Slides

Development > Programming Languages > C++

The C++ 20 Masterclass : From Fundamentals to Advanced

Learn and Master Modern C++ From Beginning to Advanced in Plain English : C++11, C++14, C++17, C++20 and More!

4.7 ★★★★☆

Created by Daniel Gakwaya

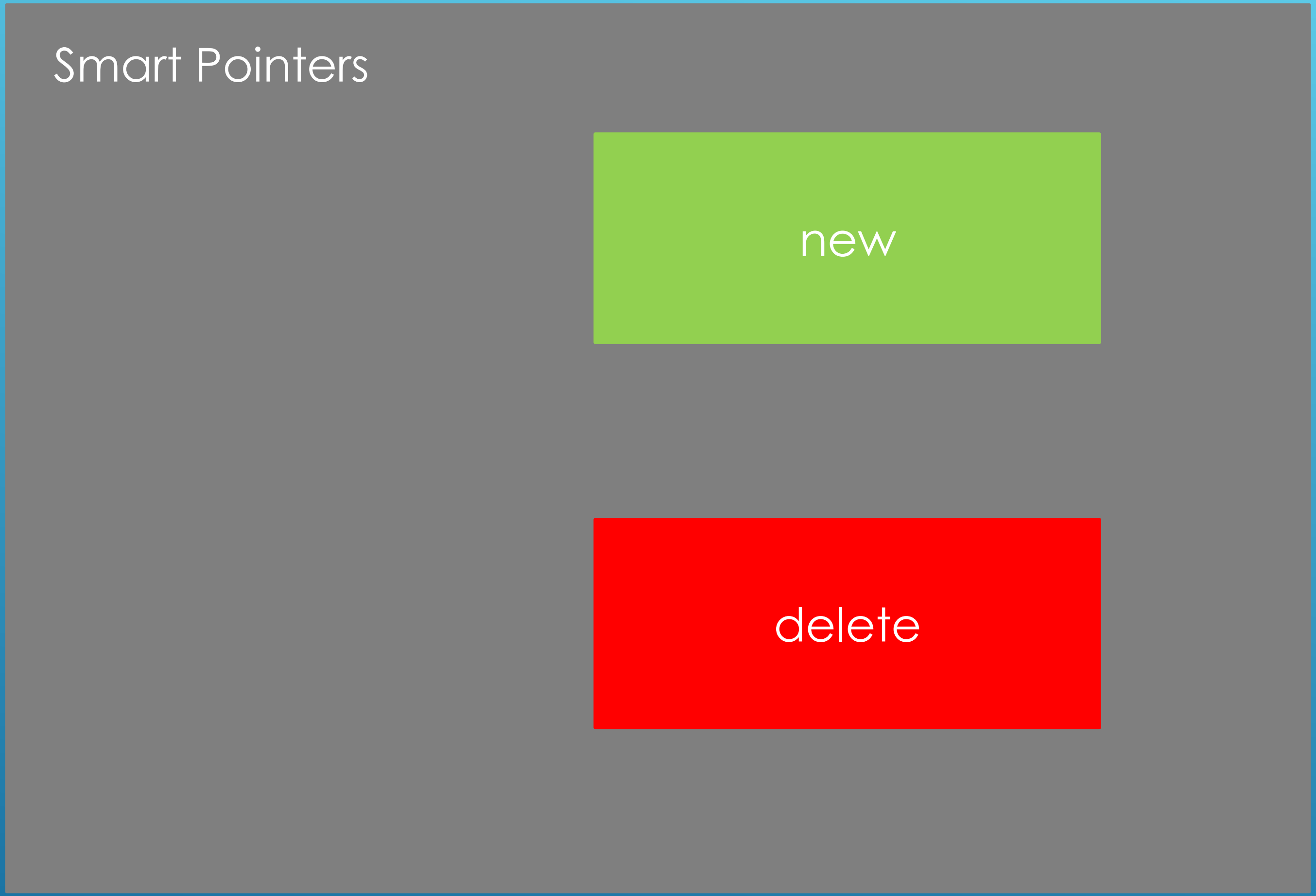
Section : Smart Pointers

Slide intentionally left empty

Smart Pointers

Manually releasing memory yourself through the delete operator for raw pointers is a pain in the neck. Smart pointers are a solution offered by modern C++ to release the memory automatically when the pointer managing the memory goes out of scope new

delete


Smart Pointers
new
delete

<memory>

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

Slide intentionally left empty

Unique pointers

- At any given moment there can only be one pointer managing the memory
- Memory is automatically released when the pointer goes out of scope

