Curiously recurring template pattern

From Wikipedia, the free encyclopedia

The **curiously recurring template pattern** (**CRTP**) is an idiom in C++ in which a class `X` derives from a class template instantiation using `X` itself as template argument.[1] More generally it is known as **F-bound polymorphism**, and it is a form of *F*-bounded quantification.

Contents

[hide]

History[edit]

The technique was formalized in the 1980s as "*F*-bounded quantification."[2] The name "CRTP" was independently coined by Jim Coplien in 1995,[3] who had observed it in some of the earliest C++ template code as well as in code examples that Timothy Budd created in his multiparadigm language Leda.[4] It is sometimes called "Upside-Down Inheritance"[5][6] due to the way it allows class hierarchies to be extended by substituting different base classes.

General form[edit]

```
// The Curiously Recurring Template Pattern (CRTP)
template<class T>
class Base
{
```

```
    // methods within Base can use template to access members of
Derived
};
class Derived : public Base<Derived>
{
    // ...
};
```

Some use cases for this pattern are static polymorphism and other metaprogramming techniques such as those described by Andrei Alexandrescu in *Modern C++ Design*.[7] It also figures prominently in the C++ implementation of the Data, Context and Interaction paradigm.[8]

Static polymorphism[edit]

Typically, the base class template will take advantage of the fact that member function bodies (definitions) are not instantiated until long after their declarations, and will use members of the derived class within its own member functions, via the use of a cast; e.g.:

```
template <class T>
struct Base
{
    void implementation()
    {
        // ...
        static_cast<T*>(this)->implementation();
        // ...
    }

    static void static_func()
    {
        // ...
        T::static_sub_func();
        // ...
```

```
    }
};

struct Derived : Base<Derived>
{
    void implementation();
    static void static_sub_func();
};
```

In the above example, note in particular that the function Base<Derived>::implementation(), though *declared* before the existence of the struct Derived is known by the compiler (i.e., before Derived is declared), is not actually *instantiated* by the compiler until it is actually *called* by some later code which occurs *after* the declaration of Derived (not shown in the above example), so that at the time the function "implementation" is instantiated, the declaration of Derived::implementation() is known.

This technique achieves a similar effect to the use of virtual functions, without the costs (and some flexibility) of dynamic polymorphism. This particular use of the CRTP has been called "simulated dynamic binding" by some.[9] This pattern is used extensively in the Windows ATL and WTL libraries.

To elaborate on the above example, consider a base class with **no virtual functions**. Whenever the base class calls another member function, it will always call its own base class functions. When we derive a class from this base class, we inherit all the member variables and member functions that weren't overridden (no constructors or destructors). If the derived class calls an inherited function which then calls another member function, that function will never call any derived or overridden member functions in the derived class.

However, if base class member functions use CRTP for all member function calls, the overridden functions in the derived class will be selected at compile time. This effectively emulates the virtual function call system at compile time without the costs in size or function call overhead (VTBL structures, and method lookups, multiple-inheritance VTBL machinery) at the disadvantage of not being able to make this choice at runtime.

Object counter[edit]

The main purpose of an object counter is retrieving statistics of object creation and destruction for a given class.[10] This can be easily solved using CRTP:

```
template <typename T>
struct counter
{
    static int objects_created;
    static int objects_alive;

    counter()
    {
        ++objects_created;
        ++objects_alive;
    }

    counter(const counter&)
    {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~counter() // objects should never be removed through pointers of
this type
    {
        --objects_alive;
    }
};
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X>
{
    // ...
```

```
};


class Y : counter<Y>
{
    // ...
};
```

Each time an object of class `X` is created, the constructor of `counter<X>` is called, incrementing both the created and alive count. Each time an object of class `X` is destroyed, the alive count is decremented. It is important to note that `counter<X>` and `counter<Y>` are two separate classes and this is why they will keep separate counts of `X`'s and `Y`'s. In this example of CRTP, this distinction of classes is the only use of the template parameter (`T` in `counter<T>`) and the reason why we cannot use a simple un-templated base class.

Polymorphic chaining[edit]

Method chaining, also known as named parameter idiom, is a common syntax for invoking multiple method calls in object-oriented programming languages. Each method returns an object, allowing the calls to be chained together in a single statement without requiring variables to store the intermediate results.

