- Solutions

Let's look at the solutions of the exercises, we had in the last lesson.

WE'LL COVER THE FOLLOWING

Solution of Problem Statement 1

```
// templateCRTPRelational.cpp

#include <iostream>
#include <string>
#include <utility>

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;
```

```
    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);
}

// 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;
};

// class Person
class Person: public Relational<Person>{
public:
    Person(std::string fst, std::string sec): first(fst), last(sec){}
    friend bool operator < (Person const& p1, Person const& p2){
        return std::make_pair(p1.first, p2.last) < std::make_pair(p2.first, p2.last);
    }
private:
```

```
    std::string first;
    std::string last;
};


int main(){

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

  Apple apple1{5};
  Apple apple2{10};
  std::cout << "apple1 < apple2: " << (apple1 < apple2) << std::endl;
  std::cout << "apple1 > apple2: " << (apple1 > apple2) << std::endl;
  std::cout << "apple1 == apple2: " << (apple1 == apple2) << std::endl;
  std::cout << "apple1 != apple2: " << (apple1 != apple2) << std::endl;
  std::cout << "apple1 <= apple2: " << (apple1 <= apple2) << std::endl;
  std::cout << "apple1 >= apple2: " << (apple1 >= apple2) << std::endl;

  std::cout << std::endl;

  Man man1{"grimm"};
  Man man2{"jaud"};
  std::cout << "man1 < man2: " << (man1 < man2) << std::endl;
  std::cout << "man1 > man2: " << (man1 > man2) << std::endl;
  std::cout << "man1 == man2: " << (man1 == man2) << std::endl;
  std::cout << "man1 != man2: " << (man1 != man2) << std::endl;
  std::cout << "man1 <= man2: " << (man1 <= man2) << std::endl;
  std::cout << "man1 >= man2: " << (man1 >= man2) << std::endl;

  std::cout << std::endl;

  Person rainer{"Rainer", "Grimm"};
  Person marius{"Marius", "Grimm"};
  std::cout << "rainer < marius: " << (rainer < marius) << std::endl;
  std::cout << "rainer > marius: " << (rainer > marius) << std::endl;
  std::cout << "rainer == marius: " << (rainer == marius) << std::endl;
  std::cout << "rainer != marius: " << (rainer != marius) << std::endl;
  std::cout << "rainer <= marius: " << (rainer <= marius) << std::endl;
  std::cout << "rainer >= marius: " << (rainer >= marius) << std::endl;

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

Explanation

In the above code, we have defined the `Person` class which contains the string variables, i.e., `first` and `last` and a `<` operator to compare two people's lengths in lines 75 – 77. The class `Person` is publicly derived (line 72) from the class `Relational<Person>`. We have implemented for classes of the kind `Relational` the greater than operator `>` (lines 12 - 17), the equality operator `==` (lines 19 - 24), the not equal operator `!=` (lines 26 - 31), the less than or equal operator `<=` (line 33 - 38) and the greater than or equal operator `>=`

(lines 40 - 45). The less than or equal and greater than or equal operators used the `equality` operator (line 37 and 44). All these operators convert their operands: `Derived const&: Derived const& d1 = static_cast<Derived const&>(op1)`.

In the main program, we have compared `Apple`, `Man`, and `Person` classes for all the above-mentioned operators.

Solution of Problem Statement 2

```
// templateCRTPCheck.cpp

#include <iostream>

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

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>{};

// struct Derived4: Base<Derived3>{};

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

int main(){

  std::cout << std::endl;

  Derived1 d1;
  execute(d1);

  Derived2 d2;
```

```
  Derived2 d2;
  execute(d2);

  Derived3 d3;
  execute(d3);

  // Derived4 d4;

  std::cout << std::endl;