<memory>

Stack variables

```
Dog dog1("Dog1");
// Calling functions on stack objects
dog1.print_dog();
```

Raw pointers

```cpp
Dog* p_dog2 = new Dog("Dog2");
int * p_int1 = new int(100);

p_dog2->print_dog();
std::cout << "Integer is : " << *p_int1 << std::endl;
std::cout << "Integer lives at address : " << p_int1 << std::endl;

//If you go out of scope withoug releasing (deleting) p_dog2 and
// p_int1 you'll have leaked memory
delete p_dog2;
delete p_int1;
```

Unique pointers

```
Dog * p_dog_3 = new Dog("Dog3");
std::unique_ptr<Dog> up_dog_4{p_dog_3}; // 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<Dog> up_dog_5 {new Dog("Dog5")};
std::unique_ptr<int> up_int2{new int(200)};
std::unique_ptr<Dog> up_dog_6{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.
up_dog_5->print_dog(); // Calling function with -> operator
//Assign to fundamental type
* up_int2 = 500;
std::cout << "Integer is : " << *up_int2 << std::endl; // dereferencing
std::cout << "Integer lives at address : " << up_int2.get() << std::endl;
```

Std::make_unique

```
std::unique_ptr<Dog> up_dog_7 = std::make_unique<Dog>("Dog7");
up_dog_7->print_dog();

std::unique_ptr<int> p_int3 = std::make_unique<int>(30);
*p_int3 =67;
std::cout << "Value pointed to by p_int3 is :"  << *p_int3 << std::endl;
std::cout << "p_int pointing at address :" << p_int3.get() << std::endl;
```

Can't copy unique pointers

```
std::unique_ptr<Dog> up_dog_8 = std::make_unique<Dog>("Dog8");
up_dog_8->print_dog();
//Can get the wrapped pointer address : useful for older raw pointer APIs
std::cout << "Dog8 memory address: " << up_dog_8.get() << std::endl;

// Copies and Assignments are not allowed with unique ptr
std::unique_ptr<Dog> up_dog_9 = up_dog_8; // Error.This also does some kind of copy
                    // More on this when we've learnt about operator overloading
std::unique_ptr<Dog> up_dog_10{up_dog_8}; // Error :  Copy constructor deleted
```

Moving ownership

```
std::unique_ptr<Dog> up_dog_11 = std::make_unique<Dog>("Dog11");
{
    std::unique_ptr<Dog> up_dog_12 = std::move(up_dog_11); // Now up_dog_12 manages Dog11
                                                           // and up_dog_11 points to nothing(nullptr)
    up_dog_12->print_dog();
    std::cout << "Dog12 memory address : " << up_dog_12.get() << std::endl;

    std::cout << "up_dog_11 is now nullptr : " << up_dog_11.get() << std::endl;

}
```

Resetting

```cpp
std::unique_ptr<Dog> up_dog_13 = std::make_unique<Dog>("Dog13");
up_dog_13.reset(); // releases memory and sets pointer to nullptr

//Can use unique pointer in if statement to see if it points somewhere valid
if(up_dog_13){
    std::cout << "up_dog_13 points somewhere valid : " << up_dog_13.get() << std::endl;
}else{
    std::cout << "up_dog_13 points is null : " << up_dog_13.get() << std::endl;
}
```

Slide intentionally left empty

Unique pointers as function parameters & return values

Passing by value

```cpp
void do_something_with_dog_v1( std::unique_ptr<Dog> d){
    d->print_info() ;
}
```

Passing by value

```
//Passing unique ptr to functions by value
std::unique_ptr<Dog> p_dog_1 = std::make_unique<Dog>("Dog1");

//Can't pass unique_ptr by value to a function : copies not allowed
do_something_with_dog_v1(p_dog_1);   // copy detected,
                                        //not allowed to copy unique ptr. Compiler error

do_something_with_dog_v1(std::move(p_dog_1)); // Ownership will move to the body
                                               // of the function and memory will be
                                               // released when function returns.
                                               // Not what you typically want.
std::cout << "delimiter" << std::endl;

Person person1("John");
person1.adopt_dog(std::move(p_dog_1)); // The same behavior when function is part of the class
std::cout << "Doing something , p_dog_1 points to :  " <<p_dog_1.get() <<  std::endl;


//An implicit move is done when object is created in place as a temporary
do_something_with_dog_v1(std::make_unique<Dog>("Temporary Dog"));
std::cout << "delimiter" << std::endl;
```

Passing by reference

```
void do_something_with_dog_v2( const std::unique_ptr<Dog>& d){
    d->set_dog_name("Rior");
    d->print_info();
    //d.reset(); // Compiler error
}
```

