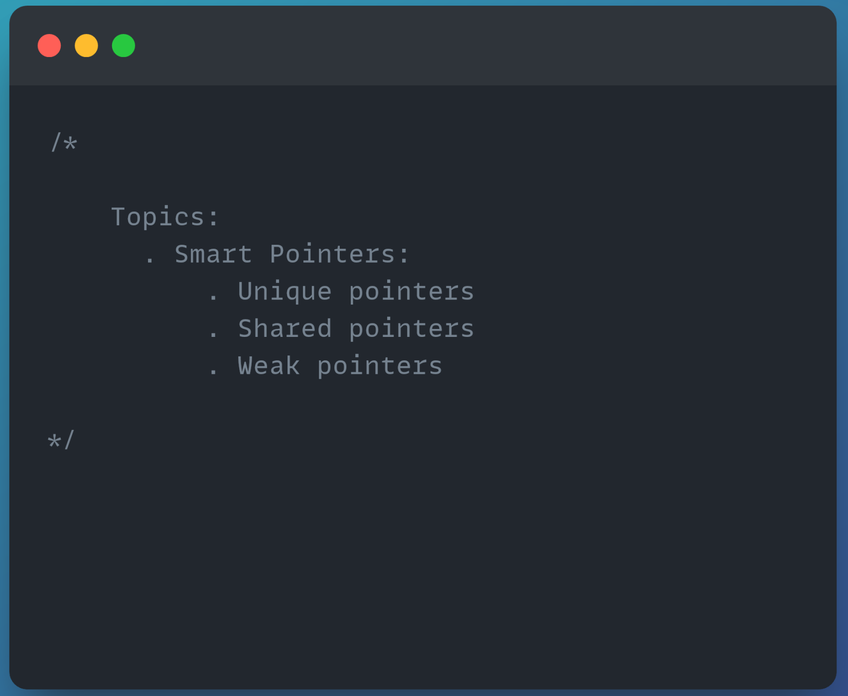
Smart Pointers



```
/*

    Topics:
      . Smart Pointers:
          . Unique pointers
          . Shared pointers
          . Weak pointers

*/
```

IMPORTANT	*In modern C++, you should use smart pointers as much as possible for your memory management needs. Memory will be automatically released when the object goes out of scope. It's natural! Just like stack objects.

Stack variables

```rust
// Stack variables
{
    int int1 = 10;
    // Calling functions on stack objects
    fmt::println("Integer is: {}", int1);

}
fmt::println( "Hitting outside scope" );
fmt::println( "Done!" );
```

Raw pointers

```cpp
// Using raw pointers : Remember to manually release memory,
//                         if you don't relase , you leak memory
int* p_int2 = new int(20);
int* p_int1 = new int(100);

fmt::println( "Integer is: {}" , *p_int2 );
fmt::println( "Integer lives at address: {}" , fmt::ptr(p_int2 ));

//If you go out of scope withoug releasing (deleting) p_int2 and
// p_int1 you'll have leaked memory
delete p_int2; // Calls the destructor
delete p_int1;
```

std::unique_ptr

```cpp
// Using unique_ptr: managed memory is owned by a single pointer at any moment.
{
    int* p_int3 = new int(30);
    std::unique_ptr<int> up_int4{p_int3}; // Can also manage a previously allocated
                                          // space managed by a raw pointer. You shouldn't
                                          // try to use the raw pointer from this point on
    std::unique_ptr<int> up_int5 {new int(50)};
    std::unique_ptr<int> up_int6{nullptr};// Can also initialize with nullptr
                    // and give it memory to manage later, we'll see how to
                    // do that with std::move later in the lecture. Just know
                    // that you can initialize a unique ptr with nullptr for now.

    //Can use unique pointer just like we use a raw pointer.
    fmt::println( "Integer is: {}" , *up_int5 ); // dereferencing
    fmt::println( "Integer lives at address: {}" , fmt::ptr(up_int5.get() ));

}
fmt::println( "Hitting the outside scope" );
```

std::unique_ptr

```cpp
//using make_unique syntax. Much cleaner (C++14 )
//Calls new internally for us, we don't have to do it ourselves
std::unique_ptr<int> up_int7 = std::make_unique<int>(70);
fmt::println( "Value pointed to by up_int7 is: {}"  , *up_int7 );
fmt::println( "up_int7 pointing at address: {}" , fmt::ptr(up_int7.get() ));
```