}
```

Explanation

We have used static polymorphism in the function template `execute` (lines 34 – 37). We invoked the method `base.interface` on each base argument. The method `Base::interface`, in lines (7 – 9), is the key point of the `CRTP` idiom. The method dispatches to the implementation of the derived class: `static_cast<Derived*>(this)->implementation()`. That is possible because the method will be instantiated when called. At this point in time, the derived classes `Derived1`, `Derived2`, and `Derived3` are fully defined. Therefore, the method `Base::interface` can use the details of its derived classes. Especially interesting is the method `Base::implementation` (lines 10 – 12). This method plays the role of a default implementation for the static polymorphism for the class `Derived3` (line 30).

The constructor of the derived class has to call the constructor of the base class. The constructor in the base class is private. Only type `T` can invoke the constructor of the base class. So, if the derived class is different from the class `T`, the code doesn't compile.

Solution to Problem Statement 3 (Case 1)

```
// dispatchPolymorphism.cpp

#include <iostream>

struct Base{
    virtual void interface(){
        std::cout << "Implementation Base" << std::endl;
    }
};

struct Derived1: Base{
    virtual void interface(){
```

```
        std::cout << "Implementation Derived1" << std::endl;
    }
};

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

struct Derived3: Base{};

void execute(Base& base){
    base.interface();
}

int main(){

    std::cout << std::endl;

    Derived1 d1;
    Base& b1 = d1;
    execute(b1);

    Derived2 d2;
    Base& b2 = d2;
    execute(b2);

    Derived3 d3;
    Base& b3 = d3;
    execute(b3);

    std::cout << std::endl;

}
```

Explanation

We have used dynamic polymorphism in the function template `execute` (lines 25 – 27). This function only accepts a reference to the class object passed. Now, we make objects of the derived classes and store references to them (lines 34, 38 and 42). The static type of `b1`, `b2`, and `b3` in lines 34, 38, 42 is `Base` and the dynamic type is `Dervived1`, `Derived2`, or `Derived3` respectively.

Solution to Problem Statement 3 (Case 2)

```
// dispatchGeneric.cpp

#include <iostream>

struct Base{
```

```
struct Base{
    virtual void interface(){
        std::cout << "Implementation Base" << std::endl;
    }
};

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

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

struct Derived3: Base{};

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

int main(){

    std::cout << std::endl;

    Derived1 d1;
    execute(d1);

    Derived2 d2;
    execute(d2);

    Derived3 d3;
    execute(d3);

    std::cout << std::endl;

}
```

Explanation

We have used static polymorphism in the function template `execute` (line 25 – 27). The method `execute` accepts a reference. Now, we can make objects of the derived classes and pass them to `execute` methods (line 34 – 41).

Solution to Problem Statement 3 (Case 3)

```
// dispatchConcepts.cpp

#include <iostream>
```

```
template <typename T>
concept bool Interface(){

    return requires(T a){
        { a.interface } -> void;
    };
}

struct Base{
    virtual void interface(){
        std::cout << "Implementation Base" << std::endl;
    }
};

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

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

struct Derived3: Base{};

template <typename Interface>
void execute(Interface& inter){
    inter.interface();
}

int main(){

    std::cout << std::endl;

    Derived1 d1;
    execute(d1);

    Derived2 d2;
    execute(d2);

    Derived3 d3;
    execute(d3);

    std::cout << std::endl;

}
```

Explanation

This implementation is based on concepts which will be part of C++20. In the concrete case, concepts mean that `execute` can only be invoked with types, which supports a function with the name `interface` returning `void`.

which supports a function with the name `interface` returning `void`.

Solution to Problem Statement 4

```
// templatesCRTPShareMe.cpp

#include <iostream>
#include <memory>

class ShareMe: public std::enable_shared_from_this<ShareMe>{
public:
  std::shared_ptr<ShareMe> getShared(){
    return shared_from_this();
  }
};

int main(){

  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;

}
```

Explanation

With the class std::enable_shared_from_this, we can create objects which return a `std::shared_ptr` on itself. For that, we have to derive the class public from `std::enable_shared_from_this`. The smart pointer `shareMe` (line 18) is copied by `shareMe1` (line 19). The call `shareMe->getShared()` in line 19 creates a new smart pointer. `getShared()` (line 8) internally uses the function `shared_from_this`. In lines 22 and 23, `shareMe.get()` and `shareMe.get()` returns a pointer to the resource. In line 26, the `shareMe.use_count()` returns the value of the reference counter.

In the next lesson, we'll learn about expression templates.