Returning by value

```cpp
std::unique_ptr<Dog> get_unique_ptr(){
    std::unique_ptr<Dog> p_dog = std::make_unique<Dog>("Function Local");
    std::cout << "unique_ptr address(in) : " << &p_dog << std::endl;
    return p_dog;  // The compiler does some optimizations and doesn't return a copy here
                    // it's returning something like a reference to the local object.
                    // We can prove this by looking at the address of objects in memory.
                    // This is not standard behavior, some compilers may actually return
                    // by value by making a copy. The compilers have some freedom to choose
                    // their own way to do things.
}
```

Slide intentionally left empty

Unique pointers and arrays

Array managed by unique_ptr

```
//Array allocated on the heap with unique_ptr. Releases space for array automatically
{
    std::cout << std::endl;
    std::cout << "Array on heap with unique ptr" << std::endl;

   auto arr_ptr = std::unique_ptr<Dog[]> ( new Dog[3]{Dog("Dog7"), Dog("Dog8") , Dog("Dog9")});

   for (size_t i{0} ; i < 3 ; ++i){
       arr_ptr[i].print_info() ;
   }

}
```

make_unique

```
//Array allocated on the heap with unique_ptr. Releases space for array automatically
{
    std::cout << std::endl;
    std::cout << "Array on heap with unique ptr" << std::endl;

   //auto arr_ptr = std::unique_ptr<Dog[]> ( new Dog[3]{Dog("Dog7"), Dog("Dog8") , Dog("Dog9")});
   auto arr_ptr = std::make_unique<Dog[]>(3);// Works. Can't initialize individual elements
   //auto arr_ptr = std::make_unique<Dog[]>(3) {Dog("Dog7"), Dog("Dog8") , Dog("Dog9")};//Compiler error
   //auto arr_ptr = std::make_unique<Dog[]>{Dog("Dog7"), Dog("Dog8") , Dog("Dog9")};//Compiler error

   for (size_t i{0} ; i < 3 ; ++i){
       arr_ptr[i].print_info() ;
   }

}
```

Slide intentionally left empty

std::unique_ptr : best practices

Unique_ptr member variables

```cpp
class Point {
public :
    Point() = default;
    Point(double x_param, double y_param)
    {
        //Can't assign the return value of make_unique directly to a
        // std::unique_ptr object. Copies not allowed. Have to explicitly move
        x = std::move(std::make_unique<double>(x_param));
        y = std::move(std::make_unique<double>(y_param));
    }

    void print_info()const{
        std::cout << "Point [ x : " << *x << ", y : " << *y << " ]" << std::endl;
    }

private :
    std::unique_ptr<double> x{};
    std::unique_ptr<double> y{};
};
```

```cpp
Point point1(10.0,20.0);
point1.print_info();

Point * point2 = new Point(30.0,40.0); // If you forget to delete
                                       // object memory won't be released. Smart
                                       // pointers won't help here.
```

```cpp
//Don’t let multiple classes manage the same resource. For example:
//Compiler allows it, but two unique_ptr may release the same memory twice : BAD!
// Even worse, one might try to use memory already deleted by other
Dog *dog{ new Dog() };
std::unique_ptr<Dog> p_dog1{ dog };
std::unique_ptr<Dog> p_dog2{ dog };



//Don't do weird stuff behind the back of unique_ptr
Dog *dog1{ new Dog() };
std::unique_ptr<Dog> p_dog3{ dog1 };
delete dog;
```

Using std::make_unique eliminates the last two problems. You're not directly dealing with the raw pointer, so you can't easily misuse it. In modern C++, strive to use smart pointers as much as possible and use new and delete directly only if really necessary.

Slide intentionally left empty

Shared Pointers


ptr_1
ptr_2
ptr_3
ptr_4
data

Shared pointers with fundamental types

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

std::cout << "The pointed to value is : " << *int_ptr_1 << std::endl;
*int_ptr_1 = 40; // Use the pointer to assign
std::cout << "The pointed to value is : " << *int_ptr_1 << std::endl;
std::cout << "Use count : " << int_ptr_1.use_count() << std::endl;

//Copying
std::cout << std::endl;
std::cout << "Copying..." << std::endl;
std::shared_ptr<int> int_ptr_2 = int_ptr_1; // Use count : 2

std::cout << "The pointed to value is (through int_ptr2)  : " << *int_ptr_2 << std::endl;
*int_ptr_2 = 70;
std::cout << "The pointed to value is (through int_ptr2) : " << *int_ptr_2 << std::endl;

std::cout << "Use count for int_ptr_1 : " << int_ptr_1.use_count() << std::endl;
std::cout << "Use count for int_ptr_2 : " << int_ptr_2.use_count() << std::endl;
```

Shared pointers with fundamental types

