Object Oriented Programming with C++

14. Exception Handling

By: Prof. Pandav Patel

Second Semester, 2020-21
Computer Engineering Department
Dharmsinh Desai University

C++ Exception Handling (Intro)

- What is compile time error?
- How should we handle runtime error?
- C++ provides a good mechanism for handling exceptions
  - Let us understand it with code

```
int main() {
        int num1, num2, result;
        cin >> num1 >> num2;
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2

Input
10 0

Output

- Like some other OOP languages. Division by zero will not raise any exception. But program will crash.

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw num2;
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    catch(int n) {      // try block must have catch block
        cout << "Exception caught " << n << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
Exception caught 0
This is a try catch demo!

- Every try block must have atleast one catch block.
- Try block detects and throws the exception using throw keyword
- Catch block catches and handles the exception
- Statements in try block which are after throw statement are not executed (if throw happens)
- Once catch block finishes its execution, statements after catch block are executed
- Catch block argument list can not be empty. And it can not have more than one argument.
- Catch block is executed only if type thrown matches the type of the catch block

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw "Division by zero was caught!";
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    catch(int n) {      // try block must have catch block
        cout << "Exception caught " << n << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output (Runtime error)
terminate called after throwing an instance of 'char const*'

If there is no matching catch block then std::terminate function would be called. Its defined in exception file

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw float(num2);
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    catch(int n) {      // try block must have catch block
        cout << "Exception caught " << n << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output (Runtime error)
terminate called after throwing an instance of 'float'

If there is no matching catch block then std::terminate function would be called. Its defined in exception file

```
int main() {
    // try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw "Division by zero was caught!";
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    // }
    // catch(int n) {
    //     cout << "Exception caught " << n << endl;
    // }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output (Runtime error)
terminate called after throwing an instance of 'char const*'

Exceptions can be thrown even when you are not in try block

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw "Division by zero was caught!";
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    catch(int n) {
        cout << "Exception caught " << n << endl;
    }
    catch(const char *str) {
        cout << "Exception caught " << str << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
Exception caught Division by zero was caught!
This is a try catch demo!

- There can be more than one catch blocks associated with try block.
- Only one matching catch block will be executed.
- It does not do implicit type coversion for matching catch block (Except converting derived class to base class - ref/pointer/object)

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        if(0 == num2) {
            throw float(num2);
        }
        result = num1 / num2;
        cout << num1 << " / " << num2 << " = " << result<<endl;
    }
    catch(int n) {
        cout << "Exception caught " << n << endl;
    }
    catch(const char *str) {
        cout << "Exception caught " << str << endl;
    }
    catch(...) {    // Notice use of ellipsis
        cout << "catch all called!!\n";
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
catch all called!!
This is a try catch demo!

- We can write a catch block which catches all the exceptions
- Catch all should be written at the last of all catch blocks. So that is only executes if previous catch blocks are not matched with the exception raised
- It is not good practice to catch exceptions which can not be handled by the code. Hence avoid catch all if that is an option

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        try{
            if(0 == num2) {
                throw "Division by zero was caught!";
            }
            result = num1 / num2;
            cout << num1 << " / " << num2 << " = " << result<<endl;
        }
        catch(const char *str) {
            cout << "Inner catch I : " << str << endl;
        }
    }
    catch(int n) {
        cout << "Exception caught " << n << endl;
    }
    catch(const char *str) {
        cout << "Exception caught " << str << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
Inner catch I : Division by zero was caught!
This is a try catch demo!

try...catch blocks can be nested. There can be try...catch nested within another try or catch block

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        try{
            if(0 == num2) {
                throw "Division by zero was caught!";
            }
            result = num1 / num2;
            cout << num1 << " / " << num2 << " = " << result<<endl;
        }
        catch(const char *str) {
            cout << "Inner catch I : " << str << endl;
            throw;
        }
        catch(const char *str) {
            cout << "Inner catch II: " << str << endl;
        }
    }
    catch(int n) {
        cout << "Outer catch I : " << n << endl;
    }
    catch(const char *str) {
        cout << "Outer catch II: " << str << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
Inner catch I : Division by zero was caught!
Outer catch II: Division by zero was caught!
This is a try catch demo!

- Catch block can **rethrow** the same exception (by using throw keyword without expression.)
- Please note that when inner catch block throws an exception, it will not be matched against remaining inner catch blocks. It will be matched directly against outer catch blocks.
- In other words, catch blocks catch throws coming from associated try block, but they do not catch throws coming from other catch blocks above them
- Rethrow (throw without expression) is allowed when an exception is presently being handled (it calls std::terminate if used otherwise)

```
int main() {
    try {
        int num1, num2, result;
        cin >> num1 >> num2;
        try{
            if(0 == num2) {
                throw "Division by zero was caught!";
            }
            result = num1 / num2;
            cout << num1 << " / " << num2 << " = " << result<<endl;
        }
        catch(const char *str) {
            cout << "Inner catch I : " << str << endl;
            throw 5;
        }
        catch(int i) {
            cout << "Inner catch II: " << i << endl;
        }
    }
    catch(int n) {
        cout << "Outer catch I : " << n << endl;
    }
    catch(const char *str) {
        cout << "Outer catch II: " << str << endl;
    }
    cout << "This is a try catch demo!";
    return 0;
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
Inner catch I : Division by zero was caught!
Outer catch I : 5
This is a try catch demo!

Catch block can throw the new exception.
Please note that when inner catch block throws an exception, it will not be matched against remaining inner catch blocks. It will be matched directly against outer catch blocks.

```
class DivideByZeroException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "DivideByZeroException raised!\n";
    }
};
int main() {
    int num1, num2, result;
    cin >> num1 >> num2;
    try {
        if(0 == num2) {
            throw *(new DivideByZeroException);
        }
        result = num1 / num2;
    }
    catch(DivideByZeroException &dbze) {
        cout << "catch DBZE: " << dbze.what();
    }
    catch(std::exception &e) {
        cout << "catch E: " << e.what();
    }
    cout << "This is a try catch demo!\n";
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
catch **DBZE**: DivideByZeroException raised!
This is a try catch demo!

- std::exception is defined in exception file. So need to #include<exception>
- It is good practice to throw exceptions which are of a type that derives from std::exception
- If what() method is not overridden in DivideByZeroException class, then dbze.what() will execute what() method of std::exception class

```
class DivideByZeroException: public std::exception {
public:
    virtual const char* what() const throw()
    {
        return "DivideByZeroException raised!\n";
    }
};
int main() {
    int num1, num2, result;
    cin >> num1 >> num2;
    try {
        if(0 == num2) {
            throw *(new DivideByZeroException);
        }
        result = num1 / num2;
    }
    catch(std::exception &e) {
        cout << "catch E: " << e.what();
    }
    catch(DivideByZeroException &dbze) {
        cout << "catch DBZE: " << dbze.what();
    }
    cout << "This is a try catch demo!\n";
}
```

Input
10 5

Output
10 / 5 = 2
This is a try catch demo!

Input
10 0

Output
catch **E**: DivideByZeroException raised!
This is a try catch demo!

- warning: exception of type ‘DivideByZeroException’ will be caught by earlier handler for ‘std::exception’
- Implicit type concersion is allowed when catch block accepts ref/pointer/object of base class and derived class object is thrown
- Catch block of specialized class should come before catch block of generalized class from which given specialized class is deriving

C++ Exception Handling (Intro)

- Good practices
  - Try block should only contain statements which may throw exception. Statements for which we do not expect any exceptions, should be kept outside try block
    - I have not followed this practive in few examples, its a mistake
  - Throw exceptions which are derived from ***std::exception*** class and avoid other types (specifically fundamental type) of exceptions.
    - We used fundamental types just for simplicity, so that concepts can be conveyed without cluttering your mind with other things.
    - Need #include<exception>
    - ***std::exception*** class has virtual method named what() which can be used to output the type of exception raised and other necessary details. You should override it when you derive from std::exception.

```
class Node {
    int i;
    Node *next;
public:
    Node() { cout << "constructor called\n"; }
    ~Node() { cout << "destructor called\n"; }
};
Node *create_node_array(int nodes) {
    Node *node = new Node[nodes];
    return node;
}
int main() {
    long number_of_nodes;
    Node *node_arr_ptr;
    while(1) {
        cin >> number_of_nodes;
        try {
            node_arr_ptr = create_node_array(number_of_nodes);
            break;
        }
        catch(std::exception &ba) { // #include<exception>
            cout << "Exception while creating node: " << ba.what() << endl;
            number_of_nodes = 0;
        }
    }
    delete [] node_arr_ptr;
    cout << "This is a try catch demo! " << node_arr_ptr;
    return 0;
}
```

Input

```
1000000000000
1000000000000
2
```

Output

```
Exception while creating node:
std::bad_alloc
Exception while creating node:
std::bad_alloc
constructor called
constructor called
destructor called
destructor called
This is a try catch demo!
0x55c07dd13f28
```

- If function does not handle the exception then it is passed to the caller to handle it
- Here exception raised by new (of type std::bad_alloc) from create_node_array will be handled in main
- Here catch block with reference of std::exception class is catching exception of std::bad_alloc. It is possible because std::bad_alloc derives from std::exception. Implicit type conversion is allowed (for the purpose of mathcing catch block) from object/reference/pointer of derived class to base class.

```
class Node {
    int i;
    Node *next;
public:
    Node() { cout << "constructor called\n"; }
    ~Node() { cout << "destructor called\n"; }
};
Node *create_node_array(int nodes) {
    Node *node = NULL;
    try {
      node = new Node[nodes];
    }
    catch(std::exception &ba) { // #include<exception>
      cout << "Exception while creating node: " << ba.what() << endl;
    }
    return node;
}
int main() {
    long number_of_nodes;
    Node *node_arr_ptr;
    while(1) {
        cin >> number_of_nodes;
        try {
            node_arr_ptr = create_node_array(number_of_nodes);
            break;
        }
        catch(std::exception &ba) { // #include<exception>
            cout << "Exception while creating node: " << ba.what() << endl;
            number_of_nodes = 0;
        }
    }
    delete [] node_arr_ptr; // It is safe to delete NULL ptr (it is like no-op)
    cout << "This is a try catch demo! " << node_arr_ptr;
    return 0;
}
```

Input

1000000000000
1000000000000
2

Output

Exception while creating node:
std::bad_alloc
This is a try catch demo! 0

- Here main exception is handles in create_node_array, hence main will not need to handle it and hence catch block in main will not execute

```
class Node {
    int i;
    Node *next;
public:
    Node() { cout << "constructor called\n"; }
    ~Node() { cout << "destructor called\n"; }
};
Node *create_node_array(int nodes) throw() {
    Node *node;
    try {
       node = new Node[nodes];
    }
    catch(std::exception &ba) { // #include<exception>
       cout << "Exception while creating node: " << ba.what() << endl;
    }
    return node;
}
int main() {
    long number_of_nodes;
    Node *node_arr_ptr;
    while(1) {
        cin >> number_of_nodes;
        try {
            node_arr_ptr = create_node_array(number_of_nodes);
            break;
        }
        catch(std::exception &ba) { // #include<exception>
            cout << "Exception while creating node: " << ba.what() << endl;
            number_of_nodes = 0;
        }
    }
    delete [] node_arr_ptr;
    cout << "This is a try catch demo! " << node_arr_ptr;
    return 0;
}
```

Input

1000000000000
1000000000000
2

Output

Exception while creating node:
std::bad_alloc
This is a try catch demo! 0

- A Function or method can specify list of exceptions it can throw **out**. It specify these exceptions by providing comma separated list of exceptions in **throw()** at end of function header.
    - e.g if function/method ends with **throw(int, double)** then it can only throw **out** int or double exception, it can not throw out other types of exceptions
- Empty throw() means that function/method can not throw **out** exception at all (because it is empty and that means function/method is not allowed to throw **out** any type of exceptions)
- throw(optional_type_list) specification, was deprecated in C++11 and removed in C++17, except for empty throw()
- In this program create_node_array handles the exception within itself and exception is not thrown out of function. Hence it is working.

```
class Node {
    int i;
    Node *next;
public:
    Node() { cout << "constructor called\n"; }
    ~Node() { cout << "destructor called\n"; }
};
Node *create_node_array(int nodes) throw() {
    Node *node = new Node[nodes];
    return node;
}
int main() {
    long number_of_nodes;
    Node *node_arr_ptr;
    while(1) {
        cin >> number_of_nodes;
        try {
            node_arr_ptr = create_node_array(number_of_nodes);
            break;
        }
        catch(std::exception &ba) { // #include<exception>
            cout << "Exception while creating node: " << ba.what() << endl;
            number_of_nodes = 0;
        }
    }
    delete [] node_arr_ptr;
    cout << "This is a try catch demo! " << node_arr_ptr;
    return 0;
}
```

Input

1000000000000
1000000000000
2

Output (runtime error)

terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc

- In this program create_node_array does not handle the exception generated from by new
- Exception generated by new can also not be thrown out of create_node_array function (notice empty throw specification at the end of its header)
- As exception can not be thrown out of create_node_array and it has not been handled within create_node_array, hence it will call std::terminate (from exception class). Whichby default aborts the program. And hence we are getting the error message shown above

```
class Node {
    int i;
    Node *next;
public:
    Node() {
        cout << "Node constructor called\n";
    }
    ~Node() {
        cout << "Node destructor called\n";
    }
};
class Node2: public Node {
public:
    int *ptr = NULL;
    Node2() {
        try {
            ptr = new int;
            cout << "Node2 constructor called\n";
            throw 5;
        }
        catch(int i) {
            delete ptr;
            // Before throwing out of constructor, compiler will emit code
            // to destroy already constructed members and base object
            // and free memory of object under construction
            throw;
        }
    }
    ~Node2() {
        delete ptr;
        cout << "Node2 destructor called\n";
    }
};
```

```
int main() {
    try {
        // block scope
        // destructor will be called when block ends
        // destructor will only be called if it was created successfully,
        // mean if there was no exception thrown out of constructor
        Node2 n2;
    }
    catch(...) {
        cout << "catch in main!\n";
    }
    cout << "This is a try catch demo!\n";
}
```

Output

```
Node constructor called
Node2 constructor called
Node destructor called
catch in main!
This is a try catch demo!
```

- In practice, almost always throw statements would be conditional, but here is example throw statements are unconditional. It is just for demonstration, you should never write code like this.
- Destructor of Node2 was not called because object n2 was not constructed successfully (as exception was thrown out of constructor, while constructing n2)
- Constructor for Node was called, because object of Node was constructed successfully without any exception coming out of constructor on Node.
- https://stackoverflow.com/questions/48282948/destroying-the-member-variables-of-an-object-when-an-exception-is-thrown-in-its

```
class Node {
    int i;
    Node *next;
public:
    Node() {
        cout << "Node constructor called\n";
    }
    ~Node() {
        cout << "Node destructor called\n";
    }
};
class Node2: public Node {
public:
    int *ptr = NULL;
    Node2() {
        try {
            ptr = new int;
            cout << "Node2 constructor called\n";
            throw 5;
        }
        catch(int i) {
            delete ptr;
            // throw;
        }
    }
    ~Node2() {
        delete ptr;
        cout << "Node2 destructor called\n";
    }
};
```

```
int main() {
    try {
        // block scope
        // destructor will be called when block ends
        // destructor will only be called if it was created successfully,
        // mean if there was no exception thrown out of constructor
        Node2 n2;
    }
    catch(...) {
        cout << "catch in main!\n";
    }
    cout << "This is a try catch demo!\n";
}
```

Output

Node constructor called
Node2 constructor called
Node2 destructor called
Node destructor called
This is a try catch demo!

- If constructor is not trowing exception out of the constructor then object (n2) will be constructed and hence it will also call the destructor of Node 2 when it goes out of scope (out of try block)
- Destructor will try delete on already deleted memory (ptr), which is undefined behaviour. We were unlucky here that it did not crash the program

C++ Exception Handling (Intro)

- Exceptions in constructor
  - Exceptions from constructor should be handled in constructor first. And then exception should be thrown outside constructor if needed (if object can not be constructed successfully)
    - Dynamically allocated memory should be deleted/freed in catch block within constructor.
    - Memory of the actual object (including other member objects) will be freed by itself.
      - Before the statement which throws exception out of the constructor, compiler will emit the code to destroy already constructed members and base object (by calling respective destructors) and it will also delete the memory of object which was under construction (memory of object whose constructor raised exception) without calling its destructor
        - https://stackoverflow.com/questions/48282948/destroying-the-member-variables-of-an-object-when-an-exception-is-thrown-in-its

```
class A {
    int i;
public:
    A(int i) {
        cout << "constructor called " << i << endl;
        this->i = i;
    }
    ~A() {
        cout << "destructor called " << i << endl;
    }
};
void h() {
    cout << "entered h()\n";
    A a3(3);
    throw 4;
    cout << "exiting h()\n";
}
void g() {
    cout << "entered g()\n";
    try {
        h();
    }
    catch(int i) {
        cout << "Caught int in g()\n";
    }
    A a2(2);
    cout << "exiting g()\n";
}
```

```
void f() {
    cout << "entered f()\n";
    A a1(1);
    g();
    cout << "exiting f()\n";
}
int main() {
    cout << "entered main()\n";
    A a(0);
    try {
        f();
    }
    catch(int i) {
        cout << "caught int in main()\n";
    }
    catch(...) {
        cout << "caught in catch all in main()\n";
    }
    cout << "exiting main()\n";
    return 0;
}
```

Output

```
entered main()
constructor called 0
entered f()
constructor called 1
entered g()
entered h()
constructor called 3
destructor called 3
Caught int in g()
constructor called 2
exiting g()
destructor called 2
exiting f()
destructor called 1
exiting main()
destructor called 0
```

- Note to myself: Explain by drawing stack frame
- When h() throws **int** exception, it is handled in g(). So all objects (having auto storage class) that were created in h() before throwing an exception need to be destructed by calling appropriate destructor. h() have created only a3 before throwing an exception, hence destructor for a3 would be called before h() passes exception to g() where it is handled.
- This process is known as stack unwinding.

```
class A {
    int i;
public:
    A(int i) {
        cout << "constructor called " << i << endl;
        this->i = i;
    }
    ~A() {
        cout << "destructor called " << i << endl;
    }
};
void h() {
    cout << "entered h()\n";
    A a3(3);
    throw '4';
    cout << "exiting h()\n";
}
void g() {
    cout << "entered g()\n";
    try {
        h();
    }
    catch(int i) {
        cout << "Caught int in g()\n";
    }
    A a2(2);
    cout << "exiting g()\n";
}
```

```
void f() {
    cout << "entered f()\n";
    A a1(1);
    g();
    cout << "exiting f()\n";
}
int main() {
    cout << "entered main()\n";
    A a(0);
    try {
        f();
    }
    catch(int i) {
        cout << "caught int in main()\n";
    }
    catch(...) {
        cout << "caught in catch all in main()\n";
    }
    cout << "exiting main()\n";
    return 0;
}
```

Output

```
entered main()
constructor called 0
entered f()
constructor called 1
entered g()
entered h()
constructor called 3
destructor called 3
destructor called 1
caught in catch all in main()
exiting main()
destructor called 0
```

- When h() throws **char** exception, it is handled by catch all block in main().
- So all objects (having auto storage class) that were created in h() before throwing an exception need to be destructed by calling appropriate destructor. h() have only created a3 before throwing an exception, hence destructor for a3 would be called before h() passes exception to g()
- Simillarly all object (having auto storage class) that were created in g() before calling h() needs to be destructed before g() passes exception to f(). But there are no objects created in g() before calling h()
- Simillarly, a1 (which has auto storage class) created in f() before calling g(), needs to be destructed before f() passes exception to main()
- This process is known as stack unwinding

```
class A {
    int i;
public:
    A(int i) {
        cout << "constructor called " << i << endl;
        this->i = i;
    }
    ~A() {
        cout << "destructor called " << i << endl;
    }
};
void h() {
    cout << "entered h()\n";
    A *a3 = new A(3);
    throw '4';
    cout << "exiting h()\n";
}
void g() {
    cout << "entered g()\n";
    try {
        h();
    }
    catch(int i) {
        cout << "Caught int in g()\n";
    }
    A a2(2);
    cout << "exiting g()\n";
}
```

```
void f() {
    cout << "entered f()\n";
    A a1(1);
    g();
    cout << "exiting f()\n";
}
int main() {
    cout << "entered main()\n";
    A a(0);
    try {
        f();
    }
    catch(int i) {
        cout << "caught int in main()\n";
    }
    catch(...) {
        cout << "caught in catch all in main()\n";
    }
    cout << "exiting main()\n";
    return 0;
}
```

Output

```
entered main()
constructor called 0
entered f()
constructor called 1
entered g()
entered h()
constructor called 3
destructor called 1
caught in catch all in main()
exiting main()
destructor called 0
```

- Destructor of object pointed by a3 is not called as part of stack unwinding process as it does not have auto storage class. This results in memory leak as a3 (pointer to an object on heap) will be lost forever without deleting object pointed by it.
- Refer to exception safety if you want to know how to avoid such situations.
  - https://en.cppreference.com/w/cpp/language/exceptions#Exception_safety

C++ Exception Handling (Intro)

- Exception in destructor
  - Since C++11, destructors default to noexcept
    - That means, by default destructors are not allowed to throw any exception. If noexcept destructor tries to throw an exception then it will result in call to std::terminate if that exception is not handled within destructor
  - Why destructors default to noexcept?
    - Because destructor would be called in order to delete all auto objects present on stack - from point where exception was thrown to the point where exception is handled (this process is known as stack unwinding).
    - If destructor throws an exception while it has been called as part of stack unwinding process, then that stack unwinding process will not be completed ever and stack unwinding for new exception (thrown from destructor) would start. It will never go back and complete stack unwinding for first exception. It would be mess.

Interesting reads

- Is it possible to have two or more active exceptions at the same time?
  - https://stackoverflow.com/questions/48088224/is-it-possible-to-have-two-or-more-active-exceptions-at-the-same-time
- Terminate called without an active exception (If you try to rethraw without active exception)
  - https://stackoverflow.com/questions/46190775/terminate-called-without-an-active-exception
- How to access exception object from catch all handler (Ans: It can not be accessed)
  - https://stackoverflow.com/questions/3132935/how-to-get-message-of-catch-all-exception
- Specifying exceptions (Exception specifications (throw, noexcept) (C++))
  - https://docs.microsoft.com/en-us/cpp/cpp/exception-specifications-throw-cpp
- Destroying the member variables of an object when an exception is thrown in its constructor in C++
  - https://stackoverflow.com/questions/48282948/destroying-the-member-variables-of-an-object-when-an-exception-is-thrown-in-its
- C++ standard exception hierarchy
  - https://stdcxx.apache.org/doc/stdlibug/18-2.html
- Creation of exception object and stack unwinding
  - https://en.cppreference.com/w/cpp/language/exceptions
  - https://en.cppreference.com/w/cpp/language/throw
- Exception safety
  - https://en.cppreference.com/w/cpp/language/exceptions#Exception_safety
- Mechanism of stack unwinding
  - https://en.cppreference.com/w/cpp/language/throw#Stack_unwinding