Classes by example: A 3-d vector class

by Douglas Wilhelm Harder

Suppose you are dealing with a number of vectors in a project where you are simulating the motion of, for example, a collection of vehicles or drones. This requires at least two vectors per device, including a position vector and a velocity vector. If each of the x, y and z components of each device were stored in separate arrays, along with additional arrays for the velocity components v_x, v_y and v_z, this would require you to track a number of different arrays.

Member variables

Suppose, instead, we can simplify this somewhat by at least having a vector class. Your first thought may be to just use an array, but there are benefits to using a class, and we will describe some of them as we step through this example.

A 3-dimensional array has three components:

```
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;
};
```

The class name is `Vector_3d` and this class has three member variables: `x_`, `y_` and `z_`. Each of these are, in this case, declared to be of type `double`; however, there is no need for all member variables to be of the same type. You can declare a local variable to be an instance of this class, and we call that instance an *object*. Consider, for example, the program in the next example.

Just like we have a naming convention for pointers (e.g., `p_`*`identifier`* specifies that this identifier is that of a pointer, `a_`*`identifier`* specifies that this identifier is that of an array), we will always use a trailing underscore to indicate member variables. This is not required, but is a common convention in many industries. You should note that this does not make it reserved: a reserved identifier either begins with an underscore or has two adjacent underscores in the identifier name.

```
#include <iostream>
#include <cmath>

// Class declarations
class Vector_3d;

// Function declarations
int main();

// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;
};

// Function definitions
int main() {
    // Set the three member variables to these values, respectively.
    //  - memory for three doubles is allocated on the call stack
    Vector_3d posn{3.25, 4.57, -2.34};

    // Print out the position and length of the vector
    std::cout << "(" << posn.x_ << ", "
                     << posn.y_ << ", "
                     << posn.z_ << ")" << std::endl;

    std::cout << std::sqrt( posn.x_*posn.x_ + posn.y_*posn.y_ + posn.z_*posn.z_ )
              << std::endl;

    // Move the vector one meter in the x-direction:
    posn.x_ += 1.0;

    // Print out the new position and length of the vector
    std::cout << "(" << posn.x_ << ", "
                     << posn.y_ << ", "
                     << posn.z_ << ")" << std::endl;

    std::cout << std::sqrt( posn.x_*posn.x_ + posn.y_*posn.y_ + posn.z_*posn.z_ )
              << std::endl;

    // When the function exits,
    //    the memory for the instance (requiring three doubles) is cleaned up

    return 0;
}
```

Member functions

Given a vector, you may ask questions about it: what is its norm? It may be possible to define a function to calculate this. Additionally, one may want to normalize a vector.

```
      // The norm calculated using a pass by reference
```

```
double norm( Vector_3d const &v );
void normalize( Vector_3d &v ) ;

double norm( Vector_3d const &v ) {
    return std::sqrt( v.x_*v.x_ + v.y_*v.y_ + v.z_*v.z_ );
}

void normalize( Vectore_3d &v ) {
    double nv{ norm( v ) };

    // Only normalize if it is not the zero vector
    if ( nv >= 1e-15 ) {
        v.x_ /= nv;
        v.y_ /= nv;
        v.z_ /= nv;
    }
}
```

It is now possible to call these function, passing it an object to this function as an argument:

```
// Function definitions
int main() {
    // Set the three member variables to these values, respectively.
    //  - memory for three doubles is allocated on the call stack
    Vector_3d posn{3.25, 4.57, -2.34};

    // Print out the position and length of the vector
    std::cout << "(" << posn.x_ << ", "
                     << posn.y_ << ", "
                     << posn.z_ << ")" << std::endl;

    std::cout << norm( posn ) << std::endl;

    // Normalize the vector
    normalize( posn );

    // Print out the new position and length of the vector
    std::cout << "(" << posn.x_ << ", "
                     << posn.y_ << ", "
                     << posn.z_ << ")" << std::endl;

    std::cout << norm( posn ) << std::endl;

    // When the function exits,
    //    the memory for the instance (requiring three doubles) is cleaned up

    return 0;
}
```

The problem with authoring a separate function is that there is no link between the function and the class. It is possible to author *member functions* of the class, functions that may be called in the same way that member variables are accessed. The purpose for this will become clear shortly.

```
// Class declarations
class Vector_3d;

// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;

        double norm() const;
        void normalize();
};

double Vector_3d::norm() const {
    return std::sqrt( x_*x_ + y_*y_ + z_*z_ );
}

void Vector_3d::normalize() {
    double nv{ norm() }; // Call the norm member function on the
                         // object this function was called on

    // Only normalize if it is not the zero vector
    if ( nv >= 1e-15 ) {
        x_ /= nv;
        y_ /= nv;
        z_ /= nv;
    }
}
```

You may notice something peculiar about these functions: none make reference to a `Vector_3d` object. This is because these member functions must be executed on instances of this class, and any reference to member variables is assumed to be referring to member variables from that object.

```
#include <iostream>
#include <cmath>

// Class declarations
class Vector_3d;

// Function declarations
int main();

// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;

        double norm() const;
        void normalize();
};

double Vector_3d::norm() const {
    return std::sqrt( x_*x_ + y_*y_ + z_*z_ );
}

void Vector_3d::normalize() {
    double nv{ norm() };

    // Only normalize if it is not the zero vector
    if ( nv >= 1e-15 ) {
        x_ /= nv;
        y_ /= nv;
        z_ /= nv;
    }
}

int main() {
    // Do not allow the user to change these canonical vectors
    //  - as constants, they are given upper-case first letters
    Vector_3d const I{1.0, 0.0, 0.0};
    Vector_3d const J{0.0, 1.0, 0.0};
    Vector_3d const K{0.0, 0.0, 1.0};

    Vector_3d posn{4.5, 2.3, 5.8};

    // Here, the 'x_', 'y' and 'z_' in the member function call refer to
    // the member variables of the object 'I'
    std::cout << I.norm() << std::endl;
    std::cout << J.norm() << std::endl;
    std::cout << K.norm() << std::endl;
    // Here, the 'x_', 'y_' and 'z_' in the member function call refer to
    // the member variables of the object 'posn'
    std::cout << posn.norm() << std::endl;

    return 0;
}
```

There is a subtle difference between the two member functions: if you are calling the norm member function, the last thing you'd expect is that one of the member variables is changed by that call. On the other hand, the normalize member function is expected to change the member variables. The `const` after the function parameter list indicates that the function is not meant to change any member variables while the function is being called:

```
double Vector_3d::norm() const {
    return std::sqrt( x_*x_ + y_*y_ + z_*z_ );
}
```

Now, there are in fact three different norms:

$$\|\mathbf{v}\|_2 = \sqrt{x^2 + y^2 + z^2}$$
$$\|\mathbf{v}\|_1 = |x| + |y| + |z|$$
$$\|\mathbf{v}\|_\infty = \max\{|x|, |y|, |z|\}$$

These are called the 2-norm, 1-norm and infinity-norm, respectively. Suppose you'd like your norm member function to compute all of these. This would therefore require a member function that takes a parameter. We will assume if the user passes 0, 1, or 2 that the user means for us to calculate the infinity-, 1- or 2-norm.

```
// We need the std::max function
#include <algorithm>

double Vector_3d::norm( int const norm_id ) const {
    assert( (norm_id == 0) || (norm_id == 1) || (norm_id == 2) );

    if ( norm_id == 2 ) {
        //                ______________
        //               / 2     2     2
        // ||v||_2 = \/ x   + y   + z
        return std::sqrt( x_*x_ + y_*y_ + z_*z_ );
    } else if ( norm_id == 1 ) {
        // ||v||_1 = |x| + |y| + |z|
        return std::abs( x_ ) + std::abs( y_ ) + std::abs( z_ );
    } else {
        // ||v||_inf = max{ |x|, |y|, |z| }
        assert( norm_id == 0 );
        return std::max( std::max( std::abs( x_ ), std::abs( y_ ) ),
                         std::abs( z_ ) );
    }
}

void Vector_3d::normalize( int const norm_id ) {
    double nv{ norm( norm_id ) };

    // Only normalize if it is not the zero vector
    if ( nv >= 1e-15 ) {
        x_ /= nv;
        y_ /= nv;
        z_ /= nv;
    }
}
```

The last function makes use of the fact that max$\{x, y, z\}$ = max$\{$max$\{x, y\}, z\}$, as the standard library implementation of the max function only takes two arguments.

Next, suppose that you want the default value of the choice of norm to be two. Recall how the default value must be specified in the function declaration? Member functions are *declared* in the class definition, so this is where default values must be specified:

```
// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;

        double norm( int const norm_id = 2 ) const;
        void normalize();
};
```

Thus, we can now call:

```
int main() {
    Vector_3d posn{4.5, 2.3, 5.8};

    // Here, the 'x_', 'y_' and 'z_' in the member function call refer to
    // the member variables of the object 'posn'
    std::cout << posn.norm() << std::endl;      // The same as posn.norm( 2 )
    std::cout << posn.norm( 1 ) << std::endl;
    std::cout << posn.norm( 0 ) << std::endl;

    return 0;
}
```

Assignment operator

If you assign one vector to another, the member variables are simply all copied over from the one object to the other:

```
int main() {
    Vector_3d posn{4.5, 2.3, 5.8};
    Vector_3d zero{0.0, 0.0, 0.0};

    std::cout << posn.norm() << std::endl;

    // This is the same as posn.x_ = zero.x_;
    //                      posn.y_ = zero.y_;
    //                      posn.z_ = zero.z_;
    posn = zero;

    std::cout << posn.norm( 1 ) << std::endl;
    std::cout << posn.norm( 0 ) << std::endl;

    return 0;
}
```

This assignment operator is provided for you by the compiler.

Given a class, the compiler will always define an assignment operator that simply copies over the values stored in the member variables of one instance to the instance on the left-hand side of the assignment operator. If this behavior is undesirable or naïve, you can define your own assignment operator.

Operator overloading

As you will recall from your course in linear algebra:

1. two vectors are equal if all of their corresponding entries are equal, and
2. two vectors are unequal if any of their corresponding entries are unequal.

Now, in C++, the standard way to compare equality is to use the == and != operators. With classes, we still can:

```
// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;

        double norm( int const norm_id = 2 ) const;
        void normalize();

        // The const indicates that the object this function
        // is being called on cannot be changed, and
        // the const indicates that the argument cannot be changed.
        bool operator==( Vector_3d const &v ) const;
        bool operator!=( Vector_3d const &v ) const;
};

bool Vector_3d::operator==( Vector_3d const &v ) const {
    return (x_ == v.x_) && (y_ == v.y_) && (z_ == v.z_);
}

bool Vector_3d::operator!=( Vector_3d const &v ) const {
    // This vector and 'v' are not equal if they are not equal:
    //  - This continues to call operator== on the same object that this
    //    function was called on
    return !operator==( v );

    // Alternative implementation...
    // return (x_ != v.x_) || (y_ != v.y_) || (z_ != v.z_);
}
```

These two are functions, just like `norm` and `normalize`, so we could do the following:

```
int main() {
    Vector_3d v1{2.3, 3.5, -5.7};
    Vector_3d v2{2.3, 3.5,  4.7};   // different from 'v1'
    Vector_3d v3{2.3, 3.5, -5.7};   // equal to 'v1'

    std::cout << v1.operator==( v2 ) << std::endl;
    std::cout << v1.operator!=( v2 ) << std::endl;

    std::cout << v1.operator==( v3 ) << std::endl;
    std::cout << v1.operator!=( v3 ) << std::endl;

    return 0;
}
```

The beauty of *operator overloading*, however, is that you can use these two operators infixed, as well:

```
int main() {
    Vector_3d v1{2.3, 3.5, -5.7};
    Vector_3d v2{2.3, 3.5,  4.7};   // different from 'v1'

    // 'v1 == v2' is the same as calling 'v1.operator==( v2 )'
    if ( v1 == v2 ) {
        std::cout << "Vectors 'v1' and 'v2' are equal" << std::endl;
    } else {
        assert( v1 != v2 );
        std::cout << "Vectors 'v1' and 'v2' are different" << std::endl;
    }

    // Copy all the member values from v1 over to v2
    //  - this is identical to v2.x_ = v1.x_;
    //                         v2.y_ = v1.y_;
    //                         v2.z_ = v1.z_;
    v2 = v1;

    // 'v1 == v2' is the same as calling 'v1.operator==( v2 )'
    if ( v1 == v2 ) {
        std::cout << "Vectors 'v1' and 'v2' are equal" << std::endl;
    } else {
        assert( v1 != v2 );
        std::cout << "Vectors 'v1' and 'v2' are different" << std::endl;
    }

    return 0;
}
```

We could, if we wanted, implement `operator<`, `operator<=`, `operator>=` and `operator>`, but there is no real definition for saying one vector is "greater" than another. In other applications, however, there may be such reasonable definitions.

Java does not have operator overloading. If you want to implement a function that returns true if two objects are equal, you must define a function with the name `equals(…)`.

Constructors

From linear algebra, you know that an entry in a vector cannot be infinity, and yet, the double-precision floating-point representation allows for not only plus-or-minus infinity, but also indeterminate floating-point numbers. Suppose you'd like to avoid this happening, but the user can always construct a vector with such values:

```
int main() {
    Vector_3d v{1.0/0.0, -1.0/0.0, std::sqrt(-1)};

    std::cout << "(" << v.x_ << ", " << v.y_ << ", " << v.z_ << ")"
              << std::endl;

    return 0;
}
```

To stop this, we can define a *constructor* that checks these values:

```
// Class definitions
class Vector_3d {
    public:
        double x_;
        double y_;
        double z_;

        Vector_3d( double const x, double const y, double const z );

        double norm( int const norm_id = 2 ) const;
        void normalize();

        bool operator==( Vector_3d const &v ) const;
        bool operator!=( Vector_3d const &v ) const;
};

Vector_3d::Vector_3d( double const x, double const y, double const z ):
x_{x},    // Initialize 'x_' with the value of the parameter 'x'
y_{y},    //      "      'y_'  "    "    "   "   "       "     'y'
z_{z} {   //      "      'z_'  "    "    "   "   "       "     'z'
    // Make an assertion that they are all finite.
    //  - not infinite and not indeterminant (NaN)
    assert ( std::isfinite( x_ ) && std::isfinite( y_ )
                                 && std::isfinite( z_ ) );
}
```

Now, with this constructor, the call

```
int main() {
    Vector_3d v{1.0/0.0, -1.0/0.0, std::sqrt(-1)};

    std::cout << "(" << v.x_ << ", " << v.y_ << ", " << v.z_ << ")"
              << std::endl;

    return 0;
}
```

will instead call the constructor with the arguments as they are passed. Now the assertion will fail and the program will terminate:

```
a.out: example.cpp:28: Vector_3d::Vector_3d(double, double, double): Assertion
`std::isfinite( x_ ) && std::isfinite( y_ ) && std::isfinite( z_ )' failed.
```

Copy constructor

Now that we've introduced the constructor, let us consider the following code:

```
int main() {
    Vector_3d v{3.2, 4.7, 6.8};
    Vector_3d u{v};

    // Changes to 'v' no longer affect 'u'
    v.x = 99.99;

    std::cout << "u = (" << u.x_ << ", " << u.y_ << ", " << u.z_ << ")"
                    << std::endl;

    return 0;
}
```

Here, we are declaring u to be a new vector, but its initial value is v. The only reasonable interpretation is that the member variables of u are assigned the member variables of v, and that's exactly what happens: u is a different vector from v and the above code is *essentially* identical to:

```
int main() {
    Vector_3d v{3.2, 4.7, 6.8};
    Vector_3d u{v.x_, v.y_, v.z_};

    // Changes to 'v' no longer affect 'u'
    v.x = 99.99;

    std::cout << "u = (" << u.x_ << ", " << u.y_ << ", " << u.z_ << ")"
                    << std::endl;

    return 0;
}
```

Given a class, the compiler will always define a copy constructor that simply copies over the values stored in the instance being copied into the instance being declared. If this behavior is undesirable or naïve, you can define your own assignment operator.

Encapsulation

Unfortunately, adding security in the constructor does not stop the user from changing the values later:

```
int main() {
    Vector_3d v{0.0, 0.0, 0.0};

    // The user is trying to by-pass our security!
    v.x =  1.0/0.0;
    v.y = -1.0/0.0;
    v.z = std::sqrt(-1)};

    std::cout << "(" << v.x_ << ", " << v.y_ << ", " << v.z_ << ")"
              << std::endl;

    return 0;
}
```

To prevent the user from by-passing our security, we will make a few changes:

1. we will prevent outside users from accessing or modifying member variables, and
2. we will introduce functions to access or modify the member variables, always checking to see that they are never set to non-finite values.

The first step to encapsulation is to make all member variables private:

```
// Class definitions
class Vector_3d {
    public:
        Vector_3d( double const x, double const y, double const z );

        double norm( int norm_id = 2 ) const;
        void normalize();

        bool operator==( Vector_3d const &v ) const;
        bool operator!=( Vector_3d const &v ) const;

    private:
        double x_;
        double y_;
        double z_;
};
```

Users other than the author of the class can only access member variables or call member functions that are public. When a member function is called, all member variables of any instance of this class are easily accessible, as they should be: it is assumed the author of the class knows how to work with these variables.

Right now, however, the user cannot access or modify the entries of the vector. This may be a problem, as a position vector or velocity vector may change.

If the users want to access or modify these member variables, they can do so with appropriately named member functions that the author of the class has authored and verified.

```
// Class definitions
class Vector_3d {
    public:
        Vector_3d( double x, double y, double z );

        double x() const;
        void x( double const new_x );
        double y() const;
        void y( double const new_y );
        double z() const;
        void z( double const new_z );

        double norm( int const norm_id = 2 ) const;
        void normalize();

        bool operator==( Vector_3d const &v ) const;
        bool operator!=( Vector_3d const &v ) const;

    private:
        double x_;
        double y_;
        double z_;
};
```

These functions are also intimately linked to the class definition by the `Vector_3d::`, as can be seen here:

```
double Vector_3d::x() const {
    return x_;
}

void Vector_3d::x( double const new_x ) {
    assert( std::isfinite( new_x ) );
    x_ = new_x;
}

double Vector_3d::y() const {
    return y_;
}

void Vector_3d::y( double const new_y ) {
    assert( std::isfinite( new_y ) );
    y_ = new_y;
}

double Vector_3d::z() const {
    return z_;
}

void Vector_3d::z( double const new_z ) {
    assert( std::isfinite( new_z ) );
    z_ = new_z;
}
```

Now the user can access and modify these member variables, but under no circumstances is the user allowed to modify the values to a forbidden value.

Some of you may say, “Wait a second, aren’t function calls more expensive than just accessing member variables?” This is true, as member functions require that the call stack is prepared and function calls are made. Fortunately, C++ allows the compiler to be *clever*. It can write code by hard-coding the function call in the place where the function is called.

The previous code we wrote can now be implemented as:

```
int main() {
    Vector_3d v{4.5, -2.7, 8.6};

    std::cout << "(" << v.x() << ", " << v.y() << ", " << v.z() << ")"
              << std::endl;

    v.y( 9.1 );   // Set 'y_' to the value 9.1

    std::cout << "(" << v.x() << ", " << v.y() << ", " << v.z() << ")"
              << std::endl;

    return 0;
}
```

Strings and printing

Instead of creating a printing function, we will create a function that returns a string. That string can then be printed.

```
#include <string>

// Class definitions
class Vector_3d {
    public:
        // ... a whole bunch of public member functions ...

        std::string to_string() const;
};

std::string Vector_3d::to_string() const {
    return "(" + std::to_string( x_ ) + ", " + std::to_string( y_ ) + ", "
               + std::to_string( z_ ) + ")";
}
```

Private helper functions

As we are demanding that all member variables are never infinity or indeterminate, it may be necessary to periodically check if a vector is indeed finite. There is no reason for the user to ever use this function, because it is of no potential use to the user, but the author of this class may need it. Thus, we introduce private member functions that can only be called by other member functions:

```
// Class definitions
class Vector_3d {
    public:
        // ... a whole bunch of public member functions ...

    private:
        double x_;
        double y_;
        double z_;

        bool isfinite() const;
};

bool Vector_3d::isfinite() const {
    return std::isfinite( x_ ) && std::isfinite( y_ ) && std::isfinite( z_ );
}
```

Because this member function is declared private, users cannot call it.

this

Previously, when something was passed by reference, it was still possible to access the address of the item being passed:

```
// In the parameter declaration, '&x' means 'x' is passed by reference.
void f( double &x );

void f( double &x ) {
    // Print out the address of the variable that was passed by reference
    std::cout << &x << std::endl;
}
```

In all of these member functions, however, you can only access the member variables. It is, however, possible to also access the address of the object the member function is being called on by a special local variable called `this`. This is a constant pointer that stores an address:

```
bool Vector_3d::operator==( Vector_3d const &v ) const {
    return (this == &v) || (
        (x_ == v.x_) && (y_ == v.y_) && (z_ == v.z_)
    );
}
```

This may look a little peculiar, but this says "if the address of this equals the address of the argument `'v'`, then then the two vectors must be equal. Thus, if you were to use this implementation, code like

```
Vector_3d posn{3.2, 5.7, -9.2};

if ( posn == posn ) {
    std::cout << "These are equal" << std::endl;
}
```

will use short-circuit evaluation to immediately return true when it is determined the address are indeed equal.

Additional operator overloading

As you are familiar, there are two vector space operations: scalar multiplication and vector addition. We may want the user to add two vectors, or multiply a vector by a scalar, so we can also introduce new operators to achieve this:

```
// Class definitions
class Vector_3d {
    public:
        Vector_3d( double x, double y, double z );

        // ...other member functions...

        Vector_3d operator+( Vector_3d const &v ) const;
        Vector_3d operator-( Vector_3d const &v ) const;
        Vector_3d operator*( double const s ) const;
        Vector_3d operator/( double const s ) const;

    private:
        double x_;
        double y_;
        double z_;
};
```

Note that **u** + **v** should evaluate to a new vector, not changing either **u** or **v**, and calculating *s***u** should again evaluate to a new vector that contains entries of **u** multiplied by *s*:

```
Vector_3d Vector_3d::operator+( Vector_3d const &v ) const {
    Vector_3d new_vec{ *this };
    new_vec.x_ += v.x_;
    new_vec.y_ += v.y_;
    new_vec.z_ += v.z_;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

Vector_3d Vector_3d::operator-( Vector_3d const &v ) const {
    Vector_3d new_vec{ *this };
    new_vec.x_ -= v.x_;
    new_vec.y_ -= v.y_;
    new_vec.z_ -= v.z_;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}
```

```
Vector_3d Vector_3d::operator*( double const s ) const {
    assert( std::isfinite( s ) );

    Vector_3d new_vec{ *this };
    new_vec.x_ *= s;
    new_vec.y_ *= s;
    new_vec.z_ *= s;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

Vector_3d Vector_3d::operator/( double const s ) const {
    assert( std::isfinite( s ) && (s != 0.0) ); // We cannot divide by zero

    Vector_3d new_vec{ *this };
    new_vec.x_ /= s;
    new_vec.y_ /= s;
    new_vec.z_ /= s;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}
```

All of these member functions contain a new feature you've probably never seen before:

```
    Vector_3d new_vec{ *this };
```

Recall from a previous section that `this` is a pointer storing the address of object on which this member function is being called on. Dereferencing that point by prefixing it by an asterisk is therefore a reference to the actual object, so this is a call to the *copy constructor*, so `new_vec` is now a new vector assigned the entries of the member variables of the object this was created by.

Next, that new vector is no longer `const`, so its member variables can be changed. Finally, the new vector is returned. Thus, we can do the following:

```
int main() {
    Vector_3d u{4.5, -2.7,  8.6};
    Vector_3d v{2.9,  0.5, -4.7};
    Vector_3d w1{u + v};
    Vector_3d w2{u*3.57};

    std::cout << "(" << w1.x() << ", " << w1.y() << ", " << w1.z() << ")"
              << std::endl;

    std::cout << "(" << w2.x() << ", " << w2.y() << ", " << w2.z() << ")"
              << std::endl;

    w2 = u*3.5 + v*2.7; // 'w2' is assigned a linear combination of 'u' and 'v'

    std::cout << "(" << w2.x() << ", " << w2.y() << ", " << w2.z() << ")"
              << std::endl;

    return 0;
}
```

It is also possible to implement the auto-assignment operators += and *= in a similar manner:

```
// Class definitions
class Vector_3d {
    public:
        Vector_3d( double x, double y, double z );

        // ...other member functions...

        Vector_3d operator+( Vector_3d const &v ) const;
        Vector_3d operator-( Vector_3d const &v ) const;
        Vector_3d operator*( double const s ) const;
        Vector_3d operator/( double const s ) const;

        Vector_3d &operator+=( Vector_3d const &v );
        Vector_3d &operator-=( Vector_3d const &v );
        Vector_3d &operator*=( double const s );
        Vector_3d &operator/=( double const s );

    private:
        double x_;
        double y_;
        double z_;
};
```

You will notice that all of these operators return a reference to the vector that was changed. This mimics the behavior of these operators for primitive data types. Implementing these, we have:

```
Vector_3d &Vector_3d::operator+=( Vector_3d const &v ) {
    // Add 'v' onto this vector
    x_ += v.x_;
    y_ += v.y_;
    z_ += v.z_;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}

Vector_3d &Vector_3d::operator-=( Vector_3d const &v ) {
    // Subtract 'v' from this vector
    x_ -= v.x_;
    y_ -= v.y_;
    z_ -= v.z_;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}
```

```
Vector_3d &Vector_3d::operator*=( double const s ) {
    // Multiply this vector by 's'
    x_ *= s;
    y_ *= s;
    z_ *= s;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}

Vector_3d &Vector_3d::operator/=( double const s ) {
    // Divide this vector by 's'
    x_ /= s;
    y_ /= s;
    z_ /= s;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}
```

Other operators you may want to overload are the unary + and - operators:

```
// Class definitions
class Vector_3d {
    public:
        Vector_3d( double x, double y, double z );

        // ...other member functions...

        Vector_3d operator+() const;
        Vector_3d operator-() const;

    private:
        double x_;
        double y_;
        double z_;
};

// Return a copy of this object
Vector_3d Vector_3d::operator+() const {
    return Vector_3d{ *this };
}

// Return this object multiplied by -1
Vector_3d Vector_3d::operator-() const {
    return *this * (-1.0);
}
```

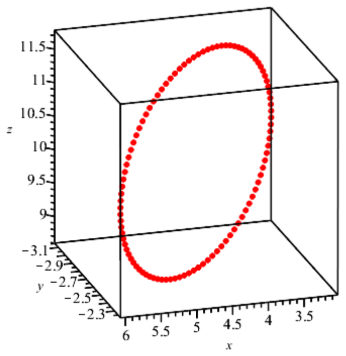
We can use this as follows:

```
int main() {
    // Absolute position in m
    Vector_3d u{4.5, -2.7,  8.6};
    std::cout << u.to_string() << std::endl;

    for ( std::k{0}; k < 100; ++k ) {
        double theta{0.3};
        double phi{0.01*k*2*M_PI};
        // Velocity vector in m/s

        Vector_3d v{std::cos(theta)*std::cos(phi),
                    std::sin(theta)*std::cos(phi),
                    std::sin(phi)};

        u += 0.1*v;
        std::cout << u.to_string() << std::endl;
    }

    return 0;
}
```

This prints out a sequence of 101 points:

```
(4.500000, -2.700000, 8.600000)
(4.595534, -2.670448, 8.600000)
(4.690879, -2.640954, 8.606279)
               ⋮
(4.309875, -2.758813, 8.618812)
(4.404655, -2.729494, 8.606279)
(4.500000, -2.700000, 8.600000)
```

We can plot these:



Other functions

There are two weaknesses in the above implementation:

1. You cannot write `0.3*v` when `v` is a vector. This is because if you write `v*0.3`, the compiler will immediately link this to `v.operator*( 0.3 )`, and there is a member function that matches the type of the argument
   ```
   Vector_3d operator*( double const s );
   ```
   however, you cannot define a member function on a primitive data type.
2. It would be nice to use `std::cout << v` and have the vector printed as desired, instead of using `std::cout << v.to_string()`.

We can implement both of these as additional functions:

```
// Class definitions
class Vector_3d {
    public:
        // Public member functions...

    private:
        // Private member variables and private helper functions...
};

// Function declarations
Vector_3d operator*( double const s, Vector_3d const &v );
std::ostream &operator<<( std::ostream &out, Vector_3d const &v );

// Member function definitions...

// Function definitions
Vector_3d operator*( double const s, Vector_3d const &v ) {
    // This calls and returns v.operator*( s );
    return v*s;
}

std::ostream &operator<<( std::ostream &out, Vector_3d const &v ) {
    return out << v.to_string();
}
```

You will note that these two functions are not declared in the `Vector_3d` class. Instead, they are global functions.

The full class

This full class is available at https://repl.it/@dwharder/Vector3d.

```
#include <iostream>
#include <string>
#include <cassert>
#include <cmath>

// Class declarations
class Vector_3d;

// Function declarations
int main();

// Class definitions
class Vector_3d {
    public:
        Vector_3d( double const x = 0.0, double const y = 0.0, double const z = 0.0 );
        // Vector_3d( Vector const &v ); Uses the default copy constructor...

        double x() const;
        void x( double const new_x );
        double y() const;
        void y( double const new_y );
        double z() const;
        void z( double const new_z );

        std::string to_string() const;

        double norm( int const norm_id = 2 ) const;
        void normalize( int const norm_id = 2 );

        bool operator==( Vector_3d const &v ) const;
        bool operator!=( Vector_3d const &v ) const;

        Vector_3d operator+( Vector_3d const &v ) const;
        Vector_3d operator-( Vector_3d const &v ) const;
        Vector_3d operator*( double const s ) const;
        Vector_3d operator/( double const s ) const;
        Vector_3d operator-() const;
        Vector_3d operator+() const;

        // Vector_3d &operator=( Vector_3d const &v ); Uses the default assignment operator...

        Vector_3d &operator+=( Vector_3d const &v );
        Vector_3d &operator-=( Vector_3d const &v );
        Vector_3d &operator*=( double const s );
        Vector_3d &operator/=( double const s );

    private:
        double x_;
        double y_;
        double z_;

        bool isfinite() const;
};

// Function declarations
Vector_3d operator*( double const s, Vector_3d const &v );
std::ostream &operator<<( std::ostream &out, Vector_3d const &v );
```

```
  /////////////////////////////////
 // Member function definitions //
/////////////////////////////////

// Constructor
Vector_3d::Vector_3d( double const x, double const y, double const z ):
x_{x},    // Initialize 'x_' with the value of the parameter 'x'
y_{y},    //      "     'y_'  "    "    "   "   "      "     'y'
z_{z} {   //      "     'z_'  "    "    "   "   "      "     'z'
    // Make an assertion that they are all finite.
    //  - not infinite and not indeterminant (NaN)
    assert ( std::isfinite( x_ ) && std::isfinite( y_ )
                                 && std::isfinite( z_ ) );
}

  ////
 // Named member functions
////

// Convert this vector to a string in the form
//     (123.4567, 234.5678, -345.6789)
std::string Vector_3d::to_string() const {
    return "(" + std::to_string( x_ ) + ", " + std::to_string( y_ ) + ", "
               + std::to_string( z_ ) + ")";
}

// Calculate the 1-, 2- or infinity-norm of this vector
double Vector_3d::norm( int const norm_id ) const {
    assert( (norm_id == 0) || (norm_id == 1) || (norm_id == 2) );

    if ( norm_id == 2 ) {
        //                _____________
        //               / 2    2    2
        // ||v||_2 = \/ x  + y  + z
        return std::sqrt( x_*x_ + y_*y_ + z_*z_ );
    } else if ( norm_id == 1 ) {
        // ||v||_1 = |x| + |y| + |z|
        return std::abs( x_ ) + std::abs( y_ ) + std::abs( z_ );
    } else {
        // ||v||_inf = max{ |x|, |y|, |z| }
        assert( norm_id == 0 );
        return std::max( std::max( std::abs( x_ ), std::abs( y_ ) ),
                         std::abs( z_ ) );
    }
}

// Normalize this vector using the 1-, 2-, or infinity-norm
void Vector_3d::normalize( int const norm_id ) {
    double nv{ norm( norm_id ) };

    // Only normalize if it is not the zero vector
    if ( nv >= 1e-15 ) {
        x_ /= nv;
        y_ /= nv;
        z_ /= nv;
    }
}
```

```
  ////
 // Accessors and mutators
////

// Access the member variable 'x'
double Vector_3d::x() const {
    return x_;
}

// Uptdate the member variable 'x' with the value 'new_x'
//  - check the new value is not infinity or NaN
void Vector_3d::x( double const new_x ) {
    assert( std::isfinite( new_x ) );
    x_ = new_x;
}

// Access the member variable 'y'
double Vector_3d::y() const {
    return y_;
}

// Update the member variable 'y' with the value 'new_y'
//  - check the new value is not infinity or NaN
void Vector_3d::y( double const new_y ) {
    assert( std::isfinite( new_y ) );
    y_ = new_y;
}

// Access the member variable 'z'
double Vector_3d::z() const {
    return z_;
}

  ////
 // Boolean-valued operators
////

// Update the member variable 'z' with the value 'new_z'
//  - check the new value is not infinity or NaN
void Vector_3d::z( double const new_z ) {
    assert( std::isfinite( new_z ) );
    z_ = new_z;
}

// Check if this vector equals 'v'
bool Vector_3d::operator==( Vector_3d const &v ) const {
    return (x_ == v.x_) && (y_ == v.y_) && (z_ == v.z_);
}

// Check if this vector does not equals 'v'
bool Vector_3d::operator!=( Vector_3d const &v ) const {
    // This vector and 'v' are not equal if they are not equal:
    //  - This continues to call operator== on the same object that this
    //    function was called on
    return !operator==( v );

    // Alternative implementation...
    // return (x_ != v.x_) || (y_ != v.y_) || (z_ != v.z_);
}
```

```
  ////
 // Vector addition and scalar multiplication
////

// Add this vector and 'v'
Vector_3d Vector_3d::operator+( Vector_3d const &v ) const {
    Vector_3d new_vec{ *this };
    new_vec.x_ += v.x_;
    new_vec.y_ += v.y_;
    new_vec.z_ += v.z_;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

// Return the result of subtracting 'v' from this vector
Vector_3d Vector_3d::operator-( Vector_3d const &v ) const {
    Vector_3d new_vec{ *this };
    new_vec.x_ -= v.x_;
    new_vec.y_ -= v.y_;
    new_vec.z_ -= v.z_;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

// Return this vector multiplied by 's'
Vector_3d Vector_3d::operator*( double const s ) const {
    assert( std::isfinite( s ) );

    Vector_3d new_vec{ *this };
    new_vec.x_ *= s;
    new_vec.y_ *= s;
    new_vec.z_ *= s;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

// Return this vector divided by 's'
Vector_3d Vector_3d::operator/( double const s ) const {
    assert( std::isfinite( s ) && (s != 0.0) ); // We cannot divide by zero

    Vector_3d new_vec{ *this };
    new_vec.x_ /= s;
    new_vec.y_ /= s;
    new_vec.z_ /= s;
    assert( new_vec.isfinite() ); // Make sure the new vector is still finite

    return new_vec;
}

// Return a copy of this object
Vector_3d Vector_3d::operator+() const {
    return Vector_3d{ *this };
}

// Return this object multiplied by -1
Vector_3d Vector_3d::operator-() const {
    return *this * (-1.0);
}
```

```
  ////
 // Auto-assignment operations for vector addition and scalar multiplication
////

// Add 'v' to this vector and ensure it is finite
Vector_3d &Vector_3d::operator+=( Vector_3d const &v ) {
    // Add 'v' onto this vector
    x_ += v.x_;
    y_ += v.y_;
    z_ += v.z_;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}

// Subtract 'v' from this vector and ensure it is finite
Vector_3d &Vector_3d::operator-=( Vector_3d const &v ) {
    // Subtract 'v' from this vector
    x_ -= v.x_;
    y_ -= v.y_;
    z_ -= v.z_;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}

// Multiply this vector 's' and ensure it is finite
Vector_3d &Vector_3d::operator*=( double const s ) {
    // Multiply this vector by 's'
    x_ *= s;
    y_ *= s;
    z_ *= s;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}

// Divide each entry by 's' and ensure it is finite
Vector_3d &Vector_3d::operator/=( double const s ) {
    // Divide this vector by 's'
    x_ /= s;
    y_ /= s;
    z_ /= s;
    assert( isfinite() ); // Make sure this vector is still finite

    return *this;
}
```

```
  //////////////////////////////
 // Private helper functions //
//////////////////////////////

// Ensure all the entries of 'v' are finite (not infinity and not NaN)
bool Vector_3d::isfinite() const {
    return std::isfinite( x_ ) && std::isfinite( y_ ) && std::isfinite( z_ );
}

  //////////////////////////
 // Function definitions //
//////////////////////////

// Calculate the scalar multiplication of s*v
Vector_3d operator*( double const s, Vector_3d const &v ) {
    // This calls and returns v.operator*( s );
    return v*s;
}

// Print the vector by printing the string
std::ostream &operator<<( std::ostream &out, Vector_3d const &v ) {
    return out << v.to_string();
}
```