```
std::cout << std::endl;
std::cout << "Initializing..." << std::endl;
std::shared_ptr<int> int_ptr_3;
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

 std::cout << "The pointed to value is (through int_ptr5)  : " << *int_ptr_5 << std::endl;
*int_ptr_5 = 100;
std::cout << "The pointed to value is (through int_ptr5) : " << *int_ptr_5 << std::endl;

std::cout << "Use count for int_ptr_1 : " << int_ptr_1.use_count() << std::endl;
std::cout << "Use count for int_ptr_2 : " << int_ptr_2.use_count() << std::endl;
std::cout << "Use count for int_ptr_3 : " << int_ptr_3.use_count() << std::endl;
std::cout << "Use count for int_ptr_4 : " << int_ptr_4.use_count() << std::endl;
std::cout << "Use count for int_ptr_5 : " << int_ptr_5.use_count() << std::endl;
```

Shared pointers with fundamental types

```
//Reset : decrements the use count and sets the pointer to nullptr
std::cout << std::endl;
std::cout << "Reset..." << std::endl;
int_ptr_5.reset(); // decrements reference count and sets int_ptr5 to nullptr
                // after this if you show use count, for int_ptr5,you'll get 0
std::cout << "Use count for int_ptr_1 : " << int_ptr_1.use_count() << std::endl;
std::cout << "Use count for int_ptr_2 : " << int_ptr_2.use_count() << std::endl;
std::cout << "Use count for int_ptr_3 : " << int_ptr_3.use_count() << std::endl;
std::cout << "Use count for int_ptr_4 : " << int_ptr_4.use_count() << std::endl;
std::cout << "Use count for int_ptr_5 : " << int_ptr_5.use_count() << std::endl;

 //Can get the raw pointer address and use the ptr in if statements (castable to bool)
std::cout << std::endl;
std::cout << "Casting to bool and using in if statements..." << std::endl;
std::cout << "int_ptr_4 : " << int_ptr_4 << std::endl;
std::cout << "int_ptr_4.get() : " << int_ptr_4.get() << std::endl;
std::cout << std::boolalpha;
std::cout << "int_ptr_4->bool : " << static_cast<bool>(int_ptr_4) << std::endl;
std::cout << "int_ptr_5->bool : " << static_cast<bool>(int_ptr_5) << std::endl;

if(int_ptr_4){
    std::cout << "int_ptr_4 pointing to something valid" << std::endl;
}else{
    std::cout << "int_ptr_4 pointing to nullptr" << std::endl;
}
```

make_shared

```
std::shared_ptr<int> int_ptr_6 = std::make_shared<int>(55);
std::cout << "The value pointed to by int_ptr_6 is : " << *int_ptr_6 << std::endl;

std::shared_ptr<Dog> dog_ptr_6 = std::make_shared<Dog>("Salz");
dog_ptr_6->print_info();

std::cout << "int_ptr_6 use count : " << int_ptr_6.use_count() << std::endl;
std::cout << "dog_ptr_6 use count : " << dog_ptr_6.use_count() << std::endl;

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

std::shared_ptr<Dog> dog_ptr_7 {nullptr};
dog_ptr_7 = dog_ptr_6;

std::cout << "int_ptr6 use count : " << int_ptr_6.use_count() << std::endl;
std::cout << "dog_ptr6 use count : " << dog_ptr_6.use_count() << std::endl;
```

Slide intentionally left empty

Creating shared pointers from unique pointers

```cpp
//Create shared pointers from unique_ptrs
std::unique_ptr<int> unique_ptr_int_1 = std::make_unique<int>(22);
std::unique_ptr<Dog> unique_ptr_dog_1 = std::make_unique<Dog>("Halz");

//Create shared pointers from unique_ptrs
//Ownership moves from unique_ptrs to shared_ptrs from now on
std::shared_ptr<int> shared_ptr_int_1 = std::move(unique_ptr_int_1);
std::shared_ptr<Dog> shared_ptr_dog_1 = std::move(unique_ptr_dog_1);
//std::shared_ptr<Dog> shared_ptr_dog_2 = unique_ptr_dog_1; // Direct assignment
                                                            // Doesn't work, you have to do
                                                            // an explicit std::move to move ownership


std::cout << "shared_ptr_int_1 use count : " << shared_ptr_int_1.use_count() << std::endl;
std::cout << "shared_ptr_dog_1 use count : " << shared_ptr_dog_1.use_count() << std::endl;
std::cout << std::boolalpha;
std::cout << "unique_ptr_int_1 : " << static_cast<bool> (unique_ptr_int_1) << std::endl;
std::cout << "unique_ptr_dog_1 : " << static_cast<bool> (unique_ptr_dog_1) << std::endl;
```

shared_ptr to unique_ptr

