

Curiously Recurring Template Pattern (CRTP) in C++

by grammer jtp


General metrics

5,236	731	106	2 min 55 sec	5 min 37 sec
characters	words	sentences	reading time	speaking time

Writing Issues

 No issues found

Plagiarism

 This text seems 100% original. Grammarly found no matching text on the Internet or in ProQuest's databases.

Unique Words

Measures vocabulary diversity by calculating the percentage of words used only once in your document

27%unique words

Rare Words

Measures depth of vocabulary by identifying words that are not among the 5,000 most common English words.

33%rare words

Word Length

Measures average word length

5.3characters per word

Sentence Length

Measures average sentence length

6.9words per sentence

Curiously Recurring Template Pattern (CRTP) in C++

Curiously Recurring Template Pattern (CRTP) in C++

Curiously Recurring Template Pattern is a programming technique that uses template-based inheritance to achieve static polymorphism. In this pattern, a base class template is parameterized by a derived class, where the derived class inherits from the class using itself as a template argument. By this, a base class can access and manipulate members of the derived class at compile time.

This pattern is used while implementing design patterns such as the singleton, visitor and factor patterns. It is also used to provide extensibility for maintaining library performance. It avoids using virtual function dispatch to enable static polymorphism.

This pattern is mainly used when a base class wants to access and modify the variables and member functions of the derived class. For example, there is a base class template vehicle and one derived class car. The template

has a drive method, which accesses the driveAction method, which is implemented in the derived class. In the main function, we create an object for the car, and this method is called the drive method. This invokes driveAction method. In this way, CRTP allows the base class to access the derived classes at compile time, which enables state polymorphism.

Example program for implementing Curiously Recurring Template Pattern

```
#include <iostream>

// Base class template using CRTP
using namespace std;
template <typename Derived>
class Shape {
public:
    void printArea() {
        cout << "Area: " << static_cast<Derived*>(this)-
>calculateArea() << endl;
    }
};

class Circle: public Shape <Circle> {
protected:
```

```
    double radius;

public:
    Circle(double r) : radius(r) {}
    double calculateArea() {
        return 3.14 * radius * radius;
    }
};

class Rectangle: public Shape <Rectangle> {
protected:
    double width;
    double height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double calculateArea() {
        return width * height;
    }
};

int main() {
    // Creating objects of Circle and Rectangle
    Circle circle(5);
    Rectangle rectangle(4, 6);
    circle.printArea();
```

```
    rectangle.printArea();  
    return 0;  
}
```

Output:

Explanation:

The above program implements CRTP. It has a base class template named Shape and Shape two derived classes, Circle and Rectangle. When the base class, Shape, tries to call the calculateArea() function, it doesn't know the specific implementation of this function in the children. This is possible only through CRTP. By passing the Circle as a template argument, the base class can call the functions of its children.

Applications of using CRTP

This pattern is used in library design and mathematical and scientific computing. CRTP is also used to develop frameworks and engines that are helpful in game development and embedded systems. CRTP is frequently used in template metaprogramming to reduce runtime

overhead. This pattern is also used in code generation tasks.

An example program which is using CRTP for event handling system

```
#include <iostream>

using namespace std;

template <typename Derived>
class EventHandler {
public:
    void handleEvent() {
        static_cast<Derived*>(this)->processEvent();
    }
};

class MouseEventHandler: public
EventHandler<MouseEventHandler> {
public:
    void processEvent() {
        cout << "Mouse event handled" << endl;
    }
};

int main() {
    MouseEventHandler mouseHandler;
```

```
    mouseHandler.handleEvent();  
    return 0;  
}
```

Output:

Explanation:

In the above program, the base class template is "EventHandler," and the derived class is "MouseEventHandler". This derived class implements the "processEvent" function. When we create an object of the derived class "MouseEventHandler" and call the "handleEvent," it calls the "processEvent" function. This is possible by passing "MouseEventHandler" as a template argument to the base class "EventHandler". So, the base class used this information to call the functions of its children.

Example program for Curiously Recurring Templating Singleton Pattern

```
#include <iostream>  
  
using namespace std;  
  
// Base class using CRTP for Singleton pattern  
template <typename Derived>
```



```
class Singleton {
public:
    static Derived& getInstance() {
        static Derived instance;
        return instance;
    }
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
protected:
    Singleton() {}
};

// Derived class implementing Singleton pattern
class MySingleton : public Singleton<MySingleton> {
public:
    void doSomething() {
        cout << "Singleton instance doing something..." <<
endl;
    }
};

int main() {
    MySingleton& instance = MySingleton::getInstance();
```

```
    instance.doSomething(); // Accessing singleton  
instance using CRTP  
    return 0;  
}
```

Output:

Explanation:

This program creates a single instance of a class using CRTP. Only one instance of the class exists throughout the program's execution. Here, "Singleton" is the base class template, and "MySingleton" is the derived class. This derived class has a "doSomething" method. In the main function, we get a single instance of "MySingleton" using the getInstance() method, and this instance is used to call the "doSomething" method.

- | | | |
|--|--|-------------|
| 1. <i>the number of trailing zeros in the binary representation of the</i> | std::builtins::u32 - Alumina Docs
https://docs.alumina-lang.net/std/builtins/u32.html | Originality |
|--|--|-------------|