C++ member variable aliases?

Asked 13 years, 4 months ago Modified 3 months ago Viewed 22k times

13

I'm pretty sure this is possible, because I'm pretty sure I've seen it done. I think it is awesome, but I will gladly accept answers along the lines of "this is a terrible idea because ____."

Say we have a basic struct.

```
struct vertex
{
    float x, y, z;
};
```

Now, I want to implement aliases on these variables.

```
vertex pos;
vertex col;
vertex arr;

pos.x = 0.0f; pos.y = 0.5f; pos.z = 1.0f;
col.r = 0.0f; col.g = 0.5f; col.b = 1.0f;
arr[0] = 0.0f; arr[1] = 0.5f; arr[2] = 1.0f;
```

Ideally the third syntax would be indistinguishable from an array. That is, if I sent `arr` as a reference parameter to a function expecting an array of floats into which it will store data (eg many of the OpenGL `glGet` functions), it would work fine.

What do you think? Possible? Possible but stupid?

c++ struct variables alias member

Share Edit Follow Flag

asked Jan 30, 2009 at 5:50
mezoid
7,401 • 12 • 21 • 36

13 Answers

Sorted by: Highest score (default)

24

What I would do is make accessors:

```
struct Vertex {
    float& r() { return values[0]; }
    float& g() { return values[1]; }
    float& b() { return values[2]; }

    float& x() { return values[0]; }
    float& y() { return values[1]; }
    float& z() { return values[2]; }

    float  operator[] (unsigned i) const { return values_[i]; }
    float& operator[] (unsigned i)       { return values_[i]; }
    operator float*() const { return values_; }

private:
    float[3] values_;
}
```

Share Edit Follow Flag

edited Jan 30, 2009 at 6:47
Loz Tourville
79.8k • 5 • 43 • 107

answered Jan 30, 2009 at 6:12
Ray Hidayat
19.7k • 4 • 43 • 44

Yup, this is both legal and avoids the aliasing problems that solution relying on unions suffer from. – Leif Jan 30, 2009 at 13:55

Good start, however, this involves having to call a memberfunction in order to access a field. You can do the following (yeah, having references as members is not something you typically do, but in some cases it can be handy): struct Vertex { Vertex(): r(values[0]), g(values[1]), b(values[2]), x(values[0]), y(values[1]), z(values[2]) {} float& r; float& g; float& b; float& x; private: float values[3]; }; – ereOne Dec 2, 2009 at 19:02

Those methods are written inline so the compiler should optimise out the member function call and access the variable directly, so it's a non issue. I compiled your code under Visual Studio 2008 and it was 3x bytes larger because of the six references you've added to the class. If the intention was for Vertex to be a lightweight classed which would be passed by value, then that's not what you'd want. – Ray Hidayat Dec 3, 2009 at 21:51

Nameless nested structs in a union are not standard C++. This, however, should work:

14

```
struct Vertex
{
private:
    typedef float Vertex::* const vert[3];
    static const vert v;

public:
    typedef size_t size_type;
    float x, y, z;

    const float& operator[](size_type i) const {
        return this->*v[i];
    }

    float& operator[](size_type i) {
        return this->*v[i];
    }
};
const Vertex::vert Vertex::v = {&Vertex::x, &Vertex::y, &Vertex::z};
```

EDIT: A little more information. The struct uses an array of 3 pointer-to-data-members to access the data in the overloaded [] operators.

The line "typedef float Vertex::* const vert" means that vert is a pointer to a float member of the Vertex struct. The [3] means that it's an array of 3 of these. In the overloaded operator[], this array is indexed and the pointer-to-data-member is dereferenced and the value returned.

Additionally, this method should work regardless of packing issues - the compiler is free to pad the Vertex structure however it likes and it'll still work just fine. An anonymous union will run into problems if the floats are packed differently.

