when the pointer should own heap memory.
- Only use them with heap memory!
- Still use raw pointers for non-owning pointers and simple addressing storing.
- #include <memory> to use smart pointers.



Smart Pointers manage memory!

- Smart pointers apart from memory allocation behave exactly as raw pointers
  - Can be set to nullptr
  - Use *ptr to dereference ptr
  - Use ptr-> to access methods
  - Smart pointers are polymorphic
- Additional functions of smart pointers
  - ptr.get() returns a raw pointer that the smart pointer manages
  - ptr.reset(raw_ptr) stops using currently managed pointer, freeing its memory if needed, sets ptr to raw-ptr

Unique Pointer (std::unique_ptr)

std::unique_ptr example

- Create an `unique_ptr` to a type `Vehicle`

```
std::unique_ptr<Vehicle> vehicle_1 =
std::make_unique<Bus>(20, 10, "Volkswagen", "LPM_");

std::unique_ptr<Vehicle> vehicle_2 =
std::make_unique<Car>(4, 60, "Ford", "Sony");
```

- Now you can have fun as we had with `raw pointers`

```
// vehicle_x is a pointer, so we can us it as it is
vehicle_1->Print();
vehicle_2->Print();
```

std::unique_ptr example

- `unique_ptr` are **unique**: This means that we can move stuff but **not** copy:

```
vehicle_2 = std::move(vehicle_1);
```

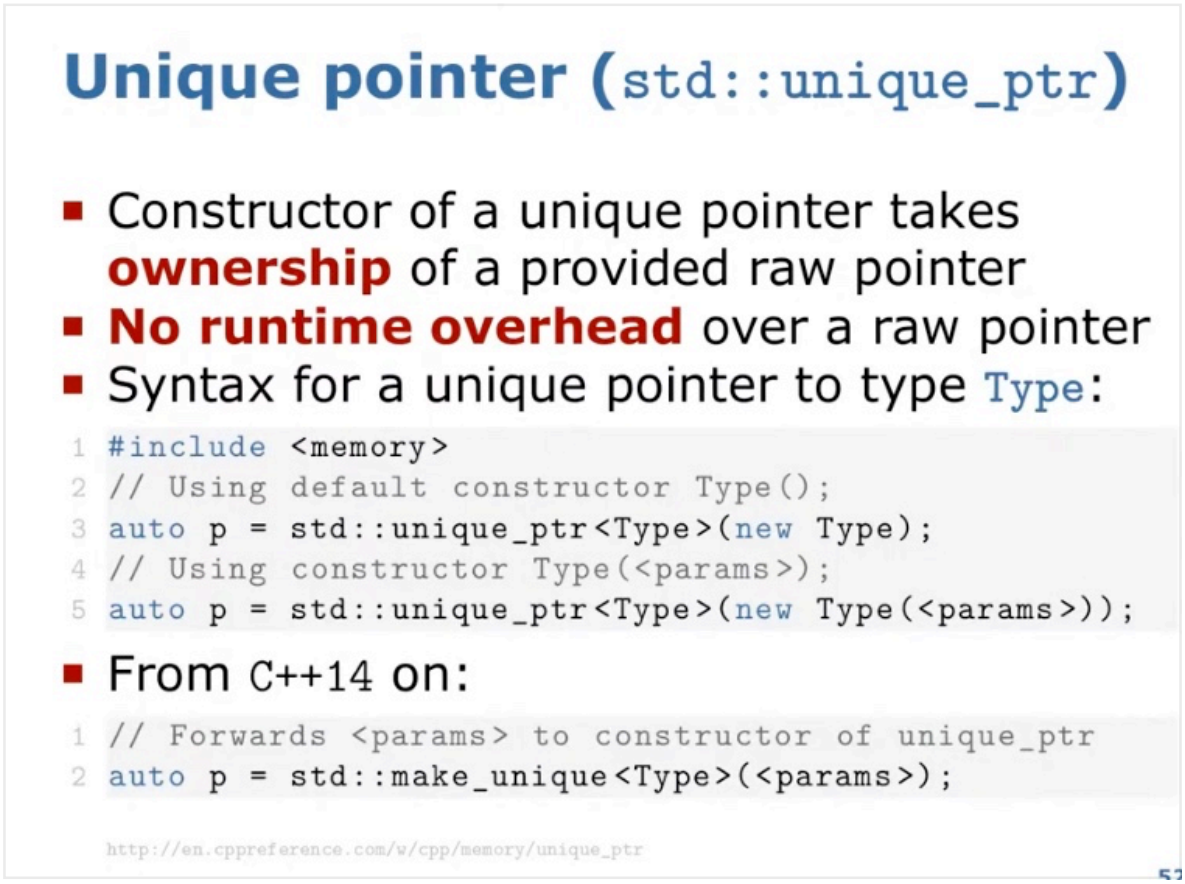
- Address of the pointers **before** the move:

```
cout << "vehicle_1 = " << vehicle_1.get() << endl;
cout << "vehicle_2 = " << vehicle_2.get() << endl;
```

```
vehicle_1 = 0x56330247ce70
vehicle_2 = 0x56330247cec0
```