std::unique_ptr: Copies not allowed

```cpp
//Copies not allowed
std::unique_ptr<int> up_int8 = std::make_unique<int>(80);
fmt::println( "up_int8 pointing at address: {}" , fmt::ptr(up_int8.get() ));

// Copies and Assignments are not allowed with unique ptr
//std::unique_ptr<int> up_int9 = up_int8; // Error.This also does some kind of copy
                    // More on this when we've learnt about operator overloading
//std::unique_ptr<int> up_int10{up_int8}; // Error :  Copy constructor deleted
```

std::unique_ptr: Moving the pointer

```cpp
// Can however move the pointer.
std::unique_ptr<int> up_int11 = std::make_unique<int>(110);
{
    std::unique_ptr<int> up_int12 = std::move(up_int11); // Now up_int12 manages int11
                                                         // and up_int11 points to nothing(nullptr)
    fmt::println( "up_int12 pointing at address :{}" , fmt::ptr(up_int12.get() ));

    fmt::println( "up_int11 is now nullptr : {}" ,fmt::ptr( up_int11.get() ));
    if(up_int11){
        fmt::println( "Pointer11 pointing to something valid" );
    }else{
        fmt::println( "Pointer11 point to nullptr" );
    }

}
fmt::println( "Hitting the outside scope" );
```

std::unique_ptr: Resetting

```cpp
// Can reset unique_ptr : releases memory and sets the pointer to nullptr
std::unique_ptr<int> up_int13 = std::make_unique<int>(130);
up_int13.reset();// releases memory and sets pointer to nullptr

// Can use unique pointer in if statement to see if it points somewhere valid
if (up_int13) {
    fmt::println("up_int13 points somewhere valid : {}", fmt::ptr(up_int13.get()));
} else {
    fmt::println("up_int13 points is null : {}", fmt::ptr(up_int13.get()));
}
```

Shared pointers

```
/*
        . Shared pointers
            . Several pointers can share the same object
            . We use reference count to keep track of how many pointers to the object we have
            . When the last pointer goes out of scope, the object is destroyed.
*/
```

Shared pointers

```cpp
std::shared_ptr<int> int_ptr_1 {new int{20}};

fmt::println( "The pointed to value is: {}" , *int_ptr_1 );
*int_ptr_1 = 40; // Use the pointer to assign
fmt::println( "The pointed to value is: {}" , *int_ptr_1 );
fmt::println( "Use count: {}" , int_ptr_1.use_count() ); //1
fmt::println( "----" );

//Copying

std::shared_ptr<int> int_ptr_2 = int_ptr_1; // Use count : 2

fmt::println( "The pointed to value is (through int_ptr2): {}" , *int_ptr_2 );
*int_ptr_2 = 70;
fmt::println( "The pointed to value is (through int_ptr2): {}" , *int_ptr_2 );
fmt::println( "Use count for int_ptr_1: {}" , int_ptr_1.use_count() );
fmt::println( "Use count for int_ptr_2: {}" , int_ptr_2.use_count() );

//Other ways to initialize shared pointers
fmt::println( "-----" );
std::shared_ptr<int> int_ptr_3; // nullptr
int_ptr_3 = int_ptr_1; // Use count : 3

std::shared_ptr<int> int_ptr_4{nullptr};
int_ptr_4 = int_ptr_1; // Use count : 4

std::shared_ptr<int> int_ptr_5{int_ptr_1}; // Use count : 5

fmt::println( "The pointed to value is (through int_ptr5): {}" , *int_ptr_5 );
*int_ptr_5 = 100;
fmt::println( "The pointed to value is (through int_ptr5): {}" , *int_ptr_5 );
```

Shared pointers: std::make_shared

```cpp
std::shared_ptr<int> int_ptr_6 = std::make_shared<int>(55);
fmt::println("The value pointed to by int_ptr_6 is: {}", *int_ptr_6);


fmt::println("int_ptr_6 use count: {}", int_ptr_6.use_count());// 1


// Share the object(data) with other shared_ptr's
fmt::println("Share the object(data) with other shared_ptr's");
std::shared_ptr<int> int_ptr_7{ nullptr };
int_ptr_7 = int_ptr_6;

fmt::println("int_ptr6 use count : {}", int_ptr_6.use_count());

fmt::println("Reset ptr6's");
int_ptr_6.reset();// decrement reference count, and set int_ptr6 to nullptr
                  // if reference count is zero, release the managed memory
fmt::println("int_ptr_6 use count : {}", int_ptr_6.use_count());
fmt::println("int_ptr_7 use count : {}", int_ptr_7.use_count());
```

Weak pointers

```
/*

            Weak pointers:
                  . They are rarely used
                  . Used to prevent circular references in shared ownership situations
                  . We'll see an example later.
*/
```