Simulating a virtual static member of a class in c++?

Asked 16 years, 3 months ago Modified 2 years, 4 months ago Viewed 7k times

12

Is there anyway to have a sort of virtual static member in C++?

For example:

```
class BaseClass {
    public:
        BaseClass(const string& name) : _name(name) {}
        string GetName() const { return _name; }
        virtual void UseClass() = 0;
    private:
        const string _name;
};


class DerivedClass : public BaseClass {
    public:
        DerivedClass() : BaseClass("DerivedClass") {}
        virtual void UseClass() { /* do something */ }
};
```

I know this example is trivial, but if I have a vector of complex data that is going to be always the same for all derived class but is needed to be accessed from base class methods?

```
class BaseClass {
    public:
        BaseClass() {}
        virtual string GetName() const = 0;
        virtual void UseClass() = 0;
};


class DerivedClass : public BaseClass {
    public:
        DerivedClass() {}
        virtual string GetName() const { return _name; }
        virtual void UseClass() { /* do something */ }
    private:
        static const string _name;
};

string DerivedClass::_name = "DerivedClass";
```

This solution does not satify me because I need reimplement the member _name and its accessor GetName() in every class. In my case I have several members that follows _name behavior and tenths of derived classes.

Any idea?

c++ virtual-functions

Share Improve this question Follow

asked Aug 29, 2008 at 15:54

martjno

4,719 ● 5 ● 35 ● 31

5 Answers

Sorted by: Highest score (default)

9

Here is one solution:

```
struct BaseData
{
  const string my_word;
  const int my_number;
};

class Base
{
public:
    Base(const BaseData* apBaseData)
    {
        mpBaseData = apBaseData;
    }
    const string getMyWord()
    {
        return mpBaseData->my_word;
    }
    int getMyNumber()
    {
        return mpBaseData->my_number;
    }
private:
    const BaseData* mpBaseData;
};

class Derived : public Base
{
public:
    Derived() : Base(&sBaseData)
    {
    }
private:
    static BaseData sBaseData;
}

BaseData Derived::BaseData = { "Foo", 42 };
```

Share Improve this answer Follow

answered Aug 29, 2008 at 20:11


Jeremy
2,338 ● 3 ● 21 ● 24

2

It seems like the answer is in the question - the method you suggested seems to be the right direction to go, except that if you have a big number of those shared members you might want to gather them into a struct or class and past that as the argument to the constructor of the base class.

If you insist on having the "shared" members implemented as static members of the derived class, you might be able to auto-generate the code of the derived classes. XSLT is a great tool for auto-generating simple classes.

In general, the example doesn't show a need for "virtual static" members, because for purposes like these you don't actually need inheritance - instead you should use the base class and have it accept the appropriate values in the constructor - maybe creating a single instance of the arguments for each "sub-type" and passing a pointer to it to avoid duplication of the shared data. Another similar approach is to use templates and pass as the template argument a class that provides all the relevant values (this is commonly referred to as the "Policy" pattern).

To conclude - for the purpose of the original example, there is no need for such "virtual static" members. If you still think they are needed for the code you are writing, please try to elaborate and add more context.

Example of what I described above:

```
class BaseClass {
    public:
        BaseClass(const Descriptor& desc) : _desc(desc) {}
        string GetName() const { return _desc.name; }
        int GetId() const { return _desc.Id; }
        X GetX() connst { return _desc.X; }
        virtual void UseClass() = 0;
    private:
        const Descriptor _desc;
};


class DerivedClass : public BaseClass {
    public:
        DerivedClass() : BaseClass(Descriptor("abc", 1,...)) {}
        virtual void UseClass() { /* do something */ }
};

class DerDerClass : public BaseClass {
    public:
        DerivedClass() : BaseClass("Wowzer", 843,...) {}
        virtual void UseClass() { /* do something */ }
};
```

I'd like to elaborate on this solution, and maybe give a solution to the de-initialization problem:

With a small change, you can implement the design described above without necessarily create a new instance of the "descriptor" for each instance of a derived class.