- Address of the pointers **after** the move:

```
vehicle_2 = 0x56330247ce70
vehicle_1 = 0
```



What makes it "unique"?

- Unique pointer has no copy constructor
- Cannot be copied, can be moved
- Guarantees that memory is always owned by a single std::unique_ptr
- A non-null std::unique_ptr always owns what it points to
- Moving a std::unique_ptr transfers ownership from the source pointer to the destination pointer (the source pointer is set to nullptr)

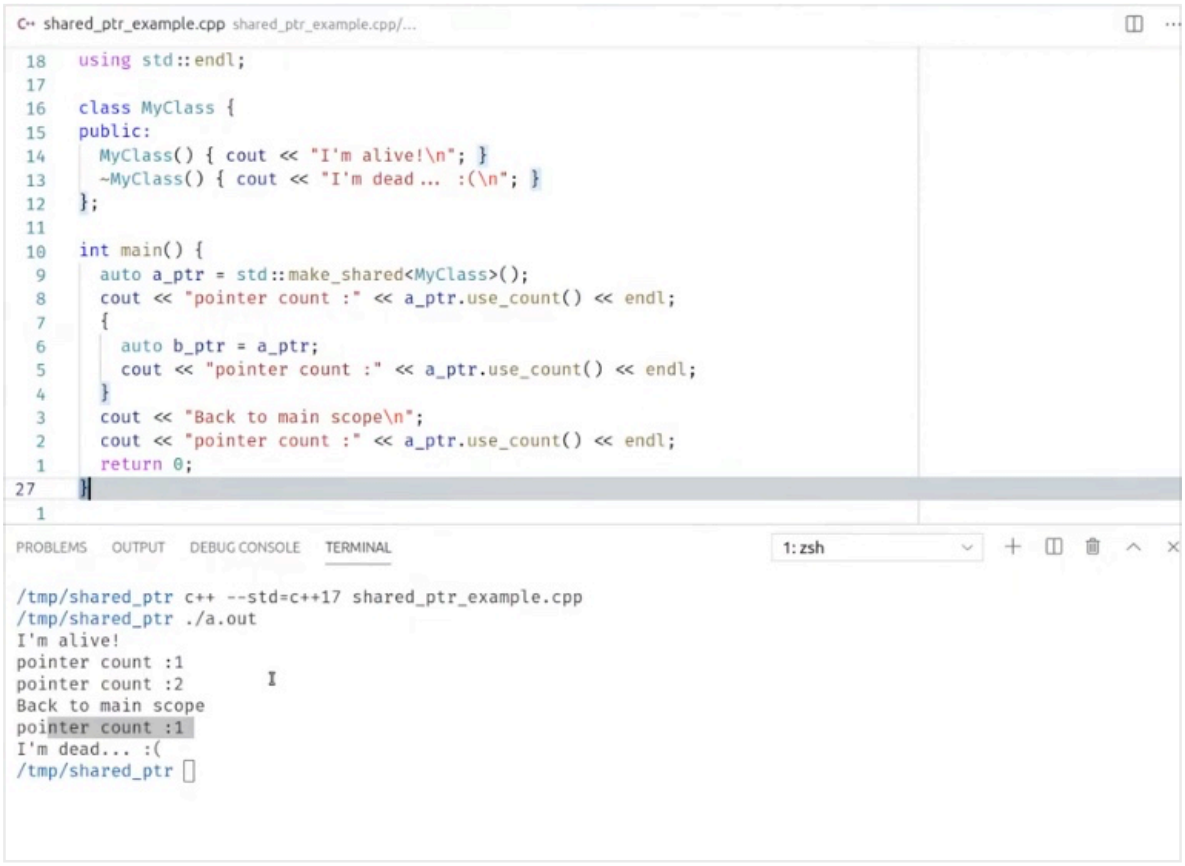
Shared Pointer (std::shared_ptr)

