Can I have polymorphic containers with value semantics in C++?

Asked 16 years, 3 months ago Modified 6 years, 8 months ago Viewed 13k times

39

As a general rule, I prefer using value rather than pointer semantics in C++ (ie using `vector<Class>` instead of `vector<Class*>`). Usually the slight loss in performance is more than made up for by not having to remember to delete dynamically allocated objects.

Unfortunately, value collections don't work when you want to store a variety of object types that all derive from a common base. See the example below.

```
#include <iostream>

using namespace std;

class Parent
{
    public:
        Parent() : parent_mem(1) {}
        virtual void write() { cout << "Parent: " << parent_mem << endl; }
        int parent_mem;
};

class Child : public Parent
{
    public:
        Child() : child_mem(2) { parent_mem = 2; }
        void write() { cout << "Child: " << parent_mem << ", " << child_mem <<
endl; }

        int child_mem;
};

int main(int, char**)
{
    // I can have a polymorphic container with pointer semantics
    vector<Parent*> pointerVec;

    pointerVec.push_back(new Parent());
    pointerVec.push_back(new Child());

    pointerVec[0]->write();
    pointerVec[1]->write();

    // Output:
    //
    // Parent: 1
    // Child: 2, 2

    // But I can't do it with value semantics
```

```
    vector<Parent> valueVec;

    valueVec.push_back(Parent());
    valueVec.push_back(Child());    // gets turned into a Parent object :(

    valueVec[0].write();
    valueVec[1].write();

    // Output:
    //
    // Parent: 1
    // Parent: 2

}
```

My question is: Can I have have my cake (value semantics) and eat it too (polymorphic containers)? Or do I have to use pointers?

c++ stl

Share
Improve this question
Follow

edited May 30, 2016 at 18:49

Agustin Meriles
4,854 ● 3 ● 30 ● 44

asked Sep 3, 2008 at 2:09

AndrewR
10.9k ● 10 ● 47 ● 58

9 Answers

Sorted by: Highest score (default)

28

Since the objects of different classes will have different sizes, you would end up running into the slicing problem if you store them as values.

One reasonable solution is to store container safe smart pointers. I normally use boost::shared_ptr which is safe to store in a container. Note that std::auto_ptr is not.

```
vector<shared_ptr<Parent>> vec;
vec.push_back(shared_ptr<Parent>(new Child()));
```

shared_ptr uses reference counting so it will not delete the underlying instance until all references are removed.

Share Improve this answer Follow

answered Sep 3, 2008 at 2:19

1800 INFORMATION
135k ● 30 ● 163 ● 242

3 `boost::ptr_vector` is often a cheaper and simpler alternative to `std::vector<boost::shared_ptr<T>>` – ben Sep 16, 2008 at 16:34

4 This answer doesn't address value semantics. shared_ptr<T> provides reference semantics over classes derived from T, for instace, shared_ptr<Base> a, b; b.reset(new Derived1); a = b; doesn't make a copy of the Derived1 object. – Aaron Sep 17, 2008 at 5:28

5 I never said my solution addressed value semantics. I said it was "a reasonable solution". If you know of a way to have polymorphic value semantics then step right up and collect your nobel prize. – 1800 INFORMATION Sep 17, 2008 at 8:38

2 There are a number of `clone_ptr` or `value_ptr` types which can achieve this. – Puppy Jan 9, 2013 at 13:16

4 **Note that `shared_ptr` is now in the `std` namespace, as of C++11.** You want to use this one instead of Boost's. – lynn Oct 22, 2015 at 16:02

13

I just wanted to point out that vector<Foo> is usually more efficient than vector<Foo*>. In a vector<Foo>, all the Foos will be adjacent to each other in memory. Assuming a cold TLB and cache, the first read will add the page to the TLB and pull a chunk of the vector into the L# caches; subsequent reads will use the warm cache and loaded TLB, with occasional cache misses and less frequent TLB faults.

Contrast this with a vector<Foo*>: As you fill the vector, you obtain Foo*'s from your memory allocator. Assuming your allocator is not extremely smart, (tcmalloc?) or you fill the vector slowly over time, the location of each Foo is likely to be far apart from the other Foos: maybe just by hundreds of bytes, maybe megabytes apart.