```cpp
//Can't transform from std::shared_ptr to std::unique_ptr
    //The reason this transformation is disabled isn't hard to think of.
    //          At any given moment, there may be any number of shared pointers
    //          spread through your entire application working on the same object,
    //          If you were to instantly make one of those a unique ptr, what do
    //          you do with the remaining copies?? Unique ptr can't have copies
    //          anyway. So the compiler prevents you from doing this.
std::unique_ptr<int> unique_ptr_illegal_1 {shared_ptr_int_3}; // Compiler error
std::unique_ptr<int> unique_ptr_illegal_2 =shared_ptr_int_3; // Compiler error
std::unique_ptr<int> unique_ptr_illegal_3 =std::move(shared_ptr_int_3); // Compiler error
```

Returning smart pointers

```
//Returning unique_ptr to unique_ptr
std::cout << std::endl;
std::cout << "Returning unique_ptr from function to unique ptr" << std::endl;
std::unique_ptr<Dog> unique_ptr_dog_2 = get_unique_ptr(); // This implicitly moves
                                                // ownership to dog_ptr9_unique
if(unique_ptr_dog_2)
    std::cout << "unique_ptr_dog_2 dog name : " << unique_ptr_dog_2->get_name() << std::endl;

//Returning unique_ptr to shared_ptr
std::cout << std::endl;
std::cout << "Returning unique_ptr from function to shared ptr" << std::endl;
std::shared_ptr<Dog> shared_ptr_dog_4= get_unique_ptr(); // This implicitly moves
                    // ownership to shared_ptr_dog_4. Implicitly does something
                    //like this :
                    // std::shared_ptr<Dog> shared_ptr_dog_4 = std::move(dog_ptr_internal);
                    //moving ownership to a shared pointer whose reference count becomes 1.
if(shared_ptr_dog_4){
    std::cout << "shared_ptr_dog_4 name      : " << shared_ptr_dog_4->get_name() << std::endl;
    std::cout << "shared_ptr_dog_4 use count : " << shared_ptr_dog_4.use_count() << std::endl;

}
```

Having your functions return unique_ptr is the preferred way to do things, as you can turn that pointer into a shared_ptr at any time, but you can't turn a shared_ptr into a unique_ptr. unique_ptr are much more flexible to work with in this case

Slide intentionally left empty

Shared Pointers with arrays

```cpp
//C++17 only : Recommended
std::shared_ptr<int[]> shared_ptr_int_arr_1( new int[10]{1,2,3,4,5,6,7,8,9,1});
std::shared_ptr<Dog[]> shared_ptr_dog_arr_1( new Dog[10]{Dog("Dog1"),Dog("Dog2")});

//Read int array
std::cout << std::endl;
std::cout << "Reading data from arrays" << std::endl;
std::cout << "Reading shared_ptr_int_arr_1: " << std::endl;
for(size_t i{0}; i < 10 ; ++i){
    std::cout << "shared_ptr_int_arr_1[" << i << "] : " << shared_ptr_int_arr_1[i] << std::endl;
}

std::cout << std::endl;
std::cout << "Reading shared_ptr_dog_arr_1: " << std::endl;
for(size_t i{0}; i < 10 ; ++i){
    std::cout << "shared_ptr_dog_arr_1[" << i << "] : " << shared_ptr_dog_arr_1[i].get_name() << std::endl;
}

//Setting elements
shared_ptr_int_arr_1[3] = 28;
shared_ptr_dog_arr_1[1] = Dog("Fluzzy");
```

- make_shared syntax isn't supported yet for raw arrays. Some compilers do offer some partial support for it, but I would not recommend using that in your code so I won't show that here. If you find yourself needing to use shared_ptr with arrays, then new is still your friend. But once the array is created, the shared_ptr is going to manage the memory , you don't need to explicitly call delete.

- You won’t need to use raw arrays with smart pointers that often though, there are better and more practical collection types we will learn about later in the course that almost remove the need for raw arrays.

Slide intentionally left empty

Shared Pointers as function parameters and return value

shared_ptr passed by value

