Functions on objects

- 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

Functions on vectors

- Consider this `Vector_3d` class
  ```cpp
  class Vector_3d {
  public:
    // Member variables
    double x_;  
    double y_;  
    double z_;  
  }
  ```

- 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
The inner product and the norm

- For example, consider these functions:
  
  ```cpp
  double inner_product( Vector_3d const &u, Vector_3d const &v );
  double norm( Vector_3d const &v );
  
  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 &v ) {
    return std::sqrt( inner_product( u, u ) );
  }
  ```

Vector addition and scalar multiplication

- Here are the vector operations:
  
  ```cpp
  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

- Here are the projection and the cross product:
  
  ```cpp
  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.x_ };  
  }
  ```

Comparisons

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

```cpp
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

- 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

Rational number class

- The class itself is straightforward:

```cpp
class Rational {
public:
    int numer_;  
    int denom_;  
};
```
• 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 } );
  }

• 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 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_ };
  }
We can now implement functions that work on rational numbers:

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

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

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

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

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

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

```
\[
\frac{p_x}{p_d} < \frac{q_x}{q_d} \iff p_xq_d < p_dq_x
\]
```

```
\[
\frac{1}{-1} < \frac{1}{1} \iff 1 \cdot 1 < -1 \cdot -1
\]
```

```
\[
\frac{a}{b} < \frac{c}{d} \text{ and } b < 0 \iff ad > cb
\]
```
Rational number class

- How about its floating-point approximation?
  ```cpp
  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:
  ```cpp
  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:
    ```cpp
    class Array {
      std::size_t capacity_;  // Fixed in capacity
      double *array_;  // Use dynamically allocated arrays
    };
    ```
  - 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
    ```cpp
    class Array {
      std::size_t capacity_;  // Fixed in capacity
      double *array_;  // Use dynamically allocated arrays
    };
    ```

- You could use this array as follows:
  ```cpp
  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;
    }
    delete[] my_array.array_;  // Use 'my_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:

```cpp
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_;  // In C++, it is completely acceptable to call 'delete[]'
        array.array_ = nullptr;  // on a null pointer
        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;
    }
}
```

- You could use this array class as follows:

```cpp
int main() {
    Array my_array{};
    array_init( my_array, 32, 1.5 );
    // Use 'my_array'
    array_clean_up( my_array );
    return 0;
}
```

Array class

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

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

- You could author other functions:

```cpp
void array_reset( Array &array, double const new_value = 0.0 );
void array_reset( Array &array, double const new_value ) {
    if ( array.array_ == nullptr ) {
        return;
    }
    for ( std::size_t k{0}; k < array.capacity_; ++k ) {
        array.array_[k] = new_value;
    }
}
```
Functions on objects

**Array class**

- You could author other functions:

  ```cpp
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_;  
  }
```

- 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

  ```cpp
  void maxmin( Array const &array,  
    std::size_t &maximum,  
    std::size_t &minimum );
  ```

- Instead, consider the following:

  ```cpp
  // 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:

```cpp
// 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};
}
```

---

**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**

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.