In the worst case, as you scan through a vector<Foo*> and dereferencing each pointer you will incur a TLB fault and cache miss -- this will end up being a *lot* slower than if you had a vector<Foo>. (Well, in the really worst case, each Foo has been paged out to disk, and every read incurs a disk seek() and read() to move the page back into RAM.)

So, keep on using vector<Foo> whenever appropriate. :-)

Share Improve this answer Follow

answered Sep 16, 2008 at 11:08

0124816
1,083 ● 6 ● 11

4 +1 for cache consideration! This problem will become much more relevant in future. – Konrad Rudolph Sep 16, 2008 at 11:13

2 It depends on how expensive Foo::Foo(const Foo&) is, as the value semantics container will need to call it on inserts. – AndrewR Sep 17, 2008 at 23:26

Good point -- however as long as you are appending, it's not an issue. (You'll have log2(n) extra copies.) Also, most accesses are reads and not writes; sometimes suffering the occasional expensive write can still be a net win by making reads faster. – 0124816 Sep 18, 2008 at 5:23

10

Yes, you can.

The boost.ptr_container library provides polymorphic value semantic versions of the standard containers. You only have to pass in a pointer to a heap-allocated object, and the container will take ownership and all further operations will provide value semantics , except for reclaiming ownership, which gives you almost all the benefits of value semantics by using a smart pointer.

Share Improve this answer Follow

answered Sep 4, 2008 at 21:39

ben

5

You might also consider boost::any. I've used it for heterogeneous containers. When reading the value back, you need to perform an any_cast. It will throw a bad_any_cast if it fails. If that happens, you can catch and move on to the next type.

I *believe* it will throw a bad_any_cast if you try to any_cast a derived class to its base. I tried it:

```
  // But you sort of can do it with boost::any.

  vector<any> valueVec;

  valueVec.push_back(any(Parent()));
  valueVec.push_back(any(Child()));        // remains a Child, wrapped in an
Any.

  Parent p = any_cast<Parent>(valueVec[0]);
  Child c = any_cast<Child>(valueVec[1]);
  p.write();
  c.write();

  // Output:
  //
  // Parent: 1
  // Child: 2, 2

  // Now try casting the child as a parent.
  try {
      Parent p2 = any_cast<Parent>(valueVec[1]);
      p2.write();
  }
  catch (const boost::bad_any_cast &e)
  {
      cout << e.what() << endl;
  }

  // Output:
  // boost::bad_any_cast: failed conversion using boost::any_cast
```

All that being said, I would also go the shared_ptr route first! Just thought this might be of some interest.

Share

Improve this answer

Follow

edited Sep 3, 2008 at 12:02

answered Sep 3, 2008 at 2:56

Adam Hollidge

759 ● 6 ● 12

While searching for an answer to this problem, I came across both this and a similar question. In the answers to the other question you will find two suggested solutions:

5

1. Use std::optional or boost::optional and a visitor pattern. This solution makes it hard to add new types, but easy to add new functionality.
2. Use a wrapper class similar to what Sean Parent presents in his talk. This solution makes it hard to add new functionality, but easy to add new types.

The wrapper defines the interface you need for your classes and holds a pointer to one such object. The implementation of the interface is done with free functions.

Here is an example implementation of this pattern:

```
class Shape
{
public:
    template<typename T>
    Shape(T t)
        : container(std::make_shared<Model<T>>(std::move(t)))
    {}

    friend void draw(const Shape &shape)
    {
        shape.container->drawImpl();
    }
    // add more functions similar to draw() here if you wish
    // remember also to add a wrapper in the Concept and Model below

private:
    struct Concept
    {
        virtual ~Concept() = default;
        virtual void drawImpl() const = 0;
    };

    template<typename T>
    struct Model : public Concept
    {
        Model(T x) : m_data(move(x)) { }
        void drawImpl() const override
        {
            draw(m_data);
        }
        T m_data;
    };

    std::shared_ptr<const Concept> container;
};
```

Different shapes are then implemented as regular structs/classes. You are free to choose if you want to use member functions or free functions (but you will have to update the above implementation to use member functions). I prefer free functions:

```
struct Circle
{
    const double radius = 4.0;
};
```

```
struct Rectangle
{
    const double width = 2.0;
    const double height = 3.0;
};

void draw(const Circle &circle)
{
    cout << "Drew circle with radius " << circle.radius << endl;
}

void draw(const Rectangle &rectangle)
{
    cout << "Drew rectangle with width " << rectangle.width << endl;
}
```