```
void use_dog_v1( std::shared_ptr<Dog> dog){
    std::cout << "shared_ptr passed by value , dog_name : " << dog->get_name() << std::endl;
    std::cout << "use count in use_dog_v1 : " << dog.use_count() << std::endl;
}
```

shared_ptr passed by non const ref

```cpp
void use_dog_v2( std::shared_ptr<Dog> & dog){
    //Since no copy is made, we won't see the reference count increment here
    dog->set_dog_name("Riol");
    //dog.reset(new Dog()); // Passed by non const ref
    std::cout << "shared_ptr passed by non const reference (dog name changed in function)
    std::cout << "use count in use_dog_v2 : " << dog.use_count() << std::endl;
}
```

shared_ptr passed by const ref

```cpp
void use_dog_v3( const std::shared_ptr<Dog> & dog){
     //Since no copy is made, we won't see the reference count increment here
    dog->set_dog_name("Simy"); // We can change the dog object even though
                                // shared_ptr is passed by ref. The const protects the shared_ptr
                                //  object itself, not the pointed to object.
    //dog.reset(new Dog()); // Passed by const ref
    std::cout << "shared_ptr passed by const reference (dog name changed in function) , dog_name
    std::cout << "use count in use_dog_v3 : " << dog.use_count() << std::endl;
}
```

Returning by value

```
//Returning by value
//     Returning a shared_ptr by value goes through return value optimization and at the
//                end no copy is made, we have a single shared ptr with a reference count of 1,
//                just like when we create a shared directly with make_shared.
std::shared_ptr<Dog> get_shared_ptr_v1(){
    std::shared_ptr<Dog> dog_ptr = std::make_shared<Dog>("Internal Dog_v1");
    std::cout << "Managed dog address(in) : " << dog_ptr.get() << std::endl;
    return dog_ptr;
}
```

Returning by Reference

NOT RECOMMENDED!

Slide intentionally left empty

Weak pointers

Non owning pointers that don’t implement the -> or * operator. You can’t use them directly to read or modify data

Using weak_ptr

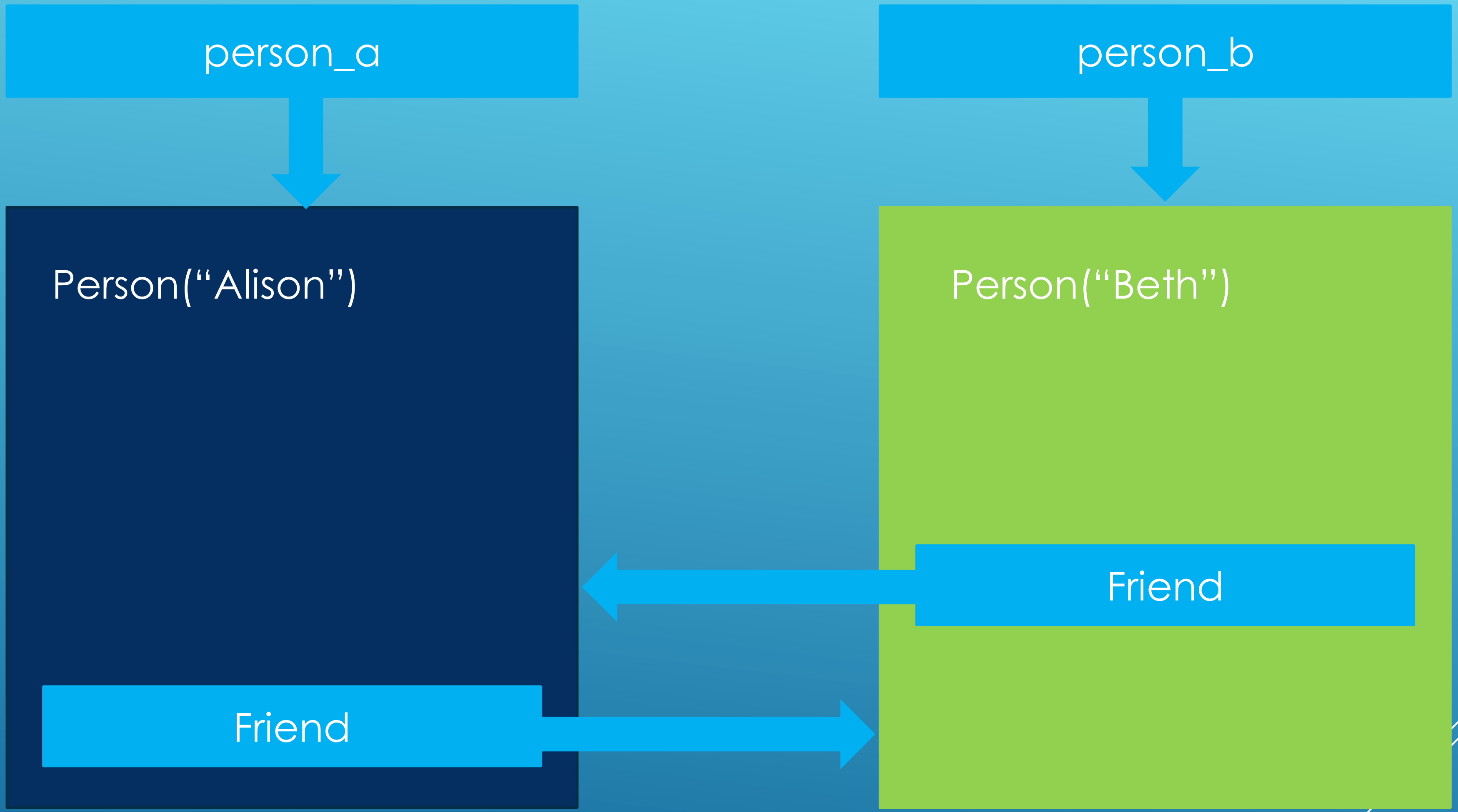
```
//Playing with basic use of weak_ptr
std::shared_ptr<Dog> shared_ptr_dog_1 = std::make_shared<Dog>("Dog1");
std::shared_ptr<int> shared_ptr_int_1 = std::make_shared<int>(200);

