UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Issues with member variables

Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

© 2018 by Douglas Wilhelm Harder and Hiren Patel.
Some rights reserved.

1

Outline

- In this lesson, we will:
  - Observe some issues with our classes
  - See that any *fixes* can be corrupted by the user
  - Introduce the idea of encapsulation
    - Protecting the data in an object from the user
  - See that the critical times in an object's life are:
    - How it is initialized when it is created
    - How the user accesses and manipulates the object during its lifetime
    - How it is cleaned up when it is destroyed

2

Issues with rational numbers

- We looked at a rational number class:
  - What happens here?

```
Rational p{};
```

  - The ratio 0/0 is not a rational number
    - But we cannot prevent a user from writing using our class like this
  - Equivalently, a user may incorrectly or accidentally set the denominator to zero:

```
p.denom_ = 0;
```

  - Under these circumstances,
    none of the arithmetic operations are properly defined

3

Ensuring a non-zero denominator

- One solution is to prefix every function and operator with a check that any denominators are not zero:

```
Rational operator+( Rational const &p,
                    Rational const &q ) {
    assert( (p.denom_ != 0.0) && (q.denom_ != 0.0) );

    return Rational{
        p.numer_*q.denom_ + q.numer_*p.denom_,
        p.denom_*q.denom_
    };
}
```

  - This is going to be expensive:
    the cost of these checks will slow any system down
  - Problem: As long as the user has access to the member variables,
    we are forced to make such checks

4

Normal forms

- Another issue:
  - These are all the same rational number:
    ```
    Rational p1{-1, 2};
    Rational p2{100, -200};
    Rational p3{-47, 94};
    Rational p4{99, -198};
    ```
  - Wouldn't it be great if all of these were stored the same way?
  - We could require:
    - The denominator must be positive
    - The greatest common divisor of the numerator and denominator should be one
  - These two rules make the representation unique
  - This is called a *normal form*

Normal forms

- For example, we could do the following:

```
Rational operator+( Rational const &p,
                    Rational const &q ) {
    assert( (p.denom_ != 0.0) && (q.denom_ != 0.0) );

    int numer{ p.numer_*q.denom_ + q.numer_*p.denom_ };
    int denom{ q.denom_*p.denom_ };
    int divisor{ gcd( numer, denom ) };

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

    return Rational{ numer/divisor, denom/divisor };
}
```

We also require a function `int gcd( int m, int n );`

Normal forms

- We cannot, however, assume this, as the user may :

```
bool operator==( Rational const &p,
                 Rational const &q ) {
    assert( (p.denom_ != 0.0) && (q.denom_ != 0.0) );

    return (p.numer_ == q.numer_)
        && (p.denom_ == q.denom_);
}
```

```
bool operator==( Rational const &p,
                 Rational const &q ) {
    assert( (p.denom_ != 0.0) && (q.denom_ != 0.0) );

    return (p.numer_*q.denom_ == q.numer_*p.denom_);
}
```

Normalizing a rational number

- Note, if we are always normalizing, we could author the following function:

```
void normalize( Rational &q ) {
    assert( q.denom_ != 0 );

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

    int divisor{ gcd( q.numer_, q.denom_ ) };

    q.numer_ /= divsor;
    q.denom_ /= divisor;
}
```

Normalizing a rational number

- We could write these two functions:

```
Rational operator+( Rational &p,
                    Rational &q ) {
    normalize( p );
    normalize( q );

    Rational result{
        p.numer_*q.denom_ + q.numer_*p.denom_,
        q.denom_*p.denom_
    };

    normalize( result );

    return result;
}
```

Problem:
We cannot declare p or q to be constant…

Normalizing a rational number

- We could write these two functions:

```
Rational operator==( Rational &p,
                     Rational &q ) {
    normalize( p );
    normalize( q );

    return (p.numer_ == q.numer_)
        && (p.denom_ == q.denom_);
}
```

Again,
we cannot declare p or q to be constant…

Normalizing a rational number

- Now, more work is done work checking and normalizing the numerator and denominator than the actual operation
  - We cannot ever assume the rational arguments are normalized, because the user may change either member variable
- If you are the only user of this class, no problem
  - In industry, however, your class will be used by:
    - Other employees or groups within the company
    - Clients and customers
    - The public in general
    - Whoever replaces you in your position when you leave or are promoted

Issues with our Array class

- Next, let us discuss the Array class:

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

- Here are all the problems:
  - The user may not even initialize an instance
    - A wild pointer `Array data;`
  - The user may initialize it with default values
    - A null pointer `Array data{};`
  - The user may accidentally assign array a null pointer
    - A memory leak `data.array_ = nullptr;`
  - The user may forget to call the clean-up function
    - A memory leak
  - The user may assign the wrong capacity

Issues with our Array class

- All these problems exist because:
  1. The user is not required to follow any sequence of statements to initialize the instance of the class
  2. At any time, the user can change any member variable to whatever they want, even if that change is accidental
  3. An object assigned to a local variable may go out of scope or the user may call delete on an object without properly cleaning any additional memory.

- In C libraries, which offer no additional protections, many commercial libraries simply assumed you were not making any mistakes
  - The assumption is the programmer is a veteran
  - This made finding bugs much more difficult, especially for new programmers

Solution

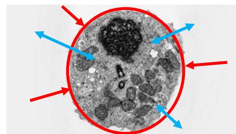
- To fix these problems, we must deal with the lifetime of the object, from creation, while it is in existence, to its destruction

  Creation → Existence → Destruction
- This includes:
  - Correctly dealing with the initialization of the object
  - Allowing the user to access and manipulate the object without ever having the opportunity to accidentally: corrupt data or leave the object in an inconsistent state
    - Not allowing the user to access member variable directly, ever
  - Correctly dealing with the clean-up or destruction of the object

- In order to introduce the terminology:
  - *Constructors* will deal with object initialization
  - *Member functions* will deal with the accessing and manipulation
  - *Destructors* will deal with the clean-up of objects

Encapsulation

- This process of protecting data from the user is called *encapsulation*
  - A biological cell has a cell membrane that *encapsulates* the cell
  - All information, including mitochondria, DNA and other proteins are protected inside the cell
    - DNA is actually protected inside the nucleus: a cell within the cell
  - The membrane prevents anything other than what is required to enter the cell
    - The membrane blocks ions and larger molecules
    - There are *transport proteins* on the membrane surface that facilitate the transfer of specific molecules like glucose



https://www.bbc.co.uk/bitesize/guides/zg9mk2p/

Encapsulation is not security

- One word of warning:
  - Encapsulation in C++ is not security
  - Even if you use all the tools we will introduce, I can, with one line of code, break that encapsulation
    - Such breaking of encapsulation requires a very deliberate statement in C++ that is normally never used

Summary

- Following this lesson, you now
  - Are aware of the issues with users being able to access and manipulate member variables
  - Understand to fix this problem,
    we cannot allow the user to access member variables
  - Know there are three aspects of an objects lifetime that we must secure for the user:
    - The object's creation
    - The accessing and manipulation of objects by the user
    - The object's destruction
  - Are aware that this will be performed encapsulation through
    - Constructors
    - Member functions
    - Destructors

17

References

[1] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

18

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.

19

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.

20