Curiously **R**ecurring **T**emplate **P**attern (aka CRTP)

By: Avi Lachmish

Agenda

- What is CRTP
- Why do we need it
- What could go wrong
- Operator implementation
- Main approaches
- references

What is CRTP

- A basic example

```
template <typename Derived> class CuriousBase
{ … };

class Curious : public CuriousBase <Curious>
{ … };
```

What is CRTP - continue

```
template <typename Derived> class CuriousBase
{
public:
   void foo() {
       Derived & derived = static_cast< Derived &>(*this);
      use derived...
   }
};

class Curious : public CuriousBase <Curious> { ... };
```

Why do we need it

- **static polymorphism** - alternative to **virtual**, avoids memory and execution time overhead.

- ***Using expression templates*** - computes expressions only when needed, removes loops and copies

Why do we need it - continue

Dynamic Polymorphism

```
struct shape {
  virtual std::string draw() = 0;
};
struct rectangle : shape{
  std::string draw() override {
    return “rectangle”;
  }
};
```

Static Polymorphism

```
template<typename T>
struct shape {
  std::string draw() {
    return static_cast<T*>(this)->draw();
};
struct rectangle : shape<rectangle>{
  std::string draw() {
    return “rectangle”;
  }
};
```

Why do we need it - continue

Dynamic Polymorphism

```
void draw(shape &s) {
  std::cout << s.draw();
}
Rectangle r;
draw(&r);
```

Static Polymorphism

```
template<typename T>
void draw(shape<T> &s) {
  std::cout << s.draw();
}
Rectangle r;
draw(&r);
```

- Of course, this comes with some **limitations** in the flexibility of static polymorphism.
  E.g. different CRTP-Derived classes cannot be addressed with a common Base pointer/reference!

Why do we need it - continue

Dynamic Polymorphism

- resolved at run time (dynamic binding) using vptr and vtable
- base class is abstract if any of its functions is a pure virtual
- memory cost to store vptr in each object, can be significant for small classes
- time cost for dynamic dispatch at every virtual function call, no inlining
- very flexible: pass base pointer/reference to a function, iterate over arrays of base pointers/references, ...

Static Polymorphism

- resolved at compile time (early binding) using CRTP templates resolution
- base class is a templated class, its methods call the ones of its derived class instantiation (static_cast<T*>(this)->call();)
- no memory overhead
- no time overhead, possible inlining for optimization
- limited flexibility: no addressing via Base pointers/references

What could go wrong – bad creation

```
class Derived1 : public Base<Derived1>
{
    ...
};

class Derived2 : public Base<Derived1> // bug in this line of code
{
    ...
};
```

What could go wrong – fix bad creation

```
template <typename T>
class Base
{
public:
    // ...
private:
    Base(){};
    friend T;
};
```

What could go wrong – method hidden

- risk with CRTP is that methods in the derived class will **hide** methods from the base class with the same name since we are not using virtual in this design pattern.

What could go wrong – incomplete type

```
template <typename DerivedT> class CuriousBase
{
public:
    using derived_type = typename DerivedT::type; // this is an error
    void foo() {
         DerivedT & derived = static_cast< DerivedT &>(*this);
        derived.bar(); //this is fine
    }
};

class Curious : public CuriousBase <Curious> {
            using type = int;
            void bar();
};
```

Corner cases - Operator implementation

- The Barton-Nackman trick

```
template<typename T>
class Array {
  static bool areEqual(Array<T> const& a, Array<T> const& b) ;

public:
  friend bool operator==(Array const& a, Array const& b) {
    return areEqual(a, b);
  }
};
```

The Barton-Nackman trick

```
template<typename Derived>
class EqualityCompareable {
public:
  friend bool operator!=(Derived const& x1, Derived const& x2) { return !(x1==x2); }
};

Class X : public EqualityCompareable<X> {
public:
 friend bool operator==(X const& x1, X const& x2) {…}
};
```

Main approaches

- **Façade -** Adding functionality, some classes provide generic functionality, that can be re-used by many other classes. The derive class is representing an interface used by the base class reused code.

- **Static interfaces -** In this case, the base class does represent the interface and the derived one does represent the implementation, as usual with polymorphism. But the difference with traditional polymorphism is that there is no virtual involved and all calls are resolved during compilation.

Main approaches - Facade

```
template <typename T>
struct NumericalFunctions {
  void scale(double multiplicator) {
      T& underlying = static_cast<T&>(*this);
      underlying.setValue(underlying.getValue() * multiplicator);
  }
  void square()  {
    T& underlying = static_cast<T&>(*this);
    underlying.setValue(underlying.getValue() * underlying.getValue());
  }
  void setToOpposite() {
    scale(-1);
  }
};
```

Main approaches – Façade continue

```
class Density : public NumericalFunctions<Density>
{
public:
    double getValue() const;
    void setValue(double value);
    ...
};
```

Main approaches – static interface

```
template <typename T>
class Amount
{
public:
    double getValue() const
    {
        return static_cast<T const&>(*this).getValue();
    }
};
```

Main approaches – static interface continue

```
class Constant42 : public Amount<Constant42> {
public:
    double getValue() const {return 42;}
};

class Variable : public Amount<Variable> {
public:
    explicit Variable(int value) : value_(value) {}
    double getValue() const {return value_;}
private:
    int value_;
};
```

Main approaches – static interface continue

```
template<typename T>
void print(Amount<T> const& amount)
{
    std::cout << amount.getValue() << '\n';
}
```

References

- Boost operators library https://www.boost.org/doc/libs/1_65_1/boost/operators.hpp
- https://www.fluentcpp.com/posts/
- C++ template the complete guide (book)