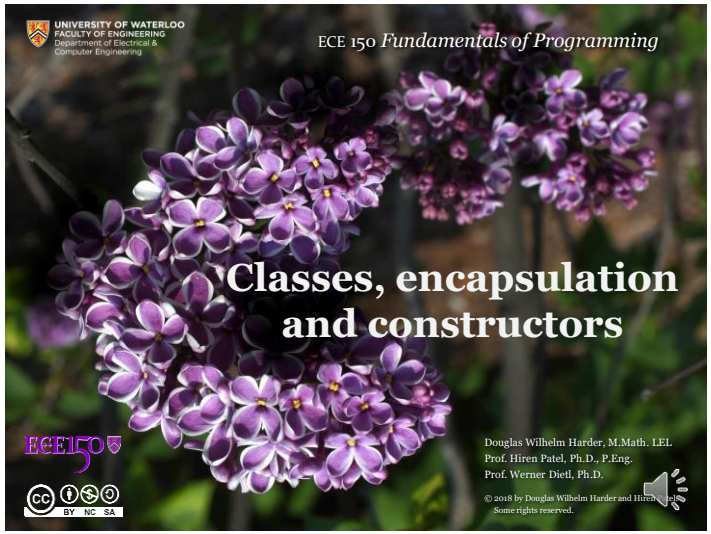

1

Outline

- In this lesson, we will:
  - Discuss constructors for rational numbers
  - Describe normalizing a rational number
  - Describe member functions to access and modify the numerator and denominator
  - Discuss authoring other functions and member functions on rational numbers
  - Describe how to do operator overloading as member functions
  - Discuss assignment, copying and the need of a destructor
  - Discuss static member functions

2

Review of encapsulation

- At this point, you now:
  - Have seen how to author a class with member variables
  - Are aware of the need for encapsulation
    - This requires member variables to be inaccessible
  - Are aware we must deal with:
    - The initialization of the objects
      - Through constructors
    - Allow users to access and manipulate objects through their lifetime
      - Through member functions
    - Clean up before the object goes out of scope or is deleted
      - Through destructors
- You have also seen various classes and seen:
  - How to use member functions
  - Perhaps noticed that initialization and clean-up seems to be more-or-less seamless: the compiler deals with this

3

Using member functions

- For example
  - You know nothing about how the `std::cout` object is designed
  - You know nothing about how a string is stored inside an instance of the `std::string` class
  - You don't know what is happening inside an instance of the `std::set` class, but it seems to have some powerful functionality
  - You don't know how the exception class stores the string that is passed to the constructor
- However, you don't need to know this,
  so long as you know what the member functions are,
  and how to access and manipulate them

4

Private member variables

- We will now start by modifying our rational number class
  - First we will no longer allow users to access the member variables

```
class Rational {
    public:

    private:
        int numer_;
        int denom_;
};
```

  - Up to now, we had a nebulous `public:` label at the top of the class
    - Anything appearing after this is accessible to any user
  - Anything after the `private:` label may only be accessed by:
    - Constructors, member functions, destructors and *friends*

Private member variables

- In theory, you can have as many such labels as you wish:

```
class Rational {
        // Anything before any label is private...
    public:
        // Some public member functions
    private:
        // Some private member variables
        // and member functions
    public:
        // More public stuff...
    private:
        // More private stuff...
    private:
        // Even more private stuff...  :-/
};
```


Private member variables

- Our approach:

```
class Rational {
    public:
        // Anything the user can access
        //  - This is information useful to everyone
        //    so it will come first

    private:
        // Anything that is only accessible in
        // constructors, member functions, destructors
        // and friends appears here
        //  - This is information useful only to authors
        //    of this class
};
```


Private member variables

- Private member variables cannot even be initialized by the user
  - By default, all member variables are always initialized to their default values (`0`, `0.0`, `'\0'`, `false`)

