

- Exercises

Let's solve a few exercises around CRTP in this lesson.

WE'LL COVER THE FOLLOWING ^

- Problem Statement 1
- Problem Statement 2
- Problem Statement 3
 - Case 1:
 - Case 2:
 - Case 3:
- Problem Statement 4

Problem Statement 1

Extend the given piece of code with a `Person` class. A `Person` should have a

- first name
- last name

You need to create two objects of the `Person` class and compare these objects using relational operators.

```
#include <iostream>
#include <string>

template<class Derived>
class Relational{};

// Relational Operators

template <class Derived>
bool operator > (Relational<Derived> const& op1, Relational<Derived> const & op2){
    Derived const& d1 = static_cast<Derived const&>(op1);
    Derived const& d2 = static_cast<Derived const&>(op2);
    return d2 < d1;
}
```



```
template <class Derived>
bool operator == (Relational<Derived> const& op1, Relational<Derived> const & op2){
    Derived const& d1 = static_cast<Derived const&>(op1);
    Derived const& d2 = static_cast<Derived const&>(op2);
    return !(d1 < d2) && !(d2 < d1);
}
```

```
template <class Derived>
bool operator != (Relational<Derived> const& op1, Relational<Derived> const & op2){
    Derived const& d1 = static_cast<Derived const&>(op1);
    Derived const& d2 = static_cast<Derived const&>(op2);
    return (d1 < d2) || (d2 < d1);
}
```

```
template <class Derived>
bool operator <= (Relational<Derived> const& op1, Relational<Derived> const & op2){
    Derived const& d1 = static_cast<Derived const&>(op1);
    Derived const& d2 = static_cast<Derived const&>(op2);
    return (d1 < d2) || (d1 == d2);
}
```

```
template <class Derived>
bool operator >= (Relational<Derived> const& op1, Relational<Derived> const & op2){
    Derived const& d1 = static_cast<Derived const&>(op1);
    Derived const& d2 = static_cast<Derived const&>(op2);
    return (d1 > d2) || (d1 == d2);
}
```

// Implement a person class here

// Person

// Apple

```
class Apple:public Relational<Apple>{
public:
    explicit Apple(int s): size{s}{};
    friend bool operator < (Apple const& a1, Apple const& a2){
        return a1.size < a2.size;
    }
private:
    int size;
};
```

// Man

```
class Man:public Relational<Man>{
public:
    explicit Man(const std::string& n): name{n}{}
    friend bool operator < (Man const& m1, Man const& m2){
        return m1.name < m2.name;
    }
private:
    std::string name;
};
```

```
int main(){
```

```
    std::cout << std::boolalpha << std::endl;
```

```
    // Call Person class object here checks them for all relational operators
```

```
std::cout << std::endl;

}
```



Problem Statement 2

In [example 1](#), how can you prevent a derived class which has the wrong template parameter: `Derived4: Base<Derived3>`

```
// templateCRTP.cpp

#include <iostream>

template <typename Derived>
struct Base{
    void interface(){
        static_cast<Derived*>(this)->implementation();
    }
    void implementation(){
        std::cout << "Implementation Base" << std::endl;
    }
};

struct Derived1: Base<Derived1>{
    void implementation(){
        std::cout << "Implementation Derived1" << std::endl;
    }
};

struct Derived2: Base<Derived2>{
    void implementation(){
        std::cout << "Implementation Derived2" << std::endl;
    }
};

struct Derived3: Base<Derived3>{};

template <typename T>
void execute(T& base){
    base.interface();
}

// Write the struct here

int main(){

    std::cout << std::endl;

    // call the function here

    std::cout << std::endl;
```

```
}
```



Problem Statement 3

The functionality of the program in [example 1](#) of the previous lesson can be implemented in various ways. Implement each variant.

Case 1:

Object-oriented with dynamic polymorphism.

```
#include <iostream>

// Implement the functionality here

int main() {
    // call the function here
    return 0;
}
```



Case 2:

Just a function template.

```
#include <iostream>

// write your function template here

int main() {
    // Call the function here
}
```



Case 3:

To solve this exercise with [Concepts](#), you have to peek into the last chapter of this course. Solving this exercise with [Concepts](#) is also a valid and elegant way.

Concepts with C++20:

```
#include <iostream>

// Implement it using C++20 concept

int main() {
    // your code goes here
}
```



Problem Statement 4

Implement the class `ShareMe` and use it.

- Objects of the class `ShareMe` should return an `std::shared_ptr` to themselves.

```
#include <iostream>

// Implement the class here

int main() {
    // After implementing the shareMe class uncomment the code given below

    /*
    std::cout << std::endl;

    // share the same ShareMe object
    std::shared_ptr<ShareMe> shareMe(new ShareMe);
    std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();

    // both resources have the same address
    std::cout << "Address of resource of shareMe "<< (void*)shareMe.get() << " " << std::endl;
    std::cout << "Address of resource of shareMe1 "<< (void*)shareMe1.get() << " " << std::endl;

    // the use_count is 2
    std::cout << "shareMe.use_count(): "<< shareMe.use_count() << std::endl;
    std::cout << std::endl;
    */
}
```



In the next lesson, we'll look at solutions to these exercises.

