1

Outline

- In this lesson, we will:
  - Discuss the need for restricting access to member variables
  - Describe how the compiler helps with overcoming additional costs
  - See how making all member variables private prevents future changes to your class from harming your existing user base
  - Define the idea of an interface to a class

2

Review of encapsulation

- Let us look at the `Vector_3d` class
  - It has three member variables: `x_`, `y_` and `z_`

- Why not leave them public and just call them `x`, `y` and `z`?

```
class Vector_3d {
    public:
        double x;
        double y;
        double z;
};
```

- After all, this would be easier and simpler, as there are absolutely no restrictions on their values
  - Nothing the user sets any of these to will affect the object

3

Using member functions

- For example, why waste the time to do the following?

```
class Vector_3d {
    public:
        Vector_3d{ double new_x = 0.0,
                   double new_y = 0.0,
                   double new_z = 0.0 };

         double x();
         double y();
         double z();
         void x( double new_x );
         void y( double new_y );
         void z( double new_z );
    private:
        double x_;
        double y_;
        double z_;
};
```

4

Using member functions

- One problem is, you are not clairvoyant,
  you cannot predict the future
- Once users are accessing your member variables,
  you can no longer change your underlying design of the class
- Suppose we do it the "hard" way:
  - Every time the user wants a member variable,
    the user must call the appropriate member function
  - However, that's not true:
    - The compiler can decide to *inline* very short function calls
    - That is, the compiler will replace the function call with code that achieves what the function call would have done

Our implementation

```
class Vector_3d {
    public:
        Vector_3d( double new_x = 0.0,
                   double new_y = 0.0,
                   double new_z = 0.0 );
         double x() const;
         double y() const;
         double z() const;
         void x( double new_x );
         void y( double new_y );
         void z( double new_z );
    private:
        double x_;
        double y_;
        double z_;
};

Vector_3d::Vector_3d( double new_x,
        double new_y, double new_z ) {
x_{ new_x },
y_{ new_y },
z_{ new_z } {
    // Empty constructor
}
```

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

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

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

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

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

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


Using member functions

- Suppose after using this class for a few months,
  you determine it is better to use an array
  instead of three separate member variables
  - If your users were all using the public member variables,
    your change would break all their code!
  - This does not make any users happy!
- However, if you changed the underlying representation to an array and all users were always calling the member functions anyway…
  - You just have to change how the member functions work

Our implementation

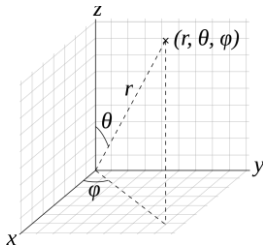
```
class Vector_3d {
    public:
        Vector_3d( double new_x = 0.0,
                   double new_y = 0.0,
                   double new_z = 0.0 );
         double x() const;
         double y() const;
         double z() const;
         void x( double new_x );
         void y( double new_y );
         void z( double new_z );
    private:
        double xyz_[3];
};

Vector_3d::Vector_3d( double new_x,
        double new_y, double new_z ) {
xyz_{ new_x, new_y, new_z } {
    // Empty constructor
}
```

```
double Vector_3d::x() const {
    return xyz_[0];
}

double Vector_3d::y() const {
    return xyz_[1];
}

double Vector_3d::z() const {
    return xyz_[2];
}

void Vector_3d::x( double new_x ) const {
    xyz_[0] = new_x;
}

void Vector_3d::y( double new_y ) const {
    xyz_[1] = new_y;
}

void Vector_3d::z( double new_z ) const {
    xyz_[2] = new_z;
}
```

s

Private member variables

- However, suppose you also implement the following member functions to return the spherical coordinates:



Wikipedia user: Andeggs

Our implementation

```
class Vector_3d {
    public:
        Vector_3d( double new_x = 0.0,
                   double new_y = 0.0,
                   double new_z = 0.0 );
        double x() const;
        double y() const;
        double z() const;
        double r() const;
        double theta() const;
        double phi() const;

        void x( double new_x );
        void y( double new_y );
        void z( double new_z );
        void r( double new_r );
        void theta( double new_theta );
        void phi( double new_phi );
    private:
        double x_;
        double y_;
        double z_;
        void to_rect( double r, double theta, double phi );
};
```