- For example,

```
int main() {
    Rational q{};
    return 0;
}
```

Private member variables

- Private member variables can also be instances of classes, in which case, their default value is whatever the default is for that class (if it has one):
  - The default for an instance of a `std::string` class is the empty string
- For example,

```
class Person {
    public:
    private:
        std::string preferred_name_;
        std::string surname_;
        int birth_year_;
        int birth_month_;
        int birth_day_;
};
```

9

Constructors

- Problem:
  - The default value is exactly what we don't want for our rational number class:
    - The default denominator is `0`
- If we want anything else than the default values, we must define a *constructor* that initializes an object

10

Constructors

- The constructor is not like other member functions:
  - A constructor has the same name as the class
  - It is never explicitly called by a user, its call is scheduled by the compiler
  - A constructor has no return value

```
class Rational {
    public:
        // Constructors
        Rational();
    private:
        int numer_;
        int denom_;
};
Rational::Rational():
numer_{0},
denom_{1} {
    // Empty constructor--all member variables
    // are simply initialized
}
```

11

Constructors

- Like all other functions, constructors must be declared and defined
  - The constructor declaration is in the class definition

```
Rational();
```

  - The constructor definition appears after the class definition

```
Rational::Rational():
numer_{0},
denom_{1} {
    // Empty constructor
    //  - all member variables are simply initialized
}
```

12

Constructors

- To indicate that this is not the definition of a function named `Rational`, but rather the definition of a constructor in the `Rational` class, we indicate this by placing `Rational::` before the constructor identifier

```
Rational::Rational():
numer_{0},
denom_{1} {
    // Empty constructor
    //  - all member variables are simply initialized
}
```

  - If you don't, the compiler will think you are trying to define a function called `Rational()` and will give you an error as you didn't specify a return type

Constructors

- Next, we have a colon and all member variables together with initial values
  - The order MUST be the same as the they are listed in the class definition

```
Rational::Rational():
numer_{0},
denom_{1} {
    // Empty constructor
    //  - all member variables are simply initialized
}
```

  - These are the member variables of the object being created
  - Each member variable will be initialized one at a time
  - Subsequent member variables can the initial values of previous member variables

Constructors

- Finally, there is the constructor body
  - This is a block of code that is executed after all member variables are initialized
  - There is nothing the constructor need do for rational numbers

```
Rational::Rational():
numer_{0},
denom_{1} {
    // Empty constructor
    //  - all member variables are initialized
    //  - the rational number is
    //          already in normal form
}
```


Member functions

- Let us write member functions that can return the numerator and the denominator

```
class Rational {
    public:
        // Constructors
        Rational();

        // Member functions
        int numer() const;
        int denom() const;
    private:
        int numer_;
        int denom_;
};
```

The `const` after the declaration says "This member function cannot change the values of any member variables."

Member functions

- We can define these member functions:

```
int Rational::numer() const {
    return numer_;
}

int Rational::denom() const {
    return denom_;
}
```

  - Member functions are declared in the class definition
  - Member functions are defined after the class definition
  - Any time you refer to any member variable in a member function, these refer to the member variables of the object that on which the member function was called

Member functions

- We can now use these member functions

```
int main() {
    Rational q{};

    std::cout << q.numer() << "/" << q.denom() << std::endl;
    return 0;
}
```

Output:
```
0/1
```

- In this `main()` function, `q` is a local variable
  - The memory allocated for `q` is cleaned up once that local variable goes out of scope
    - That is, when it goes out of scope, the memory on the stack may be reused

Member functions

- We can even implement the printing of rational numbers:

```
std::ostream &operator<<( std::ostream &out, Rational const &p ) {
    out << p.numer() << "/" << p.denom();
    return out;
}
```

- Using this, we have

```
int main() {
    Rational q{};
    std::cout << q << std::endl;
    return 0;
}
```

Output:
```
0/1
```

Member functions

- Could we improve the output?

```
std::ostream &operator<<( std::ostream &out, Rational const &p ) {
    out << p.numer();

    if ( p.denom() != 1 ) {
        out << "/" << p.denom();
    }

    return out;
}
```

