# ASSIGNMENT 9

# Exception Handling

**Problem Statement** :-
Create a C++ class named Television that has data members to hold the model number and the screen size in inches, and the price. Member functions include overloaded insertion and extraction operators. If more than four digits are entered for the model, if the screen size is smaller than 12 or greater than 70 inches, or if the price is negative or over $5000 then throw an integer. Write a main() function that instantiates a television object, allows user to enter data and displays the data members .If an exception is caught,replace all the data member values with zero values.

**Learning Objectives** :-
Exception handling

## **Theory** :-**Exception handling**

**Exception handling** is the process of responding to the occurrence, during computation, of *exceptions* – anomalous or exceptional events requiring special processing – often changing the normal flow of program execution. It is provided by specialized programming language constructs or computer hardware mechanisms.

In general, an exception is *handled* (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an *exception handler*. If exceptions are *continuable*, the handler may later resume the execution at the original location using the saved information. For example, a floating point divide by zero exception will typically, by default, allow the program to be resumed, while an out of memory condition might not be resolvable transparently.

Alternative approaches to exception handling in software are error checking, which maintains normal program flow with later explicit checks for contingencies reported using special return values or some auxiliary global variable such as C's errno or floating point status flags; or input validation to preemptively filter exceptional cases.

Excluding minor syntactic differences, there are only a couple of exception handling styles in use. In the most popular style, an exception is initiated by a special statement (`throw`, or `raise`) with an exception object (e.g. with Java or Object Pascal) or a value of a special extendable enumerated type (e.g. with Ada). The scope for exception handlers starts with a marker clause (`try`, or the language's block starter such as `begin`) and ends in the start of the first handler clause (`catch`, `except`, `rescue`). Several handler clauses can follow, and each can specify which exception types it handles and what name it uses for the exception object. A few languages also permit a clause (`else`) that is used in case no exception occurred before the end of the handler's scope was reached.

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.

- **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.
- **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.

```
try {
  line = console.readLine();

  if (line.length() == 0) {
    throw new EmptyLineException("The line read from console was empty!");
  }

  console.printLine("Hello %s!" % line);
  console.printLine("The program ran successfully");
}
catch (EmptyLineException e) {
  console.printLine("Hello!");
}
catch (Exception e) {
  console.printLine("Error: " + e.message());
}
finally {
  console.printLine("The program terminates now");
}
```

## Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

## Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
   // protected code
}catch( ExceptionName e )
{
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try
{
   // protected code
}catch(...)
{
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 0;
   double z = 0;

   try {
     z = division(x, y);
     cout << z << endl;
   }catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

## Operator overloading

Operator overloading is used because it allows the developer to program using notation closer to the target domain and allows user-defined types a similar level of syntactic support as types built into the language. It is common, for example, in scientific computing, where it allows computational representations of mathematical objects to be manipulated with the same syntax as on paper.

Object overloading can be emulated using function calls; for example, consider the integers a, b, c:

```
a + b * c
```

### Overloading the Insertion and Extraction Operators

- The I/O Stream library overloads the left shift and right shift operators as insertion and extraction operators, respectively
- When creating your own classes it is usually more relevant to overload these operators as insertion and extraction operators
- These overloaded operators *cannot be defined as member functions*
- They may be defined as global functions or **friend functions**. Usually, these operators

require access to private or protected data members of a class, therefore, they are usually defined as friend functions of the class

- Example of overloaded insertion and extraction operators:

```
#include <iostream.h>
#include <stdlib.h>

class Date{
public:
  Date() : day(0), month(0), year(0) {}
  Date(int m, int d, int y) : day(d), month(m), year(y) {}
  friend ostream& operator<<(ostream& , Date& );
  friend istream& operator>>(istream& , Date& );
private:
  int day, month, year;
};

ostream& operator<<(ostream& out, Date& DateIn){
  out << DateIn.month << '/' << DateIn.day << '/' << DateIn.year << endl;
  return out;
}

istream& operator>>(istream& in, Date& InDate){
  cout << "\nEnter another date in mm/dd/yyyy format: ";
  char temp[5];
  in.get(temp, 3, '/');
  InDate.month = atoi(temp);
  in.ignore();
  in.get(temp, 3, '/');
  InDate.day = atoi(temp);
  in.ignore();
  in.get(temp, 5, '/');
  InDate.year = atoi(temp);
  in.ignore();


  return in;
}


int main(){
  Date today(11, 4, 1997), anotherday;
  cout << "Today\'s Date: " << today << endl;

  cin >>  anotherday;

  cout << "\nThe new date is: " << anotherday << endl;

  return 0;
}
```