You can now add both `Circle` and `Rectangle` objects to the same `std::vector<Shape>`:

```
int main() {
    std::vector<Shape> shapes;
    shapes.emplace_back(Circle());
    shapes.emplace_back(Rectangle());
    for (const auto &shape : shapes) {
        draw(shape);
    }
    return 0;
}
```

The downside of this pattern is that it requires a large amount of boilerplate in the interface, since each function needs to be defined three times. The upside is that you get copy-semantics:

```
int main() {
    Shape a = Circle();
    Shape b = Rectangle();
    b = a;
    draw(a);
    draw(b);
    return 0;
}
```

This produces:

```
Drew rectangle with width 2
Drew rectangle with width 2
```

If you are concerned about the `shared_ptr`, you can replace it with a `unique_ptr`. However, it will no longer be copyable and you will have to either move all objects or

implement copying manually. Sean Parent discusses this in detail in his talk and an implementation is shown in the above mentioned answer.

Share Improve this answer Follow

answered Apr 8, 2018 at 17:46

dragly

1,475 ● 12 ● 14

This really should be the accepted answer IMO as the wrapper you mention (what I've started calling a "polymorphic value type") is the solution that provides the most idiomatic and powerful solution for this that has been suggested. `std::any` looses all notion of any concepts (like deriving from a common base exposes) of the container elements. I basically made the same suggestion as yours in my answer to a similar question. Polymorphic value types for the win! – Louis Langholtz Oct 27, 2020 at 17:55

2

> Take a look at *static_cast* and *reinterpret_cast*
> In C++ Programming Language, 3rd ed, Bjarne Stroustrup describes it on page 130. There's a whole section on this in Chapter 6.
> You can recast your Parent class to Child class. This requires you to know when each one is which. In the book, Dr. Stroustrup talks about different techniques to avoid this situation.

Do not do this. This negates the polymorphism that you're trying to achieve in the first place!

Share Improve this answer Follow

answered Sep 3, 2008 at 2:29

17 of 26

27.4k ● 13 ● 68 ● 85

2

Most container types want to abstract the particular storage strategy, be it linked list, vector, tree-based or what have you. For this reason, you're going to have trouble with both possessing and consuming the aforementioned cake (i.e., the cake is lie (NB: someone had to make this joke)).

So what to do? Well there are a few cute options, but most will reduce to variants on one of a few themes or combinations of them: picking or inventing a suitable smart pointer, playing with templates or template templates in some clever way, using a common interface for containees that provides a hook for implementing per-containee double-dispatch.

There's basic tension between your two stated goals, so you should decide what you want, then try to design something that gets you basically what you want. It *is* possible to do some nice and unexpected tricks to get pointers to look like values with clever

enough reference counting and clever enough implementations of a factory. The basic idea is to use reference counting and copy-on-demand and constness and (for the factor) a combination of the preprocessor, templates, and C++'s static initialization rules to get something that is as smart as possible about automating pointer conversions.

I have, in the past, spent some time trying to envision how to use Virtual Proxy / Envelope-Letter / that cute trick with reference counted pointers to accomplish something like a basis for value semantic programming in C++.

And I think it could be done, but you'd have to provide a fairly closed, C#-managed-code-like world within C++ (though one from which you could break through to underlying C++ when needed). So I have a lot of sympathy for your line of thought.

Share Improve this answer Follow

answered Sep 3, 2008 at 2:58

Thomas Kammeyer

2

Just to add one thing to all 1800 INFORMATION already said.

You might want to take a look at "More Effective C++" by Scott Mayers "Item 3: Never treat arrays polymorphically" in order to better understand this issue.

Share

Improve this answer

Follow

edited May 23, 2017 at 12:00

Community Bot

1 ● 1

answered Sep 3, 2008 at 7:36

Serge

7,694 ● 5 ● 41 ● 46

1

I'm using my own templated collection class with exposed value type semantics, but internally it stores pointers. It's using a custom iterator class that when dereferenced gets a value reference instead of a pointer. Copying the collection makes deep item copies, instead of duplicated pointers, and this is where most overhead lies (a really minor issue, considered what I get instead).

That's an idea that could suit your needs.

Share Improve this answer Follow

answered Sep 16, 2008 at 11:24

Johann Gerell

25.5k ● 11 ● 76 ● 125