- What if we want to use the same pointer for different resources?
- An object accessed via std::shared_ptr has its lifetime managed by those pointers through shared ownership
- No specific std::shared_ptr owns the object
- When the last std::shared_ptr pointing to and object stops pointing there, that std:;shared_ptr destroys the object it points to
- Constructed just like unique_ptr
- Can be copied
- Stores a usage pointer and a raw pointer
  - Increases usage pointer when copied
  - Decreases usage pointer when destructed
- Frees memory when counter reaches 0

- Can be initialised from a unique_ptr

```cpp
#include <memory>
// Using default constructor Type();
auto p = std::shared_ptr<Type>(new Type);
auto p = std::make_shared<Type>();

// Using constructor Type(<params>);
auto p = std::shared_ptr<Type>(new Type(<params>));
auto p = std::make_shared<Type>(<params>);
```



When to use what?

- Use smart pointers when the pointer must among memory
- By default use unique_ptr
- If multiple objects must share ownership over something, use a shared_ptr to it
- Think of any free standing new or delete as of a memory leak or a dangling pointer:
  - Don’t use delete
  - Allocate memory with make_unique, make_shared
  - Only use new in smart pointer constructor if cannot use the functions above

Typical beginner error

```cpp
int main() {
  // Allocate a variable in the stack
  int a = 42;

  // Create a pointer to that part of the memory
  int* ptr_to_a = &a;

  // Know stuff about pointers eh?
  auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);

  // Same happens with std::shared_ptr.
  auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);

  std::cout << "Program terminated correctly!!!\n";
  return 0;
}
```

Typical beginner error


```cpp
  int* ptr_to_a = &a;

  // Know stuff about pointers eh?
  auto a_unique_ptr = std::unique_ptr<int>(ptr_to_a);

  // Same happens with std::shared_ptr.
  auto a_shared_ptr = std::shared_ptr<int>(ptr_to_a);
```

```
Program terminated correctly!!!
munmap_chunk(): invalid pointer
[1]    4455 abort (core dumped)  ./wrong_unique
```

- Create a `smart pointer` from a `pointer` to a stack-managed variable
- The variable ends up being owned both by the `smart pointer` and the stack and gets deleted twice → **Error!**

Polymorphism example using smart pointers

```
#include <memory>
#include <vector>
using std::make_unique;
using std::unique_ptr;
using std::vector;

int main() {
  vector<unique_ptr<Rectangle>> shapes;
  shapes.emplace_back(make_unique<Rectangle>(10, 15));
  shapes.emplace_back(make_unique<Square>(10));

  for (const auto &shape : shapes) {
    shape->Print();
  }

  return 0;
}
```