14.x — Chapter 14 comprehensive quiz

BY ALEX ON FEBRUARY 11TH, 2017 | LAST MODIFIED BY ALEX ON MARCH 2ND, 2017

Chapter review

Exception handling provides a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code. This allows more freedom to handle errors when and how ever is most useful for a given situation, alleviating many (if not all) of the messiness that return codes cause.

A **throw** statement is used to raise an exception. **Try blocks** look for exceptions thrown by the code written or called within them. These exceptions get routed to **catch blocks**, which catch exceptions of particular types (if they match) and handle them. By default, an exception that is caught is considered handled.

Exceptions are handled immediately. If an exception is raised, control jumps to the nearest enclosing try block, looking for catch handlers that can handle the exception. If no try block is found or not catch blocks matches, the stack will be unwound until a handler is found. If no handler is found before the entire stack is unwound, the program will terminate with an unhandled exception error.

Exceptions of any data type can be thrown, including classes.

Catch blocks can be configured to catch exceptions of a particular data type, or a catch-all handler can be set up by using the ellipses (…). A catch block catching a base class reference will also catch exceptions of a derived class. All of the exceptions thrown by the standard library are derived from the std::exception class (which lives in the exception header), so catching a std::exception by reference will catch all standard library exceptions. The what() member function can be used to determine what kind of std::exception was thrown.

Inside a catch block, a new exception may be thrown. Because this new exception is thrown outside of the try block associated with that catch block, it won't be caught by the catch block it's thrown within. Exceptions may be rethrown from a catch block by using the keyword throw by itself. Do not rethrow an exception using the exception variable that was caught, otherwise object slicing may result.

Function try blocks give you a way to catch any exception that occurs within a function or an associated member initialization list. These are typically only used with derived class constructors.

You should never throw an exception from a destructor.

Finally, exception handling does have a cost. In most cases, code using exceptions will run slightly slower, and the cost of handling an exception is very high. You should only use exceptions to handle exceptional circumstances, not for normal error handling cases (e.g. invalid input).

Chapter quiz

1) Write a Fraction class that has a constructor that takes a numerator and a denominator. If the user passes in a denominator of 0, throw an exception of type std::runtime_error (included in the stdexcept header). In your main program, ask the user to enter two integers. If the Fraction is valid, print the fraction. If the Fraction is invalid, catch a std::exception, and tell the user that they entered an invalid fraction.

Here's what one run of the program should output:

```
Enter the numerator: 5
Enter the denominator: 0
Your fraction has an invalid denominator.
```

Hide Solution

```cpp
#include <iostream>
#include <stdexcept> // for std::runtime_error

class Fraction
{
private:
    int m_numerator = 0;
    int m_denominator = 1;

public:
    Fraction(int numerator = 0, int denominator = 1) :
        m_numerator(numerator), m_denominator(denominator)
    {
        if (m_denominator == 0)
            throw std::runtime_error("Invalid denominator");
    }

    friend std::ostream& operator<<(std::ostream& out, const Fraction &f1);

};

std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
    out << f1.m_numerator << "/" << f1.m_denominator;
    return out;
}

int main()
{
    std::cout << "Enter the numerator: ";
    int numerator;
    std::cin >> numerator;

    std::cout << "Enter the denominator: ";
    int denominator;
    std::cin >> denominator;

    try
    {
        Fraction f(numerator, denominator);
        std::cout << "Your fraction is: " << f << '\n';
    }
    catch (std::exception&)
    {
        std::cout << "Your fraction has an invalid denominator.\n";
    }

    return 0;
}
```