std::weak_ptr<Dog>  weak_ptr_dog_1 (shared_ptr_dog_1);
std::weak_ptr<int>  weak_ptr_int_1 (shared_ptr_int_1);


// No * , or -> operators you would expect from regular pointers
//std::cout << "weak_ptr_dog_1 use count : " << weak_ptr_dog_1.use_count() << std::endl;
std::cout << "Dog name : " << weak_ptr_dog_1->get_name() << std::endl; // Compiler error : No -> operator
std::cout << "Pointed to value : " << *weak_ptr_int_1 << std::endl; // Compiler error : No * operator
std::cout << "Pointed to address : " << weak_ptr_dog_1.get() << std::endl; // No get method

// To use a weak ptr you have to turn it into a shared_ptr with the lock method
std::cout << std::endl;
std::shared_ptr<Dog> weak_turned_shared = weak_ptr_dog_1.lock();
std::cout << "weak_turned_shared use count : " << weak_turned_shared.use_count() << std::endl;
std::cout << "Dog name : " << weak_turned_shared->get_name() << std::endl;
std::cout << "Dog name : " << shared_ptr_dog_1->get_name() << std::endl;
```

Cyclic dependency problem

```
class Person
{
public:
    Person() = default;
    Person(std::string name);
    ~Person();

    //Member functions
    void set_friend(std::shared_ptr<Person> p){
        m_friend = p;
    }

private :
    std::shared_ptr<Person> m_friend; // Initialized to nullptr
    std::string m_name {"Unnamed"};

};
```

Cyclic dependency problem

```cpp
//Circular dependencies
std::shared_ptr<Person> person_a = std::make_shared<Person>("Alison");
std::shared_ptr<Person> person_b = std::make_shared<Person>("Beth");

person_a->set_friend(person_b);
person_b->set_friend(person_a);
```



person_a
person_b
Person("Alison")
Person("Beth")
Friend
Friend

Solving cyclic dependency

```
class Person
{
public:
    Person() = default;
    Person(std::string name);
    ~Person();

    //Member functions
    void set_friend(std::shared_ptr<Person> p){
        //The assignment creates a weak_ptr out of p
        m_friend = p;
    }

private :
    std::weak_ptr<Person> m_friend;
    std::string m_name {"Unnamed"};
};
```

Slide intentionally left empty

Smart Pointers

Manually releasing memory yourself through the delete operator for raw pointers is a pain in the neck. Smart pointers are a solution offered by modern C++ to release the memory automatically when the pointer managing the memory goes out of scope

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

Unique pointers

```
Dog * p_dog_3 = new Dog("Dog3");
std::unique_ptr<Dog> up_dog_4{p_dog_3}; // 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<Dog> up_dog_5 {new Dog("Dog5")};
std::unique_ptr<int> up_int2{new int(200)};
std::unique_ptr<Dog> up_dog_6{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.
up_dog_5->print_dog(); // Calling function with -> operator
//Assign to fundamental type
* up_int2 = 500;
std::cout << "Integer is : " << *up_int2 << std::endl; // dereferencing
std::cout << "Integer lives at address : " << up_int2.get() << std::endl;
```

Std::make_unique