Our implementation

```
double Vector_3d::r() const {
    return std::sqrt( x()*x() + y()*y() + z()*z() );
}

double Vector_3d::theta() const {
    return std::acos( z()/r() );
}

double Vector_3d::phi() const {
    return std::atan2( y(), x() );
}

void Vector_3d::to_rect( double r, double theta, double phi ) {
    x_ = r*std::sin( theta )*std::cos( phi );
    y_ = r*std::sin( theta )*std::sin( phi );
    z_ = r*std::cos( theta );
}
```


Our implementation

```
void Vector_3d::r( double new_r ) const {
    to_rect( new_r, theta(), phi() );
}

void Vector_3d::theta( double new_theta ) const {
    to_rect( r(), new_theta, phi() );
}

void Vector_3d::phi( double new_phi ) const {
    to_rect( r(), theta(), new_phi );
}
```

Private member variables

- Now, your 3-dimensional vector class is being used;
  however, more and more calculations simply require the spherical coordinates and not the rectangular coordinates
- If the member variables were public, there is nothing you could do…
  - However, if they're private, just change them and rewrite the corresponding member functions
    - From a compilation point-of-view, no one will note the difference

Our implementation

```
class Vector_3d {
    public:
        Vector_3d( double new_x = 0.0,
                   double new_y = 0.0,
                   double new_z = 0.0 );
        double x() const;
        double y() const;
        double z() const;
        double r() const;
        double theta() const;
        double phi() const;

        void x( double new_x );
        void y( double new_y );
        void z( double new_z );
        void r( double new_r );
        void theta( double new_theta );
        void phi( double new_phi );
    private:
        double r_;
        double theta_;
        double phi_;
        void to_spherical( double x, double y, double z );
};
```


Our implementation

```
Vector_3d::Vector_3d( double x, double y, double z ) {
r_{ std::sqrt( x*x + y*y + z*z ) },
theta_{ std::acos( z/r_ ) },
phi_{ std::atan2( y, x ) } {
    // Empty constructor
}

double Vector_3d::x() const {
    return r_*std::sin( theta_ )*std::cos( phi_ );
}

double Vector_3d::y() const {
    return r_*std::sin( theta_ )*std::sin( phi_ );
}

double Vector_3d::z() const {
    return r_*std::cos( theta_ );
}
```


Our implementation

```
void Vector_3d::x( double new_x ) const {
    to_spherical( new_x, y(), z() );
}

void Vector_3d::y( double new_y ) const {
    to_spherical( x(), new_y, z() );
}

void Vector_3d::z( double new_z ) const {
    to_spherical( x(), y(), new_z );
}
```

Our implementation

```
double Vector_3d::r() const {
    return r_;
}

double Vector_3d::theta() const {
    return theta_;
}

double Vector_3d::phi() const {
    return phi_;
}
```

Our implementation

```
void Vector_3d::r( double new_r ) const {
    if ( new_r < 0.0 ) {
        throw std::domain_error{ "The radius cannot be negative, but got “
                                 + std::to_string( new_r ) };
    }
    r_ = new_r;
}

void Vector_3d::theta( double new_theta ) const {
    if ( (new_theta < 0.0) || (new_theta > M_PI) ) {
        throw std::domain_error{ "The inclination must be in [0, pi], but got "
                                 + std::to_string( new_theta ) };
    }
    theta_ = new_theta;
}

void Vector_3d::phi( double new_phi ) const {
    if ( (new_phi < 0.0) || (new_phi >= 2.0*M_PI) ) {
        throw std::domain_error{ "The azimuth must be in [0, 2 pi), but got "
                                 + std::to_string( new_phi ) };
    }
    phi_ = new_phi;
}
```

Member functions and interfaces

- Now, the scenario presented is not likely,
  but it should make you recognize the usefulness of restricting access to member variables, even if initially it seems unnecessary
- Also, if a function is very short, the compiler may simply decide to *inline* the function, meaning, replace the function call with equivalent code in place of the function call
  - Thus, there is no effect on run time what-so-ever
- Another term for the collection of all member functions that allow the user to access and manipulate an instance of a class is the *interface* for that class

Summary

- Following this lesson, you now
  - Understand the need for encapsulation
  - Know that the compiler will help ameliorate the potential impact of having member function calls through inlining
  - Know that another term for all member functions that allow you to access a class is the *interface*

References

[1] https://en.wikipedia.org/wiki/C++_classes
[2] https://en.wikipedia.org/wiki/Inline_expansion
[3] https://en.wikipedia.org/wiki/Spherical_coordinate_system
[4] https://en.wikipedia.org/wiki/Interface_(computing)

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.