When the named parameter object pattern is applied to an object hierarchy, things can get wrong. Suppose we have such a base class:

```
class Printer
{
public:
     Printer(ostream& pstream) : m_stream(pstream) {}

     template <typename T>
     Printer& print(T&& t) { m_stream << t; return *this; }

     template <typename T>
     Printer& println(T&& t) { m_stream << t << endl; return *this; }
```

```
private:
     ostream& m_stream;
};
```

Prints can be easily chained:

```
Printer{myStream}.println("hello").println(500);
```

However, if we define the following derived class:

```
class CoutPrinter : public Printer
{
public:
    CoutPrinter() : Printer(cout) {}

    CoutPrinter& SetConsoleColor(Color c) { ... return *this; }
};
```

we "lose" the concrete class as soon as we invoke a function of the base:

```
                        v-- we have a 'Printer' here, not a
'CoutPrinter'
CoutPrinter().print("Hello
").SetConsoleColor(Color.red).println("Printer!"); // compile error
```

This happens because 'print' is a function of the base - 'Printer' - and then it returns a 'Printer' instance.

The CRTP can be used to avoid such problem and to implement "Polymorphic chaining":[11]

```
// Base class
template <typename ConcretePrinter>
class Printer
{
```

```
public:
     Printer(ostream& pstream) : m_stream(pstream) {}

     template <typename T>
     ConcretePrinter& print(T&& t)
     {
         m_stream << t;
         return static_cast<ConcretePrinter&>(*this);
     }

     template <typename T>
     ConcretePrinter& println(T&& t)
     {
         m_stream << t << endl;
         return static_cast<ConcretePrinter&>(*this);
     }
private:
     ostream& m_stream;
};

// Derived class
class CoutPrinter : public Printer<CoutPrinter>
{
public:
    CoutPrinter() : Printer(cout) {}

    CoutPrinter& SetConsoleColor(Color c) { ... return *this; }
};

// usage
 CoutPrinter().print("Hello
").SetConsoleColor(Color.red).println("Printer!");
```

Polymorphic copy construction[edit]

When using polymorphism, one sometimes needs to create copies of objects by the base class pointer. A commonly used idiom for this is adding a virtual clone function that is defined in every derived class. The CRTP can be used to avoid having to duplicate that function or other similar functions in every derived class.

```
// Base class has a pure virtual function for cloning
class Shape {
public:
    virtual ~Shape() {};
    virtual Shape *clone() const = 0;
};
// This CRTP class implements clone() for Derived
template <typename Derived>
class Shape_CRTP : public Shape {
public:
    virtual Shape *clone() const {
        return new Derived(static_cast<Derived const&>(*this));
    }
};

// Nice macro which ensures correct CRTP usage
#define Derive_Shape_CRTP(Type) class Type: public Shape_CRTP<Type>

// Every derived class inherits from Shape_CRTP instead of Shape
Derive_Shape_CRTP(Square) {};
Derive_Shape_CRTP(Circle) {};
```

This allows obtaining copies of squares, circles or any other shapes by `shapePtr->clone()`.

Pitfalls[edit]

One issue with static polymorphism is that without using a general base class like "Shape" from the above example, derived classes cannot be stored homogeneously as each CRTP base class is a

unique type. For this reason, it is more common to inherit from a shared base class with a virtual destructor, like the example above.

See also[edit]

- Barton–Nackman trick
- F-bounded quantification

References[edit]

1. **Jump up^** *Abrahams, David; Gurtovoy, Aleksey. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley. ISBN 0-321-22725-5.*
2. **Jump up^** *William Cook; et al. (1989). "F-Bounded Polymorphism for Object-Oriented Programming" (PDF).*
3. **Jump up^** *Coplien, James O. (February 1995). "Curiously Recurring Template Patterns" (PDF). C++ Report: 24–27.*
4. **Jump up^** *Budd, Timothy (1994). Multiparadigm programming in Leda. Addison-Wesley. ISBN 0-201-82080-3.*
5. **Jump up^** *"Apostate Café: ATL and Upside-Down Inheritance". 15 March 2006. Archived from the original on 15 March 2006. Retrieved 2016-10-09.*
6. **Jump up^** *"ATL and Upside-Down Inheritance". 4 June 2003. Archived from the original on 4 June 2003. Retrieved 2016-10-09.*
7. **Jump up^** *Alexandrescu, Andrei (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley. ISBN 0-201-70431-5.*
8. **Jump up^** *Coplien, James; Bjørnvig, Gertrud (2010). Lean Architecture: for agile software development. Wiley. ISBN 0-470-68420-8.*
9. **Jump up^** *"Simulated Dynamic Binding". 7 May 2003. Retrieved 13 January 2012.*

10. **Jump up^** *Meyers, Scott (April 1998). "Counting Objects in C++". C/C++ Users Journal.*
11. **Jump up^** *Arena, Marco. "Use CRTP for polymorphic chaining". Retrieved 15 March 2017.*