```
std::unique_ptr<Dog> up_dog_7 = std::make_unique<Dog>("Dog7");
up_dog_7->print_dog();

std::unique_ptr<int> p_int3 = std::make_unique<int>(30);
*p_int3 =67;
std::cout << "Value pointed to by p_int3 is :"  << *p_int3 << std::endl;
std::cout << "p_int pointing at address :" << p_int3.get() << std::endl;
```

Slide intentionally left empty

Unique pointers as function parameters & return values

Array managed by unique_ptr

```cpp
//Array allocated on the heap with unique_ptr. Releases space for array automatically
{
     std::cout << std::endl;
     std::cout << "Array on heap with unique ptr" << std::endl;

    auto arr_ptr = std::unique_ptr<Dog[]> ( new Dog[3]{Dog("Dog7"), Dog("Dog8") , Dog("Dog9")});

    for (size_t i{0} ; i < 3 ; ++i){
        arr_ptr[i].print_info() ;
    }

}
```


ptr_1
ptr_2
ptr_3
ptr_4
data

Shared pointers with fundamental types

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

std::cout << "The pointed to value is : " << *int_ptr_1 << std::endl;
*int_ptr_1 = 40; // Use the pointer to assign
std::cout << "The pointed to value is : " << *int_ptr_1 << std::endl;
std::cout << "Use count : " << int_ptr_1.use_count() << std::endl;

//Copying
std::cout << std::endl;
std::cout << "Copying..." << std::endl;
std::shared_ptr<int> int_ptr_2 = int_ptr_1; // Use count : 2

std::cout << "The pointed to value is (through int_ptr2)  : " << *int_ptr_2 << std::endl;
*int_ptr_2 = 70;
std::cout << "The pointed to value is (through int_ptr2) : " << *int_ptr_2 << std::endl;

std::cout << "Use count for int_ptr_1 : " << int_ptr_1.use_count() << std::endl;
std::cout << "Use count for int_ptr_2 : " << int_ptr_2.use_count() << std::endl;
```

make_shared

```
std::shared_ptr<int> int_ptr_6 = std::make_shared<int>(55);
std::cout << "The value pointed to by int_ptr_6 is : " << *int_ptr_6 << std::endl;

std::shared_ptr<Dog> dog_ptr_6 = std::make_shared<Dog>("Salz");
dog_ptr_6->print_info();

std::cout << "int_ptr_6 use count : " << int_ptr_6.use_count() << std::endl;
std::cout << "dog_ptr_6 use count : " << dog_ptr_6.use_count() << std::endl;

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

std::shared_ptr<Dog> dog_ptr_7 {nullptr};
dog_ptr_7 = dog_ptr_6;

std::cout << "int_ptr6 use count : " << int_ptr_6.use_count() << std::endl;
std::cout << "dog_ptr6 use count : " << dog_ptr_6.use_count() << std::endl;
```

Shared ptr from unique ptr

```
//Create shared pointers from unique_ptrs
std::unique_ptr<int> unique_ptr_int_1 = std::make_unique<int>(22);
std::unique_ptr<Dog> unique_ptr_dog_1 = std::make_unique<Dog>("Halz");

//Create shared pointers from unique_ptrs
//Ownership moves from unique_ptrs to shared_ptrs from now on
std::shared_ptr<int> shared_ptr_int_1 = std::move(unique_ptr_int_1);
std::shared_ptr<Dog> shared_ptr_dog_1 = std::move(unique_ptr_dog_1);
//std::shared_ptr<Dog> shared_ptr_dog_2 = unique_ptr_dog_1; // Direct assignment
                                                           // Doesn't work, you have to do
                                                           // an explicit std::move to move ownership


std::cout << "shared_ptr_int_1 use count : " << shared_ptr_int_1.use_count() << std::endl;
std::cout << "shared_ptr_dog_1 use count : " << shared_ptr_dog_1.use_count() << std::endl;
std::cout << std::boolalpha;
std::cout << "unique_ptr_int_1 : " << static_cast<bool> (unique_ptr_int_1) << std::endl;
std::cout << "unique_ptr_dog_1 : " << static_cast<bool> (unique_ptr_dog_1) << std::endl;
```

Solving cyclic dependency with weak pointers

```
class Person
{
public:
    Person() = default;
    Person(std::string name);
    ~Person();

    //Member functions
    void set_friend(std::shared_ptr<Person> p){
        //The assignment creates a weak_ptr out of p
        m_friend = p;
    }

private :
    std::weak_ptr<Person> m_friend;
    std::string m_name {"Unnamed"};
};
```

Slide intentionally left empty