```
/**
 * ******* Lasalle College Vancouver *******.
 *
 * Object Oriented Programming in C++ II
 * Week 9 - Error Handling. Exception
 * @author
 * Ivaldo Tributino de Sousa <ISousa@lasallecollegevancouver.com>
 */
#pragma once

// Input/output library
#include <iostream>
#include <sstream>
#include <fstream>
using std :: cout;
using std :: endl;
using std :: cerr;

// Containers library
#include<vector>
#include<map>
using std :: vector;
using std :: map;


// Strings library
#include <string>
using std :: string;
using std :: to_string;

// Numerics library
#include <cmath>

// Utilities library
#include <optional>
#include <functional>
```

```
// Dynamic memory management
#include <memory>
using std :: unique_ptr;
using std :: shared_ptr;
using std :: make_unique;
using std :: make_shared;
// Error handling
# include <stdexcept>

// C compatibility headers
#include <cassert>
```

**test.h**

```
#include "libraries.h"

class Exc_Message : public std::exception{
private:
    std::string message_;
public:
    Exc_Message(const std::string& message): message_(message){};
    const char* what() const noexcept override {
        return message_.c_str();
    }
};

void readNullPointer()
{
    int* p = nullptr;
    try
    {
        if (!p){
            throw Exc_Message("Read from nullptr");
        }
        cout << *p << endl;
    }
    catch (const std::exception& e)
    {
        cerr << e.what() << endl;
    }
}

void writeNullPointer()
{
    int* p = nullptr;
    try
    {
        if (!p){
```

```
            throw Exc_Message("Write to nullptr");
        }
        *p = 42;
    }
    catch (const std::exception& e)
    {
        cerr << e.what() << endl;
    }
}

void deletedWeakPtr()
{

    std::shared_ptr<int> p1(new int(42));
    std::weak_ptr<int> wp(p1);
    p1.reset();
    try {
        if(wp.expired()){
            throw Exc_Message("Refers to an already deleted object");
            std::shared_ptr<int> p2(wp);
        }

    } catch(const std::exception& e) {
        cerr << e.what() << endl;
    }
}

void badFunctionCall()
{

    std::function<int()> f = nullptr;
    try {
        if(!f){
            throw Exc_Message("The function wrapper has no target.");
        }
```

```
    } catch(const std::exception& e) {
        cerr << e.what() << endl;
    }
}

map<string, std::function<void()>> test
{
    { "RNP", readNullPointer },
    { "WNP", writeNullPointer },
    { "DWP", deletedWeakPtr},
    { "BFC", badFunctionCall}
};
```

**polygon.h**

```
#pragma once
#include "libraries.h"

class Polygon{

  private: // Private members:

    // Data Members (underscore indicates a private member variable)
    unsigned int numberSides_;

  public:  // Public members:
    /**
      * Creates a triangle with one side measuring 1.
      */
    Polygon(); // Custom default constructor

    /**
      * Create a polygon using the following parameters:
      * @param numberSides.
      * @param  length.
      */
    Polygon(unsigned int numberSides); // Custom Constructor
    /**
      * Copy constructor: creates a new Polygon from another.
      * @param obj polygon to be copied.
      */
    Polygon(const Polygon & obj); // Custom Copy constructor

    double operator()(double length) const; // Function operator

    bool operator<(const Polygon& obj) const; // operator < overloading

    bool operator>(const Polygon& obj) const; // operator > overloading
    /**
      * Assignment operator for setting two Polygon equal to one another.
```

```
      * @param obj Polygon to copy into the current Polygon.
      * @return The current image for assignment chaining.
      */
    Polygon & operator=(const Polygon & obj);  // Custom assignment operator;

    /**
      * Destructor: frees all memory associated with a given Polygon object.
      * Invoked by the system.
      */
    ~Polygon(); // Destructor

     /**
      * Functions get name and get area
      */
    string shapeName();

    /**
      * Gets and sets
      */



    void setNumberSides(unsigned int numberSides);

    unsigned int getNumberSides() const;

};
```

**polygon.cpp**

```
#include "Polygon.h"
#define PI 3.14159265

Polygon :: Polygon() : numberSides_(3)
{
  cout << "Default Constructor Invoked"  << endl;
}

Polygon :: Polygon(unsigned int numberSides) : numberSides_(numberSides)
{
  assert(("Polygon is a geometrical figure with three or more sides.",
numberSides_ >= 3));

  // numberSides_ = (numberSides>2)? numberSides : 3;
  // if ((numberSides < 3) || (numberSides > INT_MAX)){
  //     throw std::runtime_error("Polygon is a geometrical figure with three
or more sides.");
  // }
  cout << "Constructor Invoked"  << endl;

}

double Polygon :: operator()(double length) const{
  double perimeter = numberSides_*length;
  double apothem = (length)/(2*tan(PI/numberSides_));
  return perimeter*apothem/2;

}

bool Polygon :: operator<(const Polygon& obj) const {
  return this->numberSides_ < obj.numberSides_;
}

bool Polygon :: operator>(const Polygon& obj) const{
```

```
  return this->numberSides_ > obj.numberSides_;
}

Polygon :: Polygon(const Polygon & obj){
  numberSides_ = obj.numberSides_;
  cout << "Copy Constructor Invoked"  << endl;
}

Polygon & Polygon :: operator=(const Polygon & obj){
  numberSides_ = obj.numberSides_;
  cout << "Assignment operator invoked"  << endl;
  return *this;
}

Polygon :: ~Polygon() {
  cout << "Polygon destroyed" << endl;
}

string Polygon::shapeName() {

  string arrayName[6] = {"triangle" , "square", "pentagon",
  "hexagon", "heptagon", "octagon"};

  string name = (numberSides_<9)? arrayName[numberSides_-3]:
to_string(numberSides_)+"_polygon";

  return name;
}

void Polygon :: setNumberSides(unsigned int numberSides){

  if (numberSides>2){
    numberSides_ = numberSides;
  }
  else{
    cout << "Please, only set values above 2." << endl;
```

```
  };

}

unsigned int Polygon ::getNumberSides() const {
  return numberSides_;
}
```

