C++ Foundation with Data Structures

Exception Handling

Exception Handling

An exception is a problem or error that arises during the execution of a program. There could be errors that cause the programs to fail or certain conditions that lead to errors. If these run time errors are not handled by the program, OS handles them and program terminates abruptly, which is not good. Few of such errors or error conditions are divide by zero, out of bound index, accessing memory not allowed to access, etc.

To avoid such conditions, C++ provides exception handling mechanism.
C++ exception handling is built upon three keywords: **try, catch, and throw**.

1. **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.
2. **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception
3. **try** − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks

Why we need Exception Handling:

1. Exception handling provide a way to transfer control from one part of a program to another and can avoid the brupt termination of program.

```
int main() {
    int n1, n2;
    try {
        cout << "Enter two nos:";
        cin >> n1 >> n2;
        if (n2 == 0)
            throw "Divide by zero";
        else
            throw n1/n2;
    }catch (char *s) {
        cout << s;
    }
    catch (int ans){
        cout << ans;
    }
    cout << "Done";
}
```

   In the above example we can see if user enters n2 as 0, then we will throw an exception and we can avoid abrupt termination of program.

2. Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

3. Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are

thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

4. Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types

Some examples to understand exception handling better

1. Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks

```
#include <iostream>
using namespace std;
int main() {
    int x = -1;
    cout << "Before try" << endl;
    try{
        cout << "Inside try \n";
        if (x < 0) {
            throw x;
            cout << "After throw" << endl;
        }
    } catch(int x) {
        cout << "Exception Caught" << endl;
    }
    cout << "After catch" << endl;
}
```

Output :

```
Before try
Inside try
Exception Caught
After catch
```

In the above example we can see the normal flow of throw, catch block.

We can observe that after the exception is thrown, the remaining try block after throw statement is not executed, instead program control goes to catch and never comes back. But all the code below the catch block is executed.
So to conclude code after throw is never executed but all the code after catch gets executed.

2. Above example shows that when exception is caught in try block it goes to a catch block whose parameter matched the throws argument, but if let's suppose I choose x to be double, then I need another catch block for that, which accepts double variable. To handle such conditions we have a special all catch block, which catches all types of exceptions.

```
#include <iostream>
```

```
using namespace std;
int main() {
    try {
        throw 10;
    } catch (char c) {
        cout <<  "character type exception" << endl;
    } catch(...) {
        cout << "Default Exception" << endl;
    }
}
```

Output:
Default Exception

We can see here that catch(...) catches all type of exceptions.
Few things about all catch blocks :

a) If there are multiple catch blocks, all catch block should be placed last.
b) Since all catch block (generic catch) is placed last, if any specific type exception occurs, it will try to find its specific catch block and if it is not found then it will go to the generic catch block.

3. Implicit type conversion doesn't happen for primitive types.

```
#include <iostream>
using namespace std;
int main() {
    try {
        throw 10;
    } catch (double c) {
        cout <<  "integer type exception" << endl;
    } catch(...) {
        cout << "Default Exception" << endl;
    }
}
```

Output :
Default Exception

Here in the above example, "throw 10" should be Implicitly get converted to double, but it doesn't get converted. Instead it get caught in generic catch block. So Implicit type casting doesn't happen for primitive type.

4. If an exception is thrown and not caught anywhere, the program terminates abnormally.

```
#include <iostream>
using namespace std;
int main() {
    try{
        char c = 'a';
        throw c;
    } catch(int x) {
        cout << "integer exception" << endl;
    }
}
```

Output :

```
terminating with uncaught exception of type char
Abort trap: 6
```

If the catch block for any throw doesn’t exists it will terminate abruptly.

5. Nested try blocks

```
#include <iostream>
using namespace std;
int main() {
    try {
        try  {
            throw 20;
        } catch (int n) {
            cout << "Inner catch " << endl;
            throw;   //Re-throwing an exception
        }
    } catch (int n) {
        cout << "Outer catch" << endl;
    }
}
```

Output :

```
Inner catch
Outer catch
```

A function can also re-throw a function using same “throw“. A function can handle a part and can ask the caller to handle remaining.

6. When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```
#include <iostream>
using namespace std;
class Test {
public:
           Test() { cout <<  "Constructor called" << endl; }
           ~Test() { cout << "Destructor called"  << endl; }
};
int main() {
           try {
            Test t1;
            throw 10;
        } catch(int i) {
            cout << "Caught " << i << endl;
        }
}
```

Output:

```
Constructor called
```

```
Destructor called
Caught 10
```

Define New Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Below is the example, which shows how you can use exception class to implement your own exception

```
#include <iostream>
#include <exception>
using namespace std;

class MyException : public exception {
    virtual const char * what () const throw () {
        return "C++ Exception";
    }
}myex;
int main() {
    try {
        throw myex;
    } catch(exception& e) {
        std::cout << e.what() << endl;
    } catch(MyException& e) {
        //Other errors
    }
}
```

Output for the above code :

```
MyException caught
C++ Exception
```

what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.