You can create a singleton object, DescriptorMap, that will hold the single instance of each descriptor, and use it when constructing the derived objects like so:

```
enum InstanceType {
    Yellow,
    Big,
    BananaHammoc
}

class DescriptorsMap{
    public:
        static Descriptor* GetDescriptor(InstanceType type) {
            if ( _instance.Get() == null) {
                _instance.reset(new DescriptorsMap());
            }
            return _instance.Get()-> _descriptors[type];
        }
    private:
        DescriptorsMap() {
            descriptors[Yellow] = new Descriptor("Yellow", 42, ...);
            descriptors[Big] = new Descriptor("InJapan", 17, ...)
            ...
        }

        ~DescriptorsMap() {
            /*Delete all the descriptors from the map*/
        }

        static autoptr<DescriptorsMap> _instance;
        map<InstanceType, Descriptor*> _descriptors;
}
```

Now we can do this:

```
class DerivedClass : public BaseClass {
    public:
        DerivedClass() :
BaseClass(DescriptorsMap.GetDescriptor(InstanceType.BananaHammoc)) {}
        virtual void UseClass() { /* do something */ }
};

class DerDerClass : public BaseClass {
    public:
        DerivedClass() :
BaseClass(DescriptorsMap.GetDescriptor(InstanceType.Yellow)) {}
        virtual void UseClass() { /* do something */ }
};
```

At the end of execution, when the C runtime performs uninitializations, it also calls the destructor of static objects, including our autoptr, which in deletes our instance of the DescriptorsMap.

So now we have a single instance of each descriptor that is also being deleted at the end of execution.

Note that if the only purpose of the derived class is to supply the relevant "descriptor" data (i.e. as opposed to implementing virtual functions) then you should make do with making the base class non-abstract, and just creating an instance with the appropriate descriptor each time.

Share

Improve this answer

Follow

edited Aug 30, 2008 at 9:07

answered Aug 29, 2008 at 16:14

Hershi

2,130 ● 2 ● 19 ● 26

1

I agree with Hershi's suggestion to use a template as the "base class". From what you're describing, it sounds more like a use for templates rather then subclassing.

You could create a template as follows (have not tried to compile this):

```
template <typename T>
class Object
{
public:

  Object( const T& newObject ) : yourObject(newObject) {} ;
  T GetObject() const { return yourObject } ;
  void SetObject( const T& newObject ) { yourObject = newObject } ;

protected:

  const T yourObject ;
} ;

class SomeClassOne
{
public:

  SomeClassOne( const std::vector& someData )
  {
    yourData.SetObject( someData ) ;
  }

private:

  Object<std::vector<int>> yourData ;
} ;
```

This will let you use the template class methods to modify the data as needed from within your custom classes that use the data and share the various aspects of the template class.

If you're intent on using inheritance, then you might have to resort to the "joys" of using a void* pointer in your BaseClass and dealing with casting, etc.

However, based on your explanation, it seems like you need templates and not inheritance.

Share Improve this answer Follow

answered Aug 29, 2008 at 17:52

Misha M

11.3k ● 17 ● 55 ● 67

1

@Hershi: the problem with that approach is that each instance of each derived class has a copy of the data, which may be expensive in some way.

Perhaps you could try something like this (I'm spit-balling without a compiling example, but the idea should be clear).

```
#include <iostream>
#include <string>
using namespace std;

struct DerivedData
{
  DerivedData(const string & word, const int number) :
    my_word(word), my_number(number) {}
  const string my_word;
  const int my_number;
};

class Base {
public:
  Base() : m_data(0) {}
  string getWord() const { return m_data->my_word; }
  int getNumber() const { return m_data->my_number; }
protected:
  DerivedData * m_data;
};


class Derived : public Base {
public:
  Derived() : Base() {
    if(Derived::s_data == 0) {
      Derived::s_data = new DerivedData("abc", 1);
    }
    m_data = s_data;
  }
private:
  static DerivedData * s_data;
```

```
};


DerivedData * Derived::s_data = 0;

int main()
{
  Base * p_b = new Derived();
  cout getWord() << endl;
}
```

Regarding the follow-up question on deleting the static object: the only solution that comes to mind is to use a smart pointer, something like the Boost shared pointer.

Share
Improve this answer
Follow

edited Aug 1, 2022 at 8:01

Glorfindel
22.6k ● 13 ● 89 ● 116

answered Aug 29, 2008 at 16:48

Pat Notz
214k ● 31 ● 94 ● 92

0

It sounds as if you're trying to avoid having to duplicate the code at the leaf classes, so why not just derive an intermediate base class from the base class. this intermediate class can hold the static data, and have all your leaf classes derive from the intermediate base class. This presupposes that one static piece of data held over all the derived classes is desired, which seems so from your example.

Share Improve this answer Follow

answered May 21, 2009 at 22:49

no-op
131 ● 4