C++ question about setting class variables

Asked 15 years, 9 months ago Modified 15 years ago Viewed 452 times

2

I'm not new to programming, but after working in Java I'm coming back to C++ and am a little confused about class variables that aren't pointers. Given the following code:

```
#include <iostream>
#include <map>

using namespace std;

class Foo {
    public:
        Foo() {
            bars[0]     = new Bar;
            bars[0]->id = 5;
        }

        ~Foo() { }

        struct Bar {
            int id;
        };

        void set_bars(map<int,Bar*>& b) {
            bars = b;
        }

        void hello() {
            cout << bars[0]->id << endl;
        }

    protected:
        map<int,Bar*> bars;
};

int main() {
    Foo foo;
    foo.hello();

    map<int,Foo::Bar*> testbars;
    testbars[0]     = new Foo::Bar;
    testbars[0]->id = 10;

    foo.set_bars(testbars);

    foo.hello();

    return(0);
}
```

I get the expected output of 5 & 10. However, my lack of understanding about references and pointers and such in C++ make me wonder if this will actually work in the wild, or if once testbars goes out of scope it will barf. Of course, here, testbars will not go out of scope before the program ends, but what if it were created in another class function as a function variable? Anyway, I guess my main question is would it better/safer for me to create the bars class variable as a pointer to the map map?

c++

Share Improve this question Follow

asked Mar 13, 2009 at 16:38
Bryan

5 Answers

Sorted by: Highest score (default)

4

> Anyway, I guess my main question is would it better/safer for me to create the bars class variable as a pointer to the map map?

No. C++ is nothing like Java in this and may other respects. If you find yourself using pointers and allocating new'd objects to them a lot, you are probably doing something wrong. To learn the right way to do things, I suggest getting hold of a copy of Accelerated C++ by Koenig & Moo,

Share Improve this answer Follow

answered Mar 13, 2009 at 16:44
anon

3

The member variable `bars` is a separate instance of a "dictionary"-like/associative array class. So when it is assigned to in set_bars, the contents of the parameter `b` are copied into `bars`. So there is no need to worry about the relative lifetimes of `foo` and `testbars`, as they are independent "value-like" entites.

You have more of a problem with the lifetimes of the Bar objects, which are currently never going to be deleted. If you add code somewhere to delete them, then you will introduce a further problem because you are copying the addresses of Bar objects (rather than the objects themselves), so you have the same object pointed to by two different maps. Once the object is deleted, the other map will continue to refer to it. This is the kind of thing that you should avoid like the plague in C++! Naked pointers to objects allocated with `new` are a disaster waiting to happen.

References (declared with &) are not different from pointers with regard to object lifetimes. To allow you to refer to the same object from two places, you can use either pointers or references, but this will still leave you with the problem of deallocation.

You can get some way toward solving the deallocation problem by using a class like `shared_ptr`, which should be included with any up-to-date C++ environment (in `std::tr1`). But then you may hit problems with cyclical pointer networks (A points to B and B points to A, for example), which will not be automatically cleaned up.

Share
Improve this answer
Follow

edited Mar 13, 2009 at 17:27

answered Mar 13, 2009 at 16:46
Daniel Earwicker
117k ● 38 ● 208 ● 286

I have another class I use to manipulate the data within the Bar objects. When that class changes the values of Bar variables, I want the changes to be maintained w/out having to pass the Bar objects back. Should I be using references (Bar&) in my map instead? – Bryan Mar 13, 2009 at 17:05

I've added a bit more to my answer. NB. I've been using C++ commercially since 1994 and it amazes me that Java/C# programmers ever want to learn it now! It's just not the right tool for most jobs today. It's unnecessarily painful to do even simple things correctly. – Daniel Earwicker Mar 13, 2009 at 17:33

1

For every new you need a corresponding delete. If you try and reference the memory after you call delete - where ever that is - then the program will indeed "barf".

If you don't then you will be fine, it's that simple.

You should design your classes so that ownership of memory is explicit, and that you **KNOW** that for every allocation you are doing an equal deallocation. Never assume another class/container will delete memory you allocated.

Hope this helps.

Share Improve this answer Follow

answered Mar 13, 2009 at 16:46
Binary Worrier
51.7k ● 20 ● 142 ● 186

0

In the code below you can pass map of Bars and then will be able to modify Bars outside of the class.

But. But unless you call set_bars again.

It is better when one object is responsible for creation and deletion of Bars. Which is not true in your case.