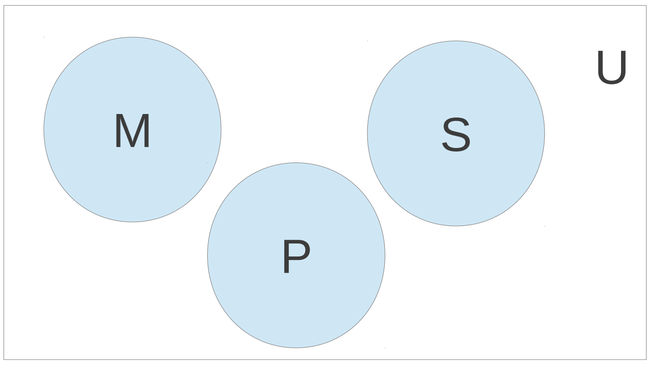
**Output:**

```
Today's Date: 11/4/1997

Enter another date in mm/dd/yyyy format: 12/1/1997

The new date is: 12/1/1997
```

**Mathematical Model:**

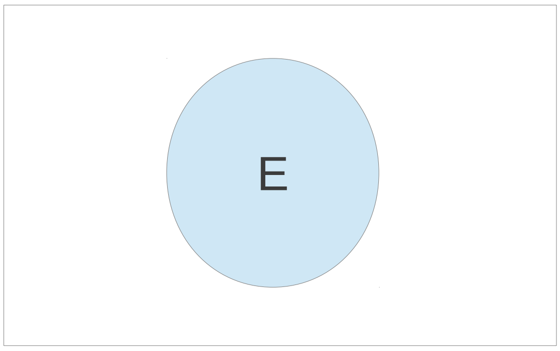
Input:

Let M={Set of model no}
P={set of Prices}
S={set of size}



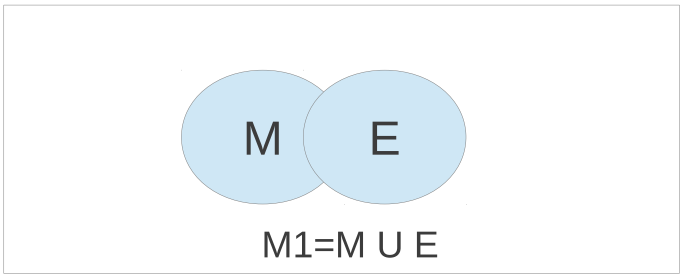
Function:
To accept data we take data from user so to detect the exception in try we check the condition and then display the exception message accordingly.

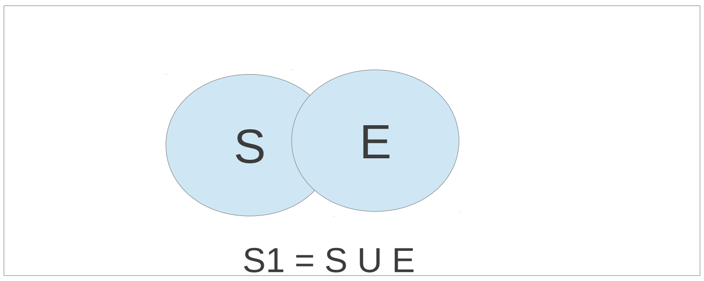
Let E={set of condition of exception}

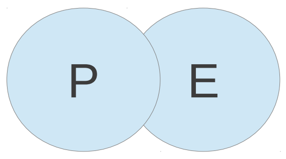


Output:
1. M1={set of exception of model number}
2. S1={set of exception of size}
3. P1={set of exception of price}

if exception is occurred







P1 = P U E

**Algorithm**

- step1:-declare class television
    - data members:-
    - i)model no.
    - ii)screen size
    - iii)price
- step 2:-overloaded '>>' operator to define accept function.
- step 3:-read model no. screen size and price from the user.
- step 4:-if(model no. is>9999)
    - throw(1)
        - if(screensize is <12 inches or >>0 inches)
    - throw(2)
        - if(price is >5000)
    - throw(3)
- step 5:-if exception occurs display message "invalid entry" and initialize all variables to zero.
- step 6:-overload '<<' operator to display.
- step 7:-create object of television class and initialize it
- step 8:-accept data and display.
- step 9:-check the exceptions opr the conditional statement.
- step 10:-stop

**Conclusion** :-
Using exception handling and operator overloading we can perform operations