Share Edit Follow Flag

edited Jan 30, 2009 at 7:25

answered Jan 30, 2009 at 7:12
Adrian
376 • 1 • 4

Because that saves memory, there's no reason for it to be per-instance. Pointer-to-data-members require an instance of the object before they can be used - a (float Vertex::*) pointing to Vertex.x (for example) can be used with any Vertex instance to get that instance's x member. – Adrian Jan 30, 2009 at 8:09

What does this (->*)() actually compile to, I wonder? Does it result in a double indirection, or does the compiler quietly inline the pointer-to-member and end up with a single indexed load op? – Craigweb Jan 30, 2009 at 11:00

Use a union?

14

```
union vertex
{
    struct { float x, y, z; };
    struct { float r, g, b; };
    float arr[3];
};
```

I wouldn't recommend it - it will lead to confusion.

Added:

As noted by Adrian in his answer, this union with anonymous struct members is not supported by ISO C++. It works in GNU G++ (with complaints about not being supported when you turn on `-ansi -pedantic`). It is reminiscent of the pre-pre-standard C days (pre-K&R 1st Edn), when structure element names had to be unique across all structures, and you could use contracted notations to get to an offset within the structure, and you could use member names from other structure types - a form of anarchy. By the time I started using C (a long time ago, but post-K&R1), that was already historical usage.

The notation shown with anonymous union members (for the two structures) is supported by C11 (ISO/IEC 9899:2011), but not by earlier versions of the C standard. Section 5.5 of ISO/IEC 14882:2011 (C++11) provides for anonymous unions, but GNU `g++` (4.9.1) does not accept the code shown with `-pedantic`, identifying "warning: ISO C++ prohibits anonymous structs [-Wpedantic]".

Since the idea will lead to confusion, I'm not particularly concerned that it isn't standard; I would not use the mechanism for this task (and I'd be leery of using anonymous structures in a union even if it was beneficial).

A concern was raised:

> The three (x-y-z, r-g-b and the array) do not necessarily align.

It is a union with three elements; the three elements start at the same address. The first two are structures containing 3 float values; There's no inheritance and there are no virtual functions to give different layouts, etc. The structures will be laid out with the three elements contiguous (in practice, even if the standard permits padding). The array also starts at the same address, and subject to 'no padding' in the structures, the elements overlap the two structures. I really don't see that there would be a problem.

Share Edit Follow Flag

edited Mar 4, 2015 at 20:12

answered Jan 30, 2009 at 5:59
Jonathan Leffler
659k • 130 • 837 • 1228

Confusion is an understatement... since it's possible to set it as one type and refer to it as another type. You'll go nuts trying to work out if it's a pos, col or an in any context. – Adam Hawes Jan 30, 2009 at 7:07

Would be technically illegal. – MSalters Jan 30, 2009 at 12:42

Another problem with it is that it leads to aliasing, which inhibits compiler optimization. If performance matters, that might be a problem. – ijw Jan 30, 2009 at 13:33

References?

4

```
template<typename T>
struct Vector3
{
    Vector3() :
        r(data[0]), g(data[1]), b(data[2]),
        x(data[0]), y(data[1]), z(data[2])
    {
    }

    T *operator *() {
        return data;
    }

    const T *operator *() const {
        return data;
    }

    T data[3];
    T &r, &g, &b;
    T &x, &y, &z;
};
```

Share Edit Follow Flag

edited Jan 30, 2009 at 13:10

answered Jan 30, 2009 at 13:05
strager
50.6k • 26 • 121 • 176

I was thinking along the same lines, but I discovered it has two flaws: 1) you add the references to each instance, which uses up more memory, e.g. on 64 bit, with T=float, you're using 500%, 533% if aligned, the amount of memory you want to be using, for a model with lots of vertexes, that might amount to quite a lot. And 2) you have to define an assignment operator, if you ever want to assign a vertex to a vertex, and you'll end up with corrupt data if you ever try to `memcpy` the structure (which is not too uncommon if you want to use it as POD with some extra helpers, not using virtuals) – Tatarize Dec 5, 2013 at 8:01

Following structure will have the requested behavior.