Output:
```
0
```

- Now, if the denominator is `1`, the rational number is printed as an integer

Constructors

- Alternatively, constructors can have parameters
  - These parameters can have default values
  - As with functions, default values must be given in the declaration
  - The declaration of a constructor is in the class definition

```
class Rational {
    public:
        Rational();
        Rational( int new_numer, int new_denom = 1 );
    private:
        int numer_;
        int denom_;
};
```

Constructors

- Now there is more work to do...
  - Question: If the user gives a zero denominator, do we just hide the error and set `denom_` to 1?
  - Hiding errors is often a mistake
    - The domain for the denominator is any non-zero integer

```
Rational::Rational( int new_numer, int new_denom ):
numer_{ new_numer },
denom_{ new_denom } {
    if ( denom_ == 0 ) {
        throw std::domain_error{
            "The denominator must be non-zero"
        };
    }
}
```

Constructors

- Recall that we want rational numbers stored in a normal form:
  - The denominator is always positive
  - The numerator and denominator share no common divisors

```
Rational::Rational( int new_numer, int new_denom ):
numer_{ new_numer },
denom_{ new_denom } {
    if ( denom_ == 0 ) {
        throw std::domain_error{
            "The denominator must be non-zero"
        };
    }

    // Normalize the rational number...
}
```

Constructors

- We could just normalize the number in the constructor...
  - However, after any arithmetic operation with rational numbers, we will have to normalize the result
  - If we are doing this umpteen times, why not write a function to do this?

```
Rational::Rational( int new_numer, int new_denom ):
numer_{ new_numer },
denom_{ new_denom } {
    if ( denom_ == 0 ) {
        throw std::domain_error{
            "The denominator must be non-zero"
        };
    }

    // Normalize the rational number...
}
```

Private member functions

- Let's add a member function that normalizes a rational number
  - Users should not access this function, so we will declare it private
  - These are sometimes called *helper* functions
    - They are meant to be used by other member functions, not users
  - It may change the member variables, so it is not `const`

```
class Rational {
    public:
        Rational();
        Rational( int new_numer, int new_denom = 1 );

        int numer() const;
        int denom() const;
    private:
        int numer_;
        int denom_;
        void normalize();
};
```


Private member functions

- Here is an implementation:

```
void Rational::normalize() {
    // By the time this function is ever called,
    // the denominator should never be zero
    assert( denom_ != 0 );

    if ( denom_ < 0 ) {
        numer_ = -numer_;
        denom_ = -denom_;
    }

    int divisor{ gcd( numer_, denom_ ) };

    numer_ /= divisor;
    denom_ /= divisor;
}
```

For now, assume `gcd(…)` is implemented somewhere else

Constructors

- How do we use this normalize member function in the constructor?
  - If a member function is called inside a constructor, member function or destructor, it is assumed to be called on the same object that this constructor, member function or destructor was called on

```
Rational::Rational( int new_numer, int new_denom ):
numer_{ new_numer },
denom_{ new_denom } {
    if ( denom_ == 0 ) {
        throw std::domain_error{
            "The denominator must be non-zero"
        };
    }

    normalize();
}
```


Constructors

- We can now use this:

```
int main() {
    Rational p{ -10, -45 };

    std::cout << p << std::endl;

    return 0;
}
```

Output:
2/9

Constructors

- You can have as many constructors as you wish, but the compiler must be able to determine exactly which constructor you are calling

```
class Rational {
    public:
        Rational();
        Rational( int new_numer = 0, int new_denom = 1 );
        // Other public member functions
    private:
        // Private member variables and
        // private member functions
};
```


Member functions

- Next, let us write member functions that allow the user to change the numerator or denominator

```
class Rational {
    public:
        // Constructors
        Rational();
        Rational( int new_numer, int new_denom = 1 );

        // Member functions
        int numer() const;
        int denom() const;
        void numer( int new_numer );
        void denom( int new_denom );
    private:
        int numer_;
        int denom_;
        void normalize();
};
```


