# Lecture review

## Exception Handling

*Exception handling* is a mechanism to handle runtime errors such as division by zero, array out of bounds, etc. It is a way to transfer control from one part of a program to another.
In C++, exceptions are thrown using the **`throw`** keyword and caught using the **`try-catch`** block. The **`try`** block contains the code that may throw an exception, and the **`catch`** block contains the code that handles the exception.

```
try {
    // code that may throw an exception
    throw MyException();
}
catch (MyException& e) {
    // code to handle the exception
}
```

The **`throw`** statement can throw any type of object, including built-in types, standard library types, and user-defined types. The **`catch`** block can catch exceptions of a specific type or a base class type.
In the following example, the `safe_division()` function throws an exception of type `std::runtime_error` if the denominator is zero. The exception is caught in the `main` function and the error message is printed.

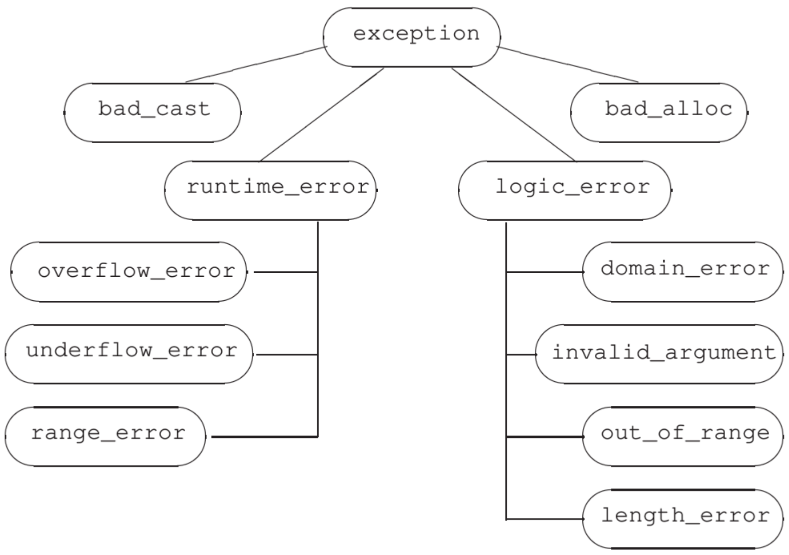
```
#include <iostream>
#include <stdexcept>
using std::cin, std::cout;

double safe_division(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero");
    }
    return a / b;
}

int main() {
    try {
        int x, y;
        cout << "Enter two numbers: ";
        cin >> x >> y;
            double result = safe_division(x, y);
        cout << "Result: " << result << "\n";
    }
    catch (std::runtime_error& e) {
        cout << "Caught an exception: " << e.what() << "\n";
    }
}
```

### The `<stdexcept>` header

The `<stdexcept>` header file contains the following class hierarchy:



The base class `std::exception` has a virtual method `what()` that returns a string describing the exception. The `std::runtime_error` class is derived from `std::exception` and is used to report runtime errors. The `std::logic_error` class is used to report errors that are caused by the program logic, such as invalid arguments to a function. Further derived classes include `std::invalid_argument`, `std::domain_error`, `std::length_error`, etc.

## Upcasting and Downcasting

*Upcasting* is the process of converting a pointer or reference of a derived class to a pointer or reference of a base class. It is done implicitly and does not require any explicit casting.
*Downcasting* is the process of converting a pointer or reference of a base class to a pointer or reference of a derived class. It is done explicitly using the **`static_cast`** or **`dynamic_cast`** operators.

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };

void f() {
    Derived d;
    Base* pb = &d;  // upcasting
    Derived* pd = static_cast<Derived*>(pb);  // downcasting
}
```

### **`static_cast`** and **`dynamic_cast`**

The **`static_cast`** and **`dynamic_cast`** operators are used for type conversion between different types of pointers and references. They can be used for upcasting, downcasting, and sideways conversions between pointers or references of classes in the inheritance hierarchy. They can also be used to convert between different types of pointers, such as converting a **`void`**`*` pointer to another type of pointer.
The **`static_cast`** is done at compile time and does not perform any runtime type checking. While the **`dynamic_cast`** is done at runtime and performs runtime type checking to ensure that the conversion is valid. When applied to pointers, the **`dynamic_cast`** returns a null pointer if the conversion is not valid. When applied to references, it throws an exception of type `std::`**`bad_cast`** if the conversion is not valid.

