Curiously recurring template pattern (CRTP)

Background:

It is recommended to refer Virtual Functions and Runtime Polymorphism as a prerequisite of this. Below is an example program to demonstrate run time polymorphism.

```
// A simple C++ program to demonstrate run-time
// polymorphism
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
    Dimension(int _X, int _Y) {mX = _X;  mY = _Y; }
private:
    int mX, mY;
};

// Base class for all image types
class Image
{
public:
    virtual void Draw() = 0;
    virtual Dimension GetDimensionInPixels() = 0;
protected:
    int dimensionX;
    int dimensionY;
};

// For Tiff Images
class TiffImage : public Image
{
public:
    void Draw() { }
    Dimension GetDimensionInPixels() {
        return Dimension(dimensionX, dimensionY);
    }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc
```

```
// Driver code that calls virtual function
int main()
{
    // An image type
    Image* pImage = new TiffImage;

    // Store time before virtual function calls
    auto then = Clock::now();

    // Call Draw 1000 times to make sure performance
    // is visible
    for (int i = 0; i < 1000; ++i)
        pImage->Draw();

    // Store time after virtual function calls
    auto now = Clock::now();

    cout << "Time taken: "
         << std::chrono::duration_cast
           <std::chrono::nanoseconds>(now - then).count()
         << " nanoseconds" << endl;

    return 0;
}
```

Run on IDE

Output :

```
Time taken: 2613 nanoseconds
```

See this for above result.

When a method is declared virtual, compiler secretly does two things for us:

1. Defines a VPtr in first 4 bytes of the class object
2. Inserts code in constructor to initialize VPtr to point to the VTable

What are VTable and VPtr?

When a method is declared virtual in a class, compiler creates a virtual table (aka VTable) and stores addresses of virtual methods in that table. A virtual pointer (aka VPtr) is then created and initialized to point to that VTable. A VTable is shared across all the instances of the class, i.e. compiler creates only one instance of VTable to be shared across all the objects of a class. Each instance of the class has its own version of VPtr. If we print the size of a class object containing at least one virtual method, the output will be sizeof(class data) + sizeof(VPtr).

Since address of virtual method is stored in VTable, VPtr can be manipulated to make calls to those virtual methods thereby violating principles of encapsulation. See below example:

```
// A C++ program to demonstrate that we can directly
// manipulate VPtr. Note that this program is based
// on the assumption that compiler store vPtr in a
// specific way to achieve run-time polymorphism.
#include <iostream>
using namespace std;

#pragma pack(1)
```

```
// A base class with virtual function foo()
class CBase
{
public:
    virtual void foo() noexcept {
        cout << "CBase::Foo() called" << endl;
    }
protected:
    int mData;
};

// A derived class with its own implementation
// of foo()
class CDerived : public CBase
{
public:
    void foo() noexcept  {
        cout << "CDerived::Foo() called" << endl;
    }
private:
    char cChar;
};

// Driver code
int main()
{
    // A base type pointer pointing to derived
    CBase *pBase = new CDerived;

    // Accessing vPtr
    int* pVPtr = *(int**)pBase;

    // Calling virtual method
    ((void(*)())pVPtr[0])();

    // Changing vPtr
    delete pBase;
    pBase = new CBase;
    pVPtr = *(int**)pBase;

    // Calls method for new base object
    ((void(*)())pVPtr[0])();

    return 0;
}
```

Run on IDE

Output :

```
CDerived::Foo() called
CBase::Foo() called
```

We are able to access vPtr and able to make calls to virtual methods through it. The memory representation of objects is explained here.

Is it wise to use virtual method?

As it can be seen, through base class pointer, call to derived class method is being dispatched. Everything seems to be working fine. Then what is the problem?

If a virtual routine is called many times (order of hundreds of thousands), it drops the performance of system, reason being each time the routine is called, its address needs to be resolved by looking through

VTable using VPtr. Extra indirection (pointer dereference) for each call to a virtual method makes accessing VTable a costly operation and it is better to avoid it as much as we can.

Curiously Recurring Template Pattern (CRTP)

Usage of VPtr and VTable can be avoided altogether through Curiously Recurring Template Pattern (CRTP). CRTP is a design pattern in C++ in which a class X derives from a class template instantiation using X itself as template argument. More generally it is known as F-bound polymorphism.

```
// Image program (similar to above) to demonstrate
// working of CRTP
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
    Dimension(int _X, int _Y)
    {
        mX = _X;
        mY = _Y;
    }
private:
    int mX, mY;
};

// Base class for all image types. The template
// parameter T is used to know type of derived
// class pointed by pointer.
template <class T>
class Image
{
public:
    void Draw()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->Draw();
    }
    Dimension GetDimensionInPixels()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->GetDimensionInPixels();
    }

protected:
    int dimensionX, dimensionY;
};


// For Tiff Images
class TiffImage : public Image<TiffImage>
{
public:
    void Draw()
    {
        // Uncomment this to check method dispatch
        // cout << "TiffImage::Draw() called" << endl;
```

```
    }
    Dimension GetDimensionInPixels()
    {
        return Dimension(dimensionX, dimensionY);
    }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc

// Driver code
int main()
{
    // An Image type pointer pointing to Tiffimage
    Image<TiffImage>* pImage = new TiffImage;

    // Store time before virtual function calls
    auto then = Clock::now();

    // Call Draw 1000 times to make sure performance
    // is visible
    for (int i = 0; i < 1000; ++i)
        pImage->Draw();

    // Store time after virtual function calls
    auto now = Clock::now();

    cout << "Time taken: "
         << std::chrono::duration_cast
         <std::chrono::nanoseconds>(now - then).count()
         << " nanoseconds" << endl;

    return 0;
}
```

