- Examples

In this lesson, we'll see a couple of examples of CRTP.

WE'LL COVER THE FOLLOWING

Example 1: Mixins with CRTP

```
// templateCRTPRelational.cpp

#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);
}

// 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;

  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;

}
```

Explanation

We have implemented, for the classes `Apple` and `Man`, the smaller operator separately (lines 51-52 and 63-64). The classes `Man` and `Apple` are publicly derived (line 48 and 60) from the class `Relational<Man>` and `Ralational<Apple>`. We have implemented for classes of the kind `Relational` the greater than operator `>` (lines 11 – 16), the equality operator `==` (lines 18 – 23), the not equal operator `!=` (lines 25 – 30), the less than or equal operator `<=` (line 32 – 37) and the greater than or equal operator `>=` (lines 39 – 44). The less than or equal (`<=`) and greater than or equal (`>=`) operators used the equality operator == (line 36 and 43). All these operators convert their operands: `Derived const&: Derived const& d1 = static_cast<Derived const&>(op1)`.

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

Example 2: Static Polymorphism with CRTP

```
// 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(){
```

```
  void implementation(){
    std::cout << "Implementation Derived2" << std::endl;
  }
};

struct Derived3: Base<Derived3>{};

template <typename T>
void execute(T& base){
  base.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` (lines 30-33). 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 methods dispatch 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 11-13). This method plays the role of a default implementation for the static polymorphism for the class `Derived3` (line 28).

We'll solve a few exercises around CRTP in the next lesson.