3

```
struct vertex
{
private:
    float data[3];
public:
    float &x, &y, &z;
    float &r, &g, &b;

    vertex() : x(data[0]), y(data[1]), z(data[2]), r(data[0]), g(data[1]), b(data[2]) {
    }

    float& operator [](int i) {
        return data[i];
    }
};
```

Share Edit Follow Flag

answered Mar 24, 2015 at 16:38
Krasimir
2,778 • 1 • 16 • 12

This is clever but note each reference adds memory to the struct, which might not be what you want here. Using accessors doesn't add any memory. – Philip Feb 20, 2018 at 21:29

@Philip I didn't notice any extra memory. The example lays out into `01 00 00 00 02 00 00 00 03 00 00 00`. C++ `&` is replaced with memory address in compile-time, isn't it? – KC_ May 16, 2020 at 8:23

You can get this with a union as others have mentioned. Overloading color and position onto the same structure like this may not be a good idea (for example, adding two colors usually means you want to saturate to 1.0, whereas adding vectors happens linearly), but overlaying a float[] on top of them like that is perfectly fine and a well accepted means of interchanging data with GL/DirectX/etc.

3

I recommend you avoid referring to the same member by different aliases in the same function scope, though, because this will drive you into a nasty hardware stall called a load-hit-store. In particular, avoid this if you can:

```
vector Foo;
Foo.x = 3.0f;
return Foo[0] + Foo[1];
```

Share Edit Follow Flag

answered Jan 30, 2009 at 6:05
Crashworks
35.3k • 12 • 99 • 160

I guess you can do some macro magic to get what you want. But that will look ugly. Why do you want to use same struct, vertex for 3 different types? Why can't you define class for color? Also keep in mind that vertex and color are not same. If you change something to vertex, that will affect the color also, if you have the same class for both.

2

Share Edit Follow Flag

answered Jan 30, 2009 at 5:59
chappar
7,357 • 12 • 41 • 57

I am not sure whether I understood the question correctly. But it looks like you need to overload the operator[] to provide array like access to your struct/class. See the example mentioned here: Operator overloading

2

Share Edit Follow Flag

edited Jan 30, 2009 at 6:19

answered Jan 30, 2009 at 6:11
Naveen
71.8k • 44 • 171 • 229

Bad idea in my opinion, at least for the example given: the downside is that, for just about any solution to this, you're probably going to be able to freely assign "rgb" instances to/from "xyz" instances, which is probably rarely sensible or correct, ie you risk giving up some useful type safety.

1

Personally, for the example you give, I'd subclass rgb and xyz types from a base `boost::array<float,3>` or similar. So both of them inherit operator[], can be passed to functions expecting arrays, and passed with more type safety to things expecting colours/coordinates. It's often you want to treat an xyz or an rgb as an array, but rare you want to treat an xyz as an rgb or vice-versa. (rgb IS A array: OK, xyz IS A array: OK, rgb IS A xyz ????: I don't think so!)

Of course that means access to e.g.z & r.g.b needs to be by accessor (forwarding to the appropriate `operator[]( ... )` rather than direct to the member. (You'd need C#'s properties for that).

Share Edit Follow Flag

answered Jan 30, 2009 at 13:49
timday
24.3k • 171 • 80 • 135

I have a template and two Vector classes below, one crazy, one sane. The template implements a simple fixed at compile time array of values. It is designed for subclassing and uses a protected array variable to avoid you having to jump through hoops to access the array. (Some folks might not like such a design. I say, if your subclasses are calling your overloaded operators, coupling might be a good idea.)

0

The crazy class allows you to have member variables called x, y, z and it acts like an array for glGetFloatv(). The same one just has accessor functions x(), y(), z() and still works with glGetFloatv(). You can use either class as a basis for other vector objects you might pass to the OpenGL library. Although the classes below are specific to points, you can obviously just do a search/replace to turn them into a rgb color classes.