Member functions

- We can implement these functions:

```
void Rational::numer( int new_numer ) {
    numer_ = new_numer;
    normalize();
}

void Rational::denom( int new_denom ) {
    if ( new_denom == 0 ) {
        throw domain_error{ "The denominator cannot be set to 0" };
    }

    denom_ = new_denom;
    normalize();
}
```


Member functions

- Suppose we want to implement an absolute value function
  - Is it a member function, or just a function

```
int main() {
    Rational q{ -5, 17 };

    std::cout << abs( q ) << std::endl;
    std::cout << q.abs() << std::endl;

    return 0;
}
```

Member functions

- Each is implemented differently:

```
// Called using abs( q )
//  - This requires a function declaration
Rational abs( Rational const &q ) {
    return Rational{ std::abs( q.numer() ), q.denom() };
}

// Called using q.abs()
//  - This requires a member function declaration
//         in the class definition
Rational Rational::abs() const {
    return Rational{ std::abs( numer() ), denom() };
}
```

The preference in C++ is to use the second.
– Don't buck the trend...it's a waste of time

Operator overloading

- How about operator overloading?

```
// This requires a function declaration
Rational operator+( Rational const &p, Rational const &q ) {
    return Rational{ p.numer()*q.denom() + q.numer()*p.denom(),
                     p.denom()*q.denom() };
}

// This requires a member function declaration
//       in the class definition
//  - The left operand is the object on which this member
//    function is called, and the right is the argument
Rational Rational::operator+( Rational const &q ) const {
    return Rational{ numer()*q.denom() + q.numer()*denom(),
                     denom()*q.denom() };
}
```

The preference in C++ is to use the second.
– Don't buck the trend...it's a waste of time

Operator overloading

- Which of these should be used?

```
Rational Rational::operator+( Rational const &q ) const {
    return Rational{ numer()*q.denom() + q.numer()*denom(),
                     denom()*q.denom() };
}

Rational Rational::operator+( Rational const &q ) const {
    return Rational{ numer_*q.denom_ + q.numer_*denom_,
                     denom_*q.denom_ };
}
```

– The compiler will likely implement first as if it was authored in the second...

Private member functions

- Here is a better implementation of `normalize()`:

```
void Rational::normalize() {
    // By the time this function is ever called,
    // the denominator should never be zero
    assert( denom() != 0 );

    if ( denom() < 0 ) {
        numer_ = -numer();
        denom_ = -denom();
    }

    int divisor{ gcd( numer(), denom() ) };

    numer_ /= divisor;
    denom_ /= divisor;
}
```

```
if ( denom_ = 0 ) {
    // Do something...
}
```

Operator overloading

- How about operator overloading?

```
// The denominator of an integer is '1'
Rational Rational::operator+( int const n ) const {
    return Rational{ numer() + n*denom(),
                     denom() };
}

// Here, the first argument is an int, so we cannot define
// this as a member function, but when we return q + n,
// this calls the above function
Rational operator+( int const n, Rational const &q ) {
    return q + n;
}
```


Our class definition

```
class Rational;

Rational operator+( int const n, Rational const &q );

class Rational {
    public:
        // Constructors
        Rational();
        Rational( int new_numer, int new_denom = 1 );

        // Member functions
        int numer() const;
        int denom() const;
        Rational abs() const;
        Rational operator+( Rational const &q ) const;
        Rational operator+( int const n ) const;

        void numer( int new_numer );
        void denom( int new_denom );
    private:
        int numer_;
        int denom_;
        void normalize();
};
```


Operator overloading

- How about unary + and unary -?

```
// '+q' evaluates to the rational number 'q'
//   - as in p + +q;
Rational Rational::operator+() const {
    return Rational{ numer(), denom() };
}

// '-q' evaluates to the additive inverse of 'q'
//   - as in p + -q;
Rational Rational::operator-() const {
    return Rational{ -numer(), denom() };
}
```


Operator overloading

- Now we can implement subtraction:

```
Rational Rational::operator-( Rational const &q ) const {
    return operator+( -q );
}

Rational Rational::operator-( int const n ) const {
    return operator+( -n );
}

Rational operator-( int const n, Rational const &q ) const {
    return (-q) + n;
}
```

Comparisons

- Also, comparisons become much easier:

```
// This will be called if you every compare p == q for
// two instances of the Rational class
bool Rational::operator==( Rational const &q ) const {
    return (numer() == q.numer()) && (denom() == q.denom());
}

// This will be called if you every compare p == n where
// p is an instance of the Rational class and n is an int
bool Rational::operator==( int const n ) const {
    return (denom() == 1) && (numer() == n);
}

// This will be called if you every compare n == q where
// q is an instance of the Rational class and n is an int
Rational operator==( int const n, Rational const &q ) {
    return q == n;
}
```


Comparisons

- Also, comparisons become much easier:

```
// This will be called if you every compare p < q for
// two instances of the Rational class
bool Rational::operator<( Rational const &q ) const {
    return numer()*q.denom() < q.numer()*denom();
}

// This will be called if you every compare p < n where
// p is an instance of the Rational class and n is an int
bool Rational::operator<( int const n ) const {
    return numer() < n*denom();
}

// This will be called if you every compare n < q where
// q is an instance of the Rational class and n is an int
Rational operator<( int const n, Rational const &q ) const {
    return Rational{ n, 1 } < q;
}
```


Assignment

- What happens with assignment?

```
int main() {
    Rational p{25, -15};
    Rational q{3, 5};
    Rational r{2, 4};

    std::cout << (p + q) << std::endl;
    std::cout << (p + r) << std::endl;
    q = r;
    std::cout << (p + q) << std::endl;

    return 0;
}
```

Output:

```
-16/15
-7/6
-7/6
```

  - By default, all member variables are just copied over
    - This is true even if they are private, because the user still does not have access to them

Initializing a new rational with another

- What happens with initializations?

```
int main() {
    Rational p{25, -15};
    Rational q{3, 5};
    Rational r{q};
    std::cout << (p + q) << std::endl;
    std::cout << (p + r) << std::endl;
    q = p;
    std::cout << (p + q) << std::endl;
    std::cout << (p + r) << std::endl;
    return 0;
}
```

Output:

```
-16/15
-16/15
-10/3
-16/15
```

  - By default, all member variables are just copied over
    - This is true even if they are private, because the user still does not have access to them

Destructor?

- Does a rational number class need a destructor?
  - No additional memory is required,
    there is nothing to clean up, so no

A gcd function?

- Recall that previously, we needed a `gcd` function in normalize member function
  - If we were running C++17, we'd have such a function...
  - Such a function is a helper function,
    but it is a helper function independent of any instance
    - That is, it is a function taking two `int`, and returning an `int`

A gcd function?

```
// This implementation assumes that 'n' is the
// denominator and thus greater than zero
int Rational::gcd( int m, int n ) {
    assert( n > 0 );

    if ( m == 0 ) {
        return n;
    } else {
        if ( m < 0 ) {
            m = -m;
        }

        while ( n != 0 ) {
            int rem{ m % n };
            m = n;
            n = rem;
        }

        return m;
    }
}
```

A gcd function?

- Because it is independent of any instance of a class,
  we can declare it to be a function associated with the class
  - This is done by declaring the function `static` in the class definition

```
class Rational {
    public:
        // All the public stuff...
    private:
        int numer_;
        int denom_;
        void normalize();
        static int gcd( int m, int n );
};
```

  - Static member functions may not access
    any non-static member variables or
    call any non-static member functions

Summary

- Following this lesson, you now
  - Understand how to make member variables private
  - Know how to define constructors
  - Understand how to author member functions
  - Understand how to do operator overloading with member functions
  - Understand the workings of assignment and initialization
  - Are aware that a destructor isn't always necessary
  - Are aware of static member functions

References

[1] https://en.wikipedia.org/wiki/C++_classes
[2] https://en.wikipedia.org/wiki/Exception_handling
[3] https://www.cplusplus.com/reference/stdexcept/

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