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

ECE 150 *Fundamentals of Programming*

Exception classes and polymorphism

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

1

Outline

- In this lesson, we will:
  - Describe inheritance in the exception classes
  - Derive our own exception class from one of these classes
  - Observe that our exception class is still treated as if it is one of its base class, and so on
  - Describe the features of polymorophism

2

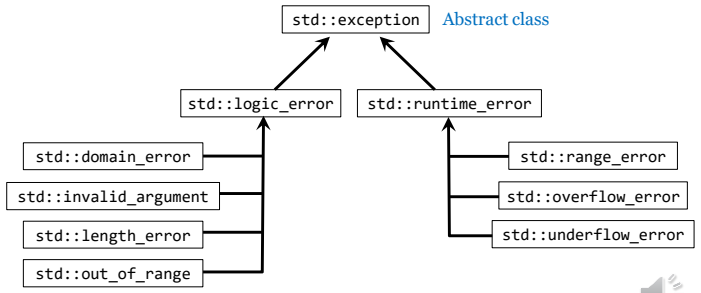
A recap so far

- To this point, we have seen how we can extend classes with inheritance
  - We can accept or modify existing functionality
  - We can introduce new functionality and member variables
- We looked at how this could apply:
  - To linked lists
  - To classes describing a graphical user interface
- Before we look at another higher-level example with respect to graphical user interfaces, we will observe concrete examples of inheritance in the `std::exception` class and its derived classes and note the application of polymorphism

3

Exceptions

- We have already discussed exceptions
  - We have not discussed their inheritance



4

Why two branches?

- A logic error is an exceptional case that should have been caught by the programmer:
  - The programmer should be checking arguments, for example, to ensure that they fall within the acceptable bounds of a function
  - It is assumed that if a logic error is thrown, there was an issue with the source code
- A runtime error is an exceptional case that could only be determined at runtime
  - Perhaps the system was not correctly designed to handle the full field of possible inputs
  - Perhaps a value exceeds the data type used to store a result

Deriving from an exception class

- The default structure for all but the base `std::exception` class is essentially a string together with a member function `what()` that returns that string converted to a C-style string:

```
class exception_name : public base_exception {
    public:
        exception_name( char const        *new_what_arg );
        exception_name( std::string const &new_what_arg );
        virtual char const *what() const noexcept;

    protected:
        std::string what_arg_;
};
```

A derived exception class

- You can derive from such an exception:

```
class int_domain_error : public std::domain_error {
    public:
        int_domain_error( char const        *new_what_arg, int new_value );
        int_domain_error( std::string const &new_what_arg, int new_value );
        virtual char const *what() const noexcept;
        virtual int value() const noexcept;

    protected:
        int value_;
};
```

A derived exception class

- Implementing the constructors calls the base class constructor

```
int_domain_error::int_domain_error( char const *new_what_arg,
                                    int         new_value ):
std::domain_error{ new_what_arg },
value_{ new_value } {
    // Empty constructor
}

int_domain_error::int_domain_error( std::string const &new_what_arg,
                                    int                new_value ):
std::domain_error{ new_what_arg },
value_{ new_value } {
    // Empty constructor
}
```

A derived exception class

- We will also override `what()` and implement the `value()` member function

```
char const *int_domain_error::what() const noexcept {
    std::clog << " Logging: " << std::domain_error::what()
              << ": " << value() << std::endl;

    return std::domain_error::what();
}

int int_domain_error::value() const noexcept {
    return value_;
}
```

Using our derived class

- Let's look at an implementation:

```
int binomial( int n, int k ) {
    if ( n < 0 ) {
        throw int_domain_error{
            "The first argument 'n' must be zero or positive", n };
    } else if ( (k < 0) || (k > n) ) {
        throw int_domain_error{
            "The second argument 'k' must be between 0 and 'n' (= "
              + std::to_string( n ) + ")", k };
    } else {
        if ( (k == 0) || (k == n) ) {
            return 1;
        } else {
            return binomial( n, k - 1 ) + binomial( n - 1, k - 1 );
        }
    }
}
```

Catching an `int_domain_error`

- Suppose we run this program:

```
int main() {
    try {
        binomial( -5, 2 );
    } catch ( int_domain_error &e ) {
        std::cout << "\"" << e.what() << "\"" << std::endl;
    }

    return 0;
}
```

Output:

```
>>> Logging: The first argument 'n' must be zero or positive: -5
"The first argument 'n' must be zero or positive"
```