```
Base* pb = new Derived;
Derived* pd = dynamic_cast<Derived*>(pb);
if (pd != nullptr) {
    // conversion is valid
}
else {
    // conversion is not valid
}

Base& rb = *pb;
try {
    Derived& rd = dynamic_cast<Derived&>(rb);
    // conversion is valid
}
catch (std::bad_cast& e) {
    // conversion is not valid
}
```

Following example uses **`dynamic_cast`** to do runtime type checking and downcasting to count the number of students in a vector of `Person` pointers.

```
    class Person { /* ... */ };
    class Student : public Person { /* ... */ };
    class Staff : public Person { /* ... */ };

    int count_students(vector<Person*>& people) {
        int count = 0;
        for (Person* p : people) {
            if (dynamic_cast<Student*>(p) != nullptr) {
                count++;
            }
        }
        return count;
    }
```

# Lab 6 – Part I

## Exercise 1

Write a `SafeArray` class that use dynamic array (fixed-size) to store integers. The class should have the following methods:

- `SafeArray(int sz)`: constructor that initializes the array to size `sz` containing all 0 values.
- `SafeArray(const SafeArray& other)`: copy constructor that creates a new `SafeArray` object that is a deep copy of `other`.
- `int& operator[](int i)`: returns the reference to the element at index `i`. It throws an exception of type `std::out_of_range` if the index is out of range.
- `int size()`: returns the size of the array.
- `void print()`: prints the elements of the array.
- `~SafeArray()`: destructor that frees the memory allocated for the array.

Demonstrate the use of the `SafeArray` class in the `main` function.

```
int main() {
    SafeArray a(5);
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    a[3] = 4;
    a[4] = 5;
    a.print();
    cout << "Size: " << a.size() << endl;
    SafeArray b = a;
    b.print();
    cout << "Size: " << b.size() << endl;
    b[0] = 10;
    b.print();
    cout << "Size: " << b.size() << endl;
    try {
        b[5] = 10;
    }
    catch (std::out_of_range& e) {
        cout << "Caught an exception of type: " << e.what() << endl;
    }
}
```

## Exercise 2

Derive a class `SafeVector` from the `SafeArray` class that uses resizable array to store integers. Add the following methods to the `SafeVector` class:

- `void push_back(int val)`: adds a new element `val` at the end of the vector.
- `void pop_back()`: removes the last element from the vector. The function should throw an exception of type `std::underflow_error` if the vector is empty.
- `int back()`: returns the last element of the vector.

Which member functions need to be virtual in the `SafeArray` class? Which needs to be overridden in the `SafeVector` class?

Write a `main` function to demonstrate the use of the `SafeVector` class. Include the case when the vector is empty and the `pop_back` function is called.

## Exercise 3 ..........................................................................................

In the C++ exception mechanism, control moves from the **`throw`** statement to the first **`catch`** statement that can handle the thrown type. When the **`catch`** statement is reached, all of the automatic variables (i.e., local and argument variables) that are in scope between the **`throw`** and **`catch`** statements are destroyed in a process that is known as *stack unwinding*.

```cpp
class MyException{};

class Dummy {
    public:
    string MyName;
    int level;
    void PrintMsg(string s) {
        cout << s  << MyName <<  endl;
    }
    Dummy(string s) : MyName(s) {
        PrintMsg("Created Dummy:");
    }
    Dummy(const Dummy& other) : MyName(other.MyName) {
        PrintMsg("Copy created Dummy:");
    }
    ~Dummy() {
        PrintMsg("Destroyed Dummy:");
    }
};

void C(Dummy d, int i) {
    cout << "Entering FunctionC" << endl;
    d.MyName = "C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i) {
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}
```

```
void A(Dummy d, int i) {
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    //  Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
    //   delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main() {
    cout << "Entering main" << endl;
    try {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e) {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}
```

The output of the program is:

```
Entering main
Created Dummy: M
Copy created Dummy: M
Entering FunctionA
Copy created Dummy: A
Entering FunctionB
Copy created Dummy: B
Entering FunctionC
Destroyed Dummy: C
Destroyed Dummy: B
Destroyed Dummy: A
Destroyed Dummy: M
Caught an exception of type: class MyException
Exiting main.
```

(a) What is the order in which the `Dummy` objects are created and then destroyed as they go out of scope.?

(b) Which functions completed their execution in the above program?

(c) Uncomment the definition of the `Dummy` pointer and the corresponding **`delete`** statement, and then run the program, will the pointer gets deleted?

(d) What happens if you remove the **`throw`** statement from the `C` function?

(e) What happens if you remove the **`catch`** block from the `main` function?

# Lab 6 – Part II

## Exercise 1

Consider the following class hierarchy. The `Shape` class is an abstract base class with a pure virtual function `area()`. The `Circle` and `Rectangle` classes are derived from the `Shape` class and implement the `area()` function.

```
class Shape {
  public:
    virtual ~Shape() {}
    virtual double area() const = 0;
};

class Circle : public Shape {
  private:
    double radius;
  public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public Shape {
  private:
    double width, height;
  public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override {
        return width * height;
    }
};
```

(a) Write a function `total_area` that takes a vector of `Shape*` pointers and returns the total area of all the shapes in the vector.

(b) Write a function `circle_area` that takes a vector of `Shape*` pointers and returns the total area of all the circles in the vector (ignoring shapes other than circles).

## Exercise 2

(a) Write a function `process` having a single **`int`** parameter `x` that performs the following:

- if `x` is `1`, exception type **`int`** is thrown;
- if `x` is `2`, an exception `"yikes"` thrown;
- otherwise, nothing is done.

(b) Write a program whose `main` function performs the following for each element `x` of `std::vector<int> {0,1,2,3}`: Call `process(x)` and then print `"okay\n"` to standard output. While doing this, exceptions of type **`int`** and **`const char`**`*` should be caught. In each case, the exception handler should print the type of the exception and the value of the exception (and allow execution of the program to continue normally).

**Exercise 3** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Derive your own exception class called `CurveBall` from the `std::exception` class to represent an arbitrary error and write a function that throws this exception approximately 25% of the time. One way to do this is to generate a random integer between 0 (inclusive) and 100 (exclusive) and, if the number is less than 25, throw the exception. Define a `main()` function to call this function 1,000 times, while recording the number of times an exception was thrown. At the end, print out the final count. Of course, if all went well, this number should fluctuate somewhere around 250.

**Exercise 4** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

Create a function called `readEvenNumber()` intended to read an even integer from the `std::cin` input stream. About 25% of the time something really odd happens inside `readEvenNumber()`, resulting in a `CurveBall` exception. You can simply reuse code from previous exercise for this. Normally, however, the function verifies the user input and returns an even number if the user enters one correctly. If the input is not valid, however, the function throws one of the following exceptions:

- If any value is entered that is not a number, it throws a `NotANumber` exception.
- If the user enters a negative number, a `NegativeNumber` exception is thrown.
- If the user enters an odd number, the function throws an `OddNumber` exception.

You should derive these new exception types from `std::domain_error`, one of the standard exception types defined in the `<stdexcept>` header. Their constructors should compose a string containing at least the incorrectly entered value and then forward that string to the constructor of `std::domain_error`.

**Hint:** After attempting to read an integer number from `std::cin`, you can check whether parsing that integer succeeded by using `std::cin.fail()`. If that member function returns **`true`**, the user entered a string that is not a number. Note that once the stream is in such a failure state, you cannot use the stream anymore until you call `std::cin.clear()`. Also, the nonnumeric value the user had entered will still be inside the stream—it is not removed when failing to extract an integer. You could, for instance, extract it using the `std::getline()` function. Putting this all together, your code might contain something like this:

```
if (std::cin.fail()) {
    std::cin.clear(); // Reset the failure state
    std::string line; // Read the erroneous input and discard it
    std::getline(std::cin, line);
    ...
```

Write a `main()` function to call `readEvenNumber()` and handle the exceptions it throws. The `main()` function should catch the exceptions and print an appropriate error message. If the exception is a `domain_error`, however, you should retry asking for an even number.