If you want you can use boost::shared_ptr< Bars > instead of Bars*. That will be more Java like behavior.

```
class Foo {
public:
    Foo() {
        bars[0]     = new Bar;
        bars[0]->id = 5;
    }

    ~Foo() { freeBarsMemory(); }

    struct Bar {
        int id;
    };

    typedef std::map<int,Bar*> BarsList;

    void set_bars(const BarsList& b) {
        freeBarsMemory();
        bars = b;
    }

    void hello() {
        std::cout << bars[0]->id << std::endl;
    }

protected:
    BarsList bars;

    void freeBarsMemory()
    {
        BarsList::const_iterator it = bars.begin();
        BarsList::const_iterator end = bars.end();

        for (; it != end; ++it)
            delete it->second;

        bars.clear();
    }
};
```

Share

Improve this answer

Follow

edited Mar 13, 2009 at 18:07

answered Mar 13, 2009 at 16:46

Mykola Golubyev

59.7k ● 15 ● 93 ● 102

To be on the safe side, I'd add "bars.clear();" at the end of freeBarsMemory(), so it is a little safer to use. – Daniel Earwicker Mar 13, 2009 at 18:03

> I'm not new to programming, but after working in Java I'm coming back to C++ and am a little confused about class variables that aren't pointers.

0

The confusion appears to come from a combination of data that is on the heap and data that is not necessarily on the heap. This is a common cause of confusion.

In the code you posted, `bars` is not a pointer. Since it's in class scope, it will exist until the object containing it (`testbars`) is destroyed. In this case `testbars` was created on the stack so it will be destroyed when it falls out of scope, regardless of how deeply nested that scope is. And when `testbars` is destroyed, subobjects of `testbars` (whether they are parent classes or objects contained within the `testbars` object) will have their destructors run at that exact moment in a well-defined order.

This is an extremely powerful aspect of C++. Imagine a class with a 10-line constructor that opens a network connection, allocates memory on the heap, and writes data to a file. Imagine that the class's destructor undoes all of that (closes the network connection, deallocates the memory on the heap, closes the file, etc.). Now imagine that creating an object of this class fails halfway through the constructor (say, the network connection is down). How can the program know which lines of the destructor will undo the parts of the constructor that succeeded? There is no general way to know this, so the destructor of that object is not run.

Now imagine a class that contains ten objects, and the constructor for each of those objects does one thing that must be rolled back (opens a network connection, allocates memory on the heap, writes data to a file, etc.) and the destructor for each of those objects includes the code necessary to roll back the action (closes the network connection, deallocates objects, closes the file, etc.). If only five objects are successfully created then only those five need to be destroyed, and their destructors **will** run at that exact moment in time.

If `testbars` had been created on the heap (via `new`) then it would only be destroyed when calling `delete`. In general it's much easier to use objects on the stack unless there is some reason for the object to outlast the scope it was created in.

Which brings me to `Foo::bar`. `Foo::bars` is a `map` that refers to objects on the heap. Well, it refers to pointers that, in this code example, refer to objects allocated on the heap (pointers can also refer to objects allocated on the stack). In the example you posted the objects these pointers refer to are never `delete`d, and because these objects are on the heap you're getting a (small) memory leak (which the operating system cleans up on program exit). According to the STL, `std::maps` like `Foo::bar` do **not** `delete` pointers they refer to when they are destroyed. Boost has a few solutions to this problem. In your case it's probably be easiest to simply not allocate these objects on the heap:

```
#include <iostream>
#include <map>

using std::map;
using std::cout;

class Foo {
    public:
        Foo() {
            // normally you wouldn't use the parenthesis on the next line
            // but we're creating an object without a name, so we need them
            bars[0] = Bar();
            bars[0].id = 5;
        }

        ~Foo() { }

        struct Bar {
            int id;
        };

        void set_bars(map<int,Bar>& b) {
            bars = b;
        }

        void hello() {
            cout << bars[0].id << endl;
        }

    protected:
        map<int,Bar> bars;
};

int main() {
    Foo foo;
    foo.hello();

    map<int,Foo::Bar> testbars;
    // create another nameless object
    testbars[0] = Foo::Bar();
    testbars[0].id = 10;

    foo.set_bars(testbars);
    foo.hello();
    return 0;
}
```

Share

Improve this answer

Follow

edited Dec 23, 2009 at 9:23

answered Mar 13, 2009 at 23:15

Max Lybbert

20k ● 5 ● 49 ● 69