**main.cpp**

```
double division(int a, int b, char c) {
    const char& model = c;
    if( b == 0 && model == 'a') {
        throw "Division by zero condition!";
    }
    else if( b == 0 && model == 'b'){
        throw INT_MAX;
    }
    return (a/(double)b);
};

struct S { // The type has to be polymorphic
    virtual void f();
};

int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

std::optional<double> divide(int a, int b) {
  if (b != 0) return a/(double)b;
  return {}; // nullopt
}

int main(){

    //************************************************
    //   ---- Throwing and Catching Exceptions -----
    //************************************************
    cout << "---- Throwing and Catching Exceptions -----" << '\n';
```

```
    try {
        cout << division(4,3, 'b') << endl;
        cout << division(4,0, 'a') << endl;
    }
    catch (char const* msg)
    {
        cerr << msg << '\n'; // cerr is the standard error stream which is
used to output the errors.
    }
    catch (const int& value)
    {
        cerr << value << '\n';
    }

    //************************************************
    //         ---- std::exception -----
    //************************************************
    cout << "---- std::exception -----" << '\n';
    S* p = nullptr;
    try {
        // An exception of this type is thrown when a typeid operator is
applied
        //to a dereferenced null pointer value of a polymorphic type.
        std::cout << typeid(*p).name() << '\n';
    }
    catch (const std::bad_alloc& e)
    {
        std::cerr << e.what() << '\n';
    }
    catch(const std::bad_typeid& e) {
        std::cerr << e.what() << '\n';
    }

    //************************************************
    //         ---- std::exception::what -----
    //************************************************
```

```
cout << "---- std::exception::what -----" << '\n';

map<int, int> m = {{0, 1},{1, 1},{2, 1},{3, 2},{4, 3},{5, 5}};

int key = 8;
try {
    m.at(key);
}
catch(const std::out_of_range& e) {
  cerr << e.what() << '\n';
  m.insert(std::pair(key,fib(key))) ;
}

for(auto & [k,v] : m){
    cout << '{' << k << ':' << v << '}' << '\n';
}

// ************************************************
//       ----  Custom exception class -----
// ************************************************
cout << "---- Custom exception class -----" << '\n';

try {
    if (m.find(7) == m.end()){
        throw Exc_Message("This key is currently unavailable.");
    }

}
catch(const std::exception &e) {
  cerr << e.what() << '\n';
}

//************************************************
//         ---- More exceptions -----
//************************************************
cout << "---- More exceptions -----" << '\n';
```

```
test["RNP"](); // readNullPointer();

test["WNP"](); // writeNullPointer();

test["DWP"](); // deletedWeakPtr();

test["BFC"](); // badFunctionCall();

// ************************************************
//         ---- ios::exceptions -----
// ************************************************
cout << "---- ios::exceptions -----" << '\n';

std::ifstream myfile ("test.txt");
if(myfile.is_open()){
    while (!myfile.eof()) myfile.get();
    myfile.close();
}
else if(myfile.fail())
{
    cout << "Error: Ifstream failbit or badbit is set." << endl;
}

std::ifstream file;
file.exceptions( std::ifstream::failbit | std::ifstream::badbit );
try {
    file.open ("test.txt");
    while (!file.eof()) file.get();
    file.close();
}
catch(std::exception const& e){

    std::cerr << "Show me the error:" << e.what() << endl;
}
catch (std::ifstream::failure const& e) {
```

```
        std::cerr << e.what() << endl;
    }
    catch(...) {
        cout << "Please show me something" << endl;
    }

    cout << "Please show me something" << endl;

    //************************************************
    //         ---- std::optional -----
    //************************************************
    cout << "---- std::optional -----" << '\n';

    cout << (divide(9,4)).value() << '\n';
    cout << *divide(9,4) << '\n';
    // cout << divide(9,0).value() << '\n';
    cout << *divide(9,0) << '\n';    // If there's no value the behaviour is
undefined!
    cout << (divide(9,0)).value_or(42) << '\n';



    // ************************************************
    //      ---- Constructor Exception -----
    // ************************************************
    cout << "---- Constructor Exception -----" << '\n';
    {
        Polygon badPoly(-1);
        cout << badPoly.getNumberSides() << '\n';
    }

    {
        try{
            Polygon badPoly(-1);
        }
        catch(const std::exception& e) {
            cerr << e.what() << '\n';
```

```
        }
    }

    // ************************************************
    //               ---- Assert -----
    // ************************************************
    cout << "---- Assert -----" << '\n';
    {
        Polygon pentagon(5);
    }
    {
        Polygon badPoly(2);
    }


    return 0;

}
```