The crazy class is crazy because the cost of the syntactic sugar vec.x instead of vec.x() is 3 reference variables. That could take up a lot of space in a large application. Use the simpler sane version.

```
template <typename T, int N>
class FixedVector {
protected:
    T arr[N];
public:
    FixedVector() {};

    FixedVector(const T* a) {
        for (int i = 0; i < N; ++i) {
            arr[i] = a[i];
        }
    }

    FixedVector(const T& other) {
        for (int i = 0; i < N; ++i) {
            arr[i] = other.arr[i];
        }
    }

    FixedVector& operator=(const T& other) {
        for (int i = 0; i < N; ++i) {
            arr[i] = other.arr[i];
        }
        return *this;
    }

    T* operator&() { return arr; }
    const T* operator&() const { return arr; }

    T& operator[](int ofs) {
        assert(ofs >= 0 && ofs < N);
        return arr[ofs];
    }

    const T& operator[](int ofs) const {
        assert(ofs >= 0 && ofs < N);
        return arr[ofs];
    }
};

class CrazyPoint : public FixedVector<float, 3> {
public:
    float &x, &y, &z;

    CrazyPoint()
    : x(arr[0]), y(arr[1]), z(arr[2])
    { arr[0] = arr[1] = arr[2] = 0.0; }

    CrazyPoint(const float* a)
    : x(arr[0]), y(arr[1]), z(arr[2])
    {
        arr[0] = a[0];
        arr[1] = a[1];
        arr[2] = a[2];
    }

    CrazyPoint(float x, float y, float z)
    : x(x), y(y), z(z)
    {
        arr[0] = x;
        arr[1] = y;
        arr[2] = z;
    }
};

class SanePoint : public FixedVector<float, 3> {
public:
    float& x() { return arr[0]; }
    float& y() { return arr[1]; }
    float& z() { return arr[2]; }

    SanePoint() { arr[0] = arr[1] = arr[2] = 0.0; }
    SanePoint(float x, float y, float z)
    {
        arr[0] = x;
        arr[1] = y;
        arr[2] = z;
    }
};

// usage
SanePoint normal;
glGetFloatv(GL_CURRENT_NORMAL, &normal);
```

Share Edit Follow Flag

answered Jan 30, 2009 at 10:20
jmucchiello
18.2k • 7 • 42 • 61

Just a warning about using reference members pointing to value members. You need to define a copy constructor (and possibly also assignment operator), if you ever copy such an object (like transfer it by value). The default copy constructor will leave you with a copy whose reference members point to the value members of the original object, not the ones of the new object. This is certainly not something you want.

0

Considering you also end up with larger objects, as already pointed out, I think using accessor methods is to be preferred over reference members.

Share Edit Follow Flag

answered Mar 9, 2009 at 16:54
KAR

You can try adding references to variables, like this:

0

```
struct test {
    float x, y, z;
    float &r, &g, &b;

    test() : r(x), g(y), b(z) {}
};
```

But your structure gets bigger (from 12 bytes to 40 bytes).

To use [] on it, use overloading of operator[], as mentioned before.

Share Edit Follow Flag

answered Jan 30, 2009 at 13:07
Nikos
14.6k • 7 • 46 • 55

I think the poster was looking for something very simple, with no performance overhead - as you would want with something like a 3D vector class. So adding virtual functions (vtable indirection cost), additional members (memory cost), unions (each new name may require a recompile of all users), or even preprocessor magic (increased program size, cross-type equivalency) is undesirable.

0

Real world use case would be to to take a templated Vector3 class (which could be double-based or float-based) and apply it to other scenarios in a user-friendly manner. It is likely defined with [x, y, z] members, but if you want to use it for rotations, you might want [psi, theta, phi], for speeds [dx, dy, dz], etc.

For the entire type, you can use the following to alias it at compile time: using Rotation3 = Vector3;

But there appears to be nothing as simple or performant that you can do for aliasing the underlying variables, right?

Share Edit Follow Flag

edited Feb 18 at 21:13

answered Feb 18 at 20:26
Tom
1 • 1