Run on IDE

Output :

```
Time taken: 732 nanoseconds
```

See this for above result.

Virtual method vs CRTP benchmark

The time taken while using virtual method was 2613 nanoseconds. This (small) performance gain from CRTP is because the use of a VTable dispatch has been circumvented. Please note that the performance depends on a lot of factors like compiler used, operations performed by virtual methods. Performance numbers might differ in different runs, but (small) performance gain is expected from CRTP.

Note: If we print size of class in CRTP, it can bee seen that VPtr no longer reserves 4 bytes of memory.

```
cout << sizeof(Image) << endl;
```

Questions? Keep them coming. We would love to answer.

Reference(s)

https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

This article is contributed by **Aashish Barnwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Design Pattern

Login to Improve this Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Recommended Posts:

Flyweight Design Pattern
Implementing Iterator pattern of a single Linked List
Iterator Pattern
Command Pattern
Proxy Design Pattern
Dependency Inversion Principle (SOLID)
Design Video Sharing System Like Youtube
Design Scalable System like Foursquare
Design Scalable System like Instagram
Mediator Design Pattern

(Login to Rate)

4.3 Average Difficulty : **4.3/5.0**
Based on **3** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List

Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Share this post!

10 Comments **GeeksforGeeks** 1 **Login**

Recommend **Share** **Sort by Newest**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Anuj Mahajan • 10 months ago

Can the destructor be called in correct order using this mechanism ?

• Reply • Share ›

Ranjan Mahajan • 2 years ago

Virtual mechanism - 27310 nanoseconds
CRTP - 47109 nanoseconds

I dont see any performace gain or have I missed something
Even in multiple runs, performance is low with CRTP

• Reply • Share ›

Aashish Barnwal → Ranjan Mahajan • 2 years ago

There was a typo in perf numbers. We have fixed this.

• Reply • Share ›

Ranjan Mahajan → Aashish Barnwal • 2 years ago

With low loop counter, results may be in favour of CRTP
(Thats probably because nanosecond prescision is normally not possible using clock APIs)

With high loop counter, the results are always in favour of Virtual mechanism.

The reason is also obvious.
Virtual mechanism requires a few arithmetic operations to find function pointer and then calls that function (Arithmetic operations are way faster on modern processors)

CRTP requires 2 function call overheads (1 call to base class function and another call to derived class function).
This can be checked in assembly output of program too.

Function call overhead is much more than 2 or 3 arithmetic operations to fetch function pointer.

So, either we are missing something in CRTP implementation or perhaps this technique is not very promising on modern processors

But it is a good technique none the less. Thanks for sharing.. Cheers :)

• Reply • Share ›

Aashish Barnwal → Ranjan Mahajan • 2 years ago

Result will be in favor of CRTP even if the loop counter is high. Make sure you are not doing any I/O ops in method. To see significant diff, the operation are not doing any I/O ops in method. To see significant diff, the operation performed by the method should be significantly minimal as compared to the actual call made to that method. Can you share the details where you are running the code?

Reply • Share ›

Ranjan Mahajan → Aashish Barnwal • 2 years ago

I ran the code on G4G IDE and on my linux machine.
CRTP took almost twice as long as virtual mechanism

G4G IDE results (with counter 100000)
CRTP- Time taken: 517811 nanoseconds
Virtual - Time taken: 274038 nanoseconds

Reply • Share ›

Aashish Barnwal → Ranjan Mahajan • 2 years ago

Code with CRTP executes much faster when we enable optimization flag "-O3" and enable inlining as well. I have tried this on linux and observed that CRTP is almost 100 times faster than old virtual method. You can also try this through make file. Let me know your results.
This link talks about benchmark comparison between CRTP and Virtual methods:
https://nativecoding.wordpr...

Reply • Share ›

ajit kumar • 2 years ago

But as per your print , CRTP method is taking more time than dynamic polymorphism method ..!!!! or am i missing some thing ??

Reply • Share ›

Aashish Barnwal → ajit kumar • 2 years ago

There was a typo in perf numbers. We have fixed this. Try to check perf numbers locally, You should be able to see the diff.

Reply • Share ›

ajit kumar → Aashish Barnwal • 2 years ago

Thanks aashish

Reply • Share ›