⇒ Exception handling allows you to manage runtime errors in a systematic manner.

⇒ Using Exception handling, your pgm can automatically invoke an error-handling routine when an error occurs.

⇒ C++ Exception handling is built upon 3 keywords:

try →
catch
throw.

try block:- the pgm stmts that we want to monitor for Exceptions are contained in a try block.

throw block: if an Exception (ie an error) occurs within the try block, it is thrown (using throw).

catch block: the Exception is caught, using catch and processed.

Note:- Exceptions that are thrown are caught by a catch stmt, which immediately follows the try stmt.

Syntax:

```
try {
        // try block
}
catch (type1 arg)
{
    // catch block
}
catch (type2 arg)
{
    // catch block
}
.
.
.
```

Note: When Exception is thrown, it is caught by its corresponding catch stmt, which processes the Exception.

Note: There can be more than one catch stmt associated with a try.

⇒ which catch stmt is used is determined by the type of the Exception.

Note: when an Exception is caught, arg will receive its value.

⇒ Any type of data may be caught, including classes that we create.

⇒ if no Exception is thrown, then no catch stmt is Executed.

General form of the throw is:

throw Exception;

Note: if you through an exception for which there is no applicapable catch stmt, an abnormal pgm termination may occur.

Note: Throwing an unhandled Exception causes the standard library function terminate() to be invoked.

⇓

By default, terminate() calls abort() to stop your program.

Exception Handling

⇒ Exception Handling in C++ is a process to handle runtime errors.

```
#include <iostream>
float division(int x, int y)
{
    return (x/y);
}
int main()
{
    int i=50;
    int j=0;
    float k=0;
    k=division(i,j);
    cout<<k;
    cout<<"Hello";
    return 0;
}
```

O/P:

Floating point Exception (Core dumped)

```
int main()
{
    int i=50;
    int j=0;
    float k=i/j;
    cout<<k;
    return 0;
}
```

O/P: floating point Exception (Core dumped)

throwing & Catching Exceptions.

```
float division(int x, int y)
{
    if(y==0)
    {
        throw "Attempt to divide by zero";
    }
    return (x/y);
}
```

```
int main()
{
    int i=50;
    int j=0;
    float k;
    try {
        k=division(i,j);
        cout<<k;
    }
    catch (const char* e)
    {
        cout<<e;
    }
    cout<<"hello";
    return 0;
}
```

O/P: Attempt to Divide by zero
hello.

Advantages of Exceptions:

When we perform Exception handling the normal flow of the Application can be maintained even after runtime errors.

Note:- in C++ Exception is an event or object which is thrown at runtime.

⇒ It is a runtime error which can be handled. if we don't handle the Exception, it prints Exception menage and terminates the pgm.

Note:- All Exceptions are derived from std:: exception class.

Ex:-

```
#include <iostream>
using namespace std;
int main()
{
    try {
        cout << "Inside a try block
        throw 100; // throws an error
        cout << " Hello how are you";
    }
    catch (int i)
    {
        cout << " caught an exception value is" << i;
    }
}
```

O/p: Inside a try block
caught an Exception value is: 100

Note:- Once an exception is thrown, control passes to the catch and the try block is terminated.

Note: The type of the exception must match the type specified in a catch stmt.

Ex:-

```
int main()
{
    try { cout << "Inside try block"
        throw 100;
    } cout << " Hello";
    catch (double i)
    {
        cout << " caught an Exception value is" << i;
    }
}
```

O/p: inside try block
terminate called after throwing an instance of int.

Abnormal termination will occur.

Note: An exception can be thrown from outside the try block as long as it is thrown by a f^n that is called from within try block.

```
/* Throwing an exception from a f^n outside the try block */
void xtest(int i)
{
  cout<<"Inside xtest, i value is :"<<i<<endl;
  if(i) throw i;
}
int main()
{
  try {
      xtest(0);
      xtest(1);
      xtest(2); // not executed
  }
  catch (int i)
  {
  cout<<" caught an exception value is<<i<<endl;
  }
}
```

O/P

```
inside xtest ivalue is:0
inside xtest value is:1
caught an exception value
is 1
```

Note: A try block can be localized to a function. When this is the case, each time the f^n is entered, the exception handling relative to that f^n is reset.

/* localize a try/catch to a function */

```
void xhandler(int test)
{
  try {
    if(test) throw test;
  }
  catch(int i)
  {
    cout << "caught Exception # : " << i << endl;
  }
}
int main()
{
  xhandler(1);
  xhandler(2);
  xhandler(3);
  return 0;
}
```

O/P:
caught Exception # 1
caught Exception # 2
caught Exception # 3

/* pgm to read a no. and display its reciprocal. it uses exception handling to guard against division by zero.

```
float reciprocal(int K);
void main()
{
  int K;
  cout << "enter any no";
  cin >> K;
  try {
    float r = reciprocal(K);
    cout << "its reciprocal is" << r;
  }
  catch(int i)
  {
    cout << "Reciprocal of zero is not defined";
  }
}
float reciprocal(int K)
{
  if(K==0) throw(10);
  return (1/(float) K);
}
```

Catching class types / Throwing of objects

⇒ An Exception can be of any type, including class types that we create.

⇒ Actually, in real-world pgms, most Exceptions will be class types rather than built-in types.

⇒
```
#include <iostream>
#include <cstring>
using namespace std;
class MyException
{
    public:
        char mymsg[30];
        MyException()
        {
            *mymsg = 0;
        }
        MyException(char *s)
        {
            strcpy(mymsg, s);
        }
};
int main()
{
    int K;
    try
    {
        cout<<"enter a positive no";
        cin>>K;
        if(K<0)
            throw MyException("negative-no");
    }
    catch(MyException e)
    {
        cout<<e.mymsg;
    }
    return 0;
}
```

Using multiple catch stmts

→ A single try block can have multiple catch blocks.

Ex:-

```
int main()
{ int n;
  cout<<"enter any no";
  cin>>n;
      try {
            if(n>0) throw n;
            else throwsh "negative";
          }
      catch (int ex)
        {
          cout<<" positive Integer Exception";
        }
      catch (const char *ex)
        {
          cout<<" