Object Oriented Methodology

Week – *6*, Lecture – *3*

Exception Handling in C++

SAURABH SRIVASTAVA

VISITING FACULTY

IIIT LUCKNOW

Let us write a User Manual for a toy !!

Assume that you are given a task to write a User Manual for a Toy

- How will you structure this Manual?

You have decided to include two sets of information in the Manual:

- How to operate the toy normally?
- If something doesn't go as expected, how to troubleshoot the problem?

There are two ways to structure this User Manual

- First, separate the Normal Operation from the Troubleshooting
- Second, put the Troubleshooting points next to the respective Normal Operation points

Two Ways to structure a User Manual

Normal Operation Section

- Normal Operation – Step 1
- Normal Operation – Step 2
- Normal Operation – Step 3

…

Troubleshooting Section

- Troubleshooting Scenario A
- Troubleshooting Scenario B

…

vs

Default section

- Normal Operation – Step 1
- Normal Operation – Step 2
- Troubleshooting Scenario A
- Normal Operation – Step 3
- Troubleshooting Scenario B

…

Let us write a User Manual for a toy !!

Assume that you are given a task to write a User Manual for a Toy

- How will you structure this Manual?

You have decided to include two sets of information in the Manual:

- How to operate the toy normally?
- If something doesn't go as expected, how to troubleshoot the problem?

There are two ways to structure this User Manual

- First, separate the Normal Operation from the Troubleshooting
- Second, put the Troubleshooting points next to the respective Normal Operation points

Which one would you prefer?

- Exception Handling is a phenomenon to structure code in a fashion that is similar to the first version
- The idea is to keep all the "error handling code" separate from the core logic of the application

Using `try`, `catch` and `throw`

The way to segregate the core logic from the error handling in C++ is via the use of `try` and `catch`

A `try` (or a `try` block more formally) contains a piece of code, whose execution may involve errors

- For example, any piece of code that performs division, may have to prepare for the “division by 0 problem”

A `catch` (or a `catch` block more formally) handles any errors that occur in the associated `try`

- A `catch`, thus must be preceded by a `try` block

The `try` blocks could be nested to any level

- Thus, it is fine to have a `try-catch` pair inside another `try-catch` pair
- The only requirement is that they appear contiguously (and not differently)

The try-catch phenomenon

```
{
    // Application logic
    try
    {
        // logic with probable issues
    }
    catch(relevant_type1)
    {
        // handle the exception
    }
    // more Application logic
    try
    {
        // some more Application logic
        try
        {
            // some more logic with probable issue
        }
        catch(relevant_type2)
        {
            // handle the exception
        }
    }
    catch(relevant)type3)
    {
        // handle the exception
    }
    // more Application logic
}
```

The `try` blocks can be nested – but the associated `catch` blocks must follow them (nothing should be in between)

Using `try`, `catch` and `throw`

The way to segregate the core logic from the error handling in C++ is via the use of `try` and `catch`

A `try` (or a `try` block more formally) contains a piece of code, whose execution may involve errors

- For example, any piece of code that performs division, may have to prepare for the “division by 0 problem”

A `catch` (or a `catch` block more formally) handles any errors that occur in the associated `try`

- A `catch`, thus must be preceded by a `try` block

The `try` blocks could be nested to any level

- Thus, it is fine to have a `try-catch` pair inside another `try-catch` pair
- The only requirement is that they appear contiguously (and not differently)

To execute a particular piece of error handling code, the `throw` keyword is used

- Typically, a `throw` statement is placed inside a conditioned block like `if` or `switch`, e.g.

```
if(<error condition>)
    throw …
```

A typical Exception Handling code

```
statement_i;
try
{
    statement_j;
    if(<error condition>_x)
        throw <variable or object of Type_a>;
    statement_k;
    if(<error condition>_y)
        throw <variable or object of Type_b>;
    statement_l;
}
catch(Type_a var)
{
    // error handling for <error condition>_x
}
catch(Type_b var2)
{
    // error handling for <error condition>_y
}
statement_m;
```

The sequence of events that occur here are as follows:

1. $statement_i$ is executed
2. $statement_j$ is executed
3. A check for $<error\text{-}condition>_x$ is made
   a. If the condition is true, the control transfers to the first $catch$ block (one with $Type_a$ variable)
   b. Otherwise, $statement_k$ is executed
4. A check for $<error\text{-}condition>_y$ is made
   a. If the condition is true, the control transfers to the second $catch$ block (one with $Type_b$ variable)
   b. Otherwise, statementl is executed
5. Irrespective of the flow, $statement_m$ is executed

The terminate() routine

Assume that a `throw` uses the variable of a type for which there is no matching `catch`

- This also includes any enclosing `try-catch` pairs, in case there is nesting

Then, a system routine called `terminate()` is invoked

- By default, it invokes another routine called `abort()`, which stops the execution of the program

You can, by the way, define a *default* catch block, which is invoked if no other catch matches

- Something like `catch(…) { // default error handler }`

Also, assume that a `throw` appears ”out-of-place”, i.e. it is not within any `try` block

In this case too, the `terminate()` routine is called

- You can, provide your own terminate handler, if you wish
- We will not cover this in the class though !!

Let us now see some code !!

Homework !!

Read this reference to know more about Exception Handling in C++

- http://www.cplusplus.com/doc/tutorial/exceptions/

Modify the code that we saw and include special validation for the battery-operated toys

- **Hint**: You can extend the `ToyValidator` class

Additional Reading

If you managed to get the book – *C++: The Complete Reference, by Herbert Schildt, 4th Edition*

- Read Chapter 19