Catching a `std::domain_error`

- Suppose we run this program:

```
int main() {
    try {
        binomial( 5, -2 );
    } catch ( std::domain_error &e ) {
        std::cout << "\"" << e.what() << "\"" << std::endl;
    }

    return 0;
}
```

Output:

```
>>> Logging: The second argument 'k' must be between 0 and 'n' (= 5): -2
"The second argument 'k' must be between 0 and 'n' (= 5)"
```

Catching a `std::domain_error`

- Note that we cannot call additional features of the derived class:

```
int main() {
    try {
        binomial( 5, -2 );
    } catch ( std::domain_error &e ) {
        std::cout << "\"" << e.what() << "\"" << std::endl;
        std::cout << "\"" << e.value() << "\"" << std::endl;
    }

    return 0;
}
```


Catching a `std::logic_error`

- Suppose we run this program:

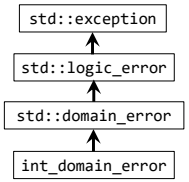
```
int main() {
    try {
        binomial( 5, 13 );
    } catch ( std::logic_error &e ) {
        std::cout << "\"" << e.what() << "\"" << std::endl;
    }

    return 0;
}
```

Output:

```
>>> Logging: The second argument 'k' must be between 0 and 'n' (= 5): 13
"The second argument 'k' must be between 0 and 'n' (= 5)"
```


Catching a `std::exception`

- Even though we cannot create an instance to `std::exception`, we can catch an exception for class derived from that class




```
int main() {
    try {
        binomial( 99, 101 );
    } catch ( std::exception &e ) {
        std::cout << "\"" << e.what() << "\"" << std::endl;
    }

    return 0;
}
```

Output:

```
>>> Logging: The second argument 'k' must be between 0 and 'n' (= 99): 101
"The second argument 'k' must be between 0 and 'n' (= 99)"
```


Example classes

```
class A {
  public:
    virtual void a() const;
};

class B : public A {
  public:
    virtual void b() const;
};

class C : public B {
  public:
    virtual void c() const;
    virtual void a() const override;
};

class D : public C {
  public:
    virtual void d() const;
    virtual void b() const override;
};
```

```
void A::a() const {
  std::cout << "Calling A::a()" << std::endl;
}

void B::b() const {
  std::cout << "Calling B::b()" << std::endl;
}

void C::c() const {
  std::cout << "Calling C::c()" << std::endl;
}

void C::a() const {
  std::cout << "Calling C::a()" << std::endl;
}

void D::d() const {
  std::cout << "Calling D::d()" << std::endl;
}

void D::b() const {
  std::cout << "Calling D::b()" << std::endl;
}
```

Example classes

```
int main() {
    D data{};
    data.a();                Calling C::a()
    data.b();                Calling D::b()
    data.c();                Calling C::c()
    data.d();                Calling D::d()

    C &c_ref{ data };
    c_ref.a();               Calling C::a()
    c_ref.b();               Calling D::b()
    c_ref.c();               Calling C::c()

     B &b_ref{ data };
     b_ref.a();              Calling C::a()
     b_ref.b();              Calling D::b()

     A &a_ref{ data };
     a_ref.a();              Calling C::a()

     return 0;
}
```


Example classes

- Additionally, we can create dynamically allocated instances of these classes, and yet have the most appropriate member function called

```
int main() {
    A *array[4]{};

    array[0] = new A{};
    array[1] = new B{};
    array[2] = new C{};
    array[3] = new D{};

    for ( int k{0}; k < 4; ++k ) {
        array[k]->a();
        delete array[k];
    }

    return 0;
}
```

Output:

```
Calling A::a()
Calling A::a()
Calling C::a()
Calling C::a()
```


Polymorphism

- The features that
  - An instance of a derived class can be assigned to a reference variable of a base class
  - The address of an instance of a derived class can be assigned to a pointer to a base class

  together with the fact that if you call a member function on such an assigned instance that the most appropriate member function is called—even if that member function is overridden in a class derived from the base class in question—are features of polymorphism

Polymorphism

- We saw this with exceptions, where our derived exception could never-the-less be caught by any of the base classes
- Also, when the `what()` member function was called on reference variables of the base classes, the version we overrode in our class was still the one that was called

Summary

- Following this lesson, you now
  - Seen how inheritance is used in the standard exception classes
  - Know that you can extend these exception classes
  - Understand how polymorphism works with classes

References

[1] https://en.wikipedia.org/wiki/Exception_handling
[2] https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

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.