1

Outline

- In this lesson, we will:
  - Look at functions that investigate and manipulate objects
  - Describe functions that investigate or query objects
    - That is, simply access values stored in member variables
  - Describe functions that manipulate or modify objects
    - That is, modify the values stored in member variables
  - We will look at numerous examples, including:
    - A 3-dimensional vector class
    - A rational number class
    - An array class
    - A pair class

2

Functions on vectors

- Consider this `Vector_3d` class

```
class Vector_3d;

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

3

Functions on vectors

- We will begin with functions that deal with vectors
  - The inner product and the 2-norm
  - The sum of two vectors and scalar multiplication
  - The cross product and the projection

4

The inner product and the norm

- For example, consider these functions:

```
double inner_product( Vector_3d const &u, Vector_3d const &v );
double norm( Vector_3d const &u );

double inner_product( Vector_3d const &u, Vector_3d const &v ) {
    return u.x_*v.x_ + u.y_*v.y_ + u.z_*v.z_;
}

double norm( Vector_3d const &u ) {
    return std::sqrt( inner_product( u, u ) );
}
```


Vector addition and scalar multiplication

- Here are the vector operations:

```
Vector_3d add( Vector_3d const &u, Vector_3d const &v );
Vector_3d mul( double const a, Vector_3d const &u );
Vector_3d mul( Vector_3d const &u, double const a );

Vector_3d add( Vector_3d const &u, Vector_3d const &v ) {
    return Vector_3d{ u.x_ + v.x_, u.y_ + v.y_,  u.z_*v.z_ };
}

Vector_3d mul( double const a, Vector_3d const &u ) {
    return Vector_3d{ a*u.x_, a*u.y_, a*u.z_ };
}

Vector_3d mul( Vector_3d const &u, double const a ) {
    return mul( a, u );
}
```


Projection and cross product

$$\text{proj}_{\mathbf{v}}(\mathbf{u}) = \frac{\langle \mathbf{v}, \mathbf{u} \rangle}{\langle \mathbf{v}, \mathbf{v} \rangle}\mathbf{v} = \frac{\mathbf{v} \cdot \mathbf{u}}{\mathbf{v} \cdot \mathbf{v}}\mathbf{v}$$

- Here are the projection and the cross product:

```
Vector_3d proj( Vector_3d const &u, Vector_3d const &v );
Vector_3d cross( Vector_3d const &u, Vector_3d const &v );

// Project the vector 'u' onto the vector 'v'
Vector_3d proj( Vector_3d const &u, Vector_3d const &v ) {
    return mul( inner( v, u )/inner( v, v ), v );
}

Vector_3d cross(Vector_3d const &u, Vector_3d const &v ) {
    return Vector_3d{ u.y_*v.z_ - u.z_*v.y_,
                      u.z_*v.x_ - u.x_*v.z_,
                      u.x_*v.y_ - u.y_*v._z };
}
```


Comparisons

- We may also want to test if two vectors are equal:

```
bool equal( Vector_3d const &u, Vector_3d const &v );

// Two vectors are equal if all their entries are equal
bool equal( Vector_3d const &u, Vector_3d const &v ) {
    return (u.x_ == v.x_) && (u.y_ = v.y_) && (u.z_ == v.z_);
}
```

Modifying a vector

- Lets look at some functions that modify a vector:
  - Normalizing a vector
  - Adding a vector onto a given vector
  - Multiplying a given vector by a scalar

Modifying a vector

- To modify a vector, it is necessary to pass it by reference:

```
void normalize( Vector_3d &u );

void normalize( Vector_3d &u ) {
    double norm_u{ norm( u ) };

    if ( norm_u != 0.0 ) {
        u.x_ /= norm_u;
        u.y_ /= norm_u;
        u.z_ /= norm_u;
    }
}
```

Modifying a vector

- To modify a vector, it is necessary to pass it by reference:

```
void add_to( Vector_3d &u, Vector_3d const &v );
void mul_by( Vector_3d &u, double const a );

void add_to( Vector_3d &u , Vector_3d const &v ) {
    u.x_ += v.x_;
    u.y_ += v.y_;
    u.z_ += v.z_;
}

void mul_by( Vector_3d &u , double const a ) {
    u.x_ *= a;
    u.y_ *= a;
    u.z_ *= a;
}
```

Rational number class

- Next, let's consider a rational number class
  - Now we will store the numerator and denominator as integers
  - The operations on rational numbers are more involved
- The class itself is straight-forward:

```
class Rational {
    public:
        int numer_;
        int denom_;
};
```

Rational number class

- Now, from your course in linear algebra, you understand that the properties of vectors differ from the properties of fields
  - Rational numbers, real numbers and complex numbers

Rational number class

- We can now implement functions that work on rational numbers:

```
Rational add( Rational const &p, Rational const &q );
Rational add( Rational const &p, int const n );

Rational add( Rational const &p, Rational const &q ) {
    return Rational{
        p.numer_*q.denom_ + p.denom_*q.numer_,
        p.denom_*q.denom_
    };
}

Rational add( Rational const &p, int const n ) {
    return add( p, Rational{ n, 1 } );
}
```

$$\frac{p_n}{p_d} + \frac{q_n}{q_d} = \frac{p_n q_d + p_d q_n}{p_d q_d}$$

Rational number class

- We can now implement functions that work on rational numbers:

```
Rational mul( Rational const &p, Rational const &q );
Rational mul( Rational const &p, int const n );

Rational mul( Rational const &p, Rational const &q ) {
    return Rational{ p.numer_*q.numer_, p.denom_*q.denom_ };
}

Rational mul( Rational const &p, int const n ) {
    return mul( p, Rational{ n, 1 } );
}
```

Rational number class

- We can now implement functions that work on rational numbers:

```
Rational negate( Rational const &p );
Rational inverse( Rational const &p );

Rational negate( Rational const &p ) {
    return Rational{ -p.numer_, p.denom_ };
}

Rational inverse( Rational const &p ) {
    assert( p.numer_ != 0 );
    return Rational{ p.denom_, p.numer_ };
}
```

Rational number class

- We can now implement functions that work on rational numbers:

```
Rational abs( Rational const &p );

// Requires 'cmath' library
Rational abs( Rational const &p ) {
    return Rational{ std::abs( p.numer_ ),
                     std::abs( p.denom_ ) };
}
```

Rational number class

- We can now implement functions that work on rational numbers:

```
bool equal( Rational const &p, Rational const &q );
bool equal( Rational const &p, int const n );

bool equal( Rational const &p, Rational const &q ) {
    return p.numer_*q.denom_ == p.denom_*q.numer_;
}

bool equal( Rational const &p, int const n ) {
    return equal( p, Rational{ n, 1 } );
}
```

$$\frac{p_n}{p_d} = \frac{q_n}{q_d} \quad \Leftrightarrow \quad p_n q_d = q_n p_d$$

Rational number class

- How about comparisons?

```
bool less_than( Rational const &p, Rational const &q );

bool less_than( Rational const &p, Rational const &q ) {
    return p.numer_*q.denom_ < p.denom_*q.numer_;
}
```

$$\frac{p_n}{p_d} < \frac{q_n}{q_d} \quad \Leftrightarrow \quad p_n q_d < q_n p_d$$

$$\frac{1}{-1} < \frac{1}{1} \quad \Leftrightarrow \quad 1 \cdot 1 < 1 \cdot -1$$

$$\frac{a}{b} < \frac{c}{d} \text{ and } b < 0 \quad \Leftrightarrow \quad ad > cb$$

Rational number class

- How about comparisons?

```
bool less_than( Rational const &p, Rational const &q );

bool less_than( Rational const &p, Rational const &q ) {
    if ( ((p.denom_ >= 0) && (q.denom_ >= 0))
           || ((p.denom_ < 0) && (q.denom_ < 0)) ) {
        return p.numer_*q.denom_ < p.denom_*q.numer_;
    } else {
        return p.numer_*q.denom_ > p.denom_*q.numer_;
    }
}
```

Rational number class

- How about its floating-point approximation?

```
double real( Rational const &p );

double real( Rational const &p ) {
    return (1.0*p.numer_)/p.denom_;
}
```

- Instead of tricking the compiler to convert an integer to a double, we can explicitly tell it to do so:

```
double real( Rational const &p ) {
    return static_cast<double>( p.numer_ )
            /static_cast<double>( p.denom_ );
}
```


Array class

- Recall that we previously had to create an array and store the capacity in a separate variable?
  - This is the ideal situation for a class

```
class Array {
    std::size_t capacity_;
    double array_[capacity_];
};
```

  - Problem: C++ does not allow classes, the size of which cannot be determined at compile time!

Array class

- In C++, arrays in classes must either be
  - Fixed in capacity
  - Use dynamically allocated arrays

```
class Array {
    std::size_t capacity_;
    double *array_;
};
```


Array class

- You could use this array as follows:

```
int main() {
    Array my_array{32, new double[32]{}};

    for ( std::size_t k{0}; k < my_array.capacity_; ++k ) {
        my_array.array_[k] = 1.5;
    }

    // Use 'my_array'

    delete[] my_array.array_;
    my_array.array_ = nullptr;
    my_array.capacity_ = 0;

    return 0;
}
```

Array class

- This initialization and clean-up, however, could be better performed by appropriate functions:

```
void array_init( Array &array, std::size_t const capacity,
                               double      const initial_value = 0.0 );

void array_init( Array &array, std::size_t const capacity,
                               double      const initial_value ) {
    if ( array.array_ != nullptr ) {
        delete[] array.array_;
        array.array_ = nullptr;
        array.capacity_ = 0;
    }

    array.capacity_ = capacity;
    array.array_ = new double[array.capacity_];

    for ( std::size_t k{0}; k < array.capacity_; ++k ) {
        array.array_[k] = initial_value;
    }
}
```

Array class

- This initialization and clean-up, however, could be better performed by appropriate functions:

```
void array_clean_up( Array &array );

void array_clean_up( Array &array ) {
    // In C++, it is completely acceptable to call 'delete[]'
    // on a null pointer
    delete[] array.array_;
    array.array_ = nullptr;
    array.capacity_ = 0;
}
```

Array class

- You could use this array class as follows:

```
int main() {
    Array my_array{};
    array_init( my_array, 32, 1.5 );

    // Use 'my_array'

    array_clean_up( my_array );

    return 0;
}
```

Array class

- You could author other functions:

```
void array_reset( Array &array, double const new_value = 0.0 );

void array_reset( Array &array, double const new_value ) {
    // Cannot reset if there is no array allocated
    if ( array.array_ == nullptr ) {
        return;
    }

    for ( std::size_t k{0}; k < array.capacity_; ++k ) {
        array.array_[k] = new_value;
    }
}
```

Array class

- You could author other functions:

```
double array_average( Array &array );

double array_average( Array &array ) {
    // Cannot calculate an average if there is no array allocated
    if ( array.array_ == nullptr ) {
        return 0.0;
    }

    double sum{ 0.0 };

    for ( std::size_t k{0}; k < array.capacity_; ++k ) {
        sum += array.array_[k];
    }

    return sum/array.capacity_;
}
```

Array class

- There are many other array functions you could author:
  - Checking if the array is sorted
  - Sorting the array
  - Searching the array linearly
  - Searching the array using a binary search
- This, however, will be left for later
  - Right now, our biggest problem is we are always required to check if the `array_` member variable is `nullptr`
  - When we fix this, we'll author a better array class

Pair class

- To this point, when a function is meant to act on a specific object, that object is passed by reference, usually as the first argument
  - This is whether or not the object is being changed or if it simply being investigated
  - Information generally is returned through a return value
- We have seen that functions return an instance of a type
  - If we wanted more information returned, additional arguments had to be passed by reference

```
void maxmin( Array const &array,
             std::size_t &imaximum,
             std::size_t &iminimum );
```

Pair class

- Instead, consider the following:

```
// Class declarations
class Pair;

// Function declarations
Pair maxmin( Array const &array );

// Class definitions
class Pair {
    public:
        std::size_t first_;
        std::size_t second_;
};
```

Pair class

- We can now use this:

```
// Function definitions
Pair maxmin( Array const &array ) {
    assert( array.capacity_ > 0 );

    std::size_t imax{ 0 };
    std::size_t imin{ 0 };

    for ( std::size_t k{1}; k < array.capacity_; ++k ) {
        if ( array.array_[k] > array.array_[imax] ) {
            imax = k;
        } else if ( array.array_[k] < array.array_[imin] ) {
            imin = k;
        }
    }

    return Pair{ imax, imin };
}
```


Pair class

- We can now use this:

```
int main() {
    Array data{};
    array_init( &data, 10 );    // Default value is 0

    for ( std::size_t k; k < data.capacity_; ++k ) {
        std::cout << "Enter a number: ";
        std::cin >> data.array_[k];
    }

    Pair result{ maxmin( data ) };

    std::cout << "The max is " << data.array_[result.first_]
              << std::endl;
    std::cout << "The min is " << data.array_[result.second_]
              << std::endl;
    return 0;
}
```


Summary

- Following this lesson, you now
  - Seen numerous examples of classes
  - Looked at functions that access and manipulate objects
  - Understand it is necessary to check what is being passed
  - Seen how to return multiple return values by simply constructing a new class

References

[1] https://en.wikipedia.org/wiki/C++_classes

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