- Examples

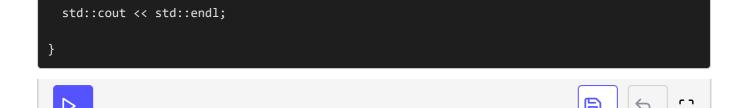
In this lesson, we'll see a couple of examples of CRTP.

WE'LL COVER THE FOLLOWING Example 1: mixins with CRTP Explanation Example 2: static polymorphism with CRTP Explanation

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) \mid \mid (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) \mid | (d1 == d2);
}
// Apple
class Apple:public Relational<Apple>{
    explicit Apple(int s): size{s}{};
    friend bool operator < (Apple const& a1, Apple const& a2){</pre>
        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){</pre>
        return m1.name < m2.name;</pre>
private:
    std::string name;
};
int main(){
  std::cout << std::boolalpha << std::endl;</pre>
  Apple apple1{5};
  Apple apple2{10};
  std::cout << "apple1 < apple2: " << (apple1 < apple2) << std::endl;</pre>
  std::cout << "apple1 > apple2: " << (apple1 > apple2) << std::endl;</pre>
  std::cout << "apple1 == apple2: " << (apple1 == apple2) << std::endl;</pre>
  std::cout << "apple1 != apple2: " << (apple1 != apple2) << std::endl;</pre>
  std::cout << "apple1 <= apple2: " << (apple1 <= apple2) << std::endl;</pre>
  std::cout << "apple1 >= apple2: " << (apple1 >= apple2) << std::endl;</pre>
  std::cout << std::endl;</pre>
  Man man1{"grimm"};
  Man man2{"jaud"};
  std::cout << "man1 < man2: " << (man1 < man2) << std::endl;</pre>
  std::cout << "man1 > man2: " << (man1 > man2) << std::endl;</pre>
  std::cout << "man1 == man2: " << (man1 == man2) << std::endl;</pre>
  std::cout << "man1 != man2: " << (man1 != man2) << std::endl;</pre>
  std::cout << "man1 <= man2: " << (man1 <= man2) << std::endl;</pre>
  std::cout << "man1 >= man2: " << (man1 >= man2) << std::endl;</pre>
```



Explanation

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

Example 2: static polymorphism with CRTP

```
// templateCRTP.cpp
                                                                                            G
#include <iostream>
template <typename Derived>
struct Base{
  void Interface(){
    static_cast<Derived*>(this)->implementation();
  void implementation(){
    std::cout << "Implementation Base" << std::endl;</pre>
};
struct Derived1: Base<Derived1>{
  void implementation(){
    std::cout << "Implementation Derived1" << std::endl;</pre>
};
struct Derived2: Base<Derived2>{
  void implementation(){
```

```
std::cout << "Implementation Derived2" << std::endl;</pre>
};
struct Derived3: Base<Derived3>{};
template <typename T>
void execute(T& base){
  base.Interface();
int main(){
  std::cout << std::endl;</pre>
  Derived1 d1;
  execute(d1);
  Derived2 d2;
  execute(d2);
  Derived3 d3;
  execute(d3);
  std::cout << std::endl;</pre>
```

Explanation

We have used static polymorphism in the function template execute (lines 30-33). We then invoked the method base.Interface on each base argument. These 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() which 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. The method Base::implementation (lines 11-13) is especially interesting. This method works as the default implementation for the static polymorphism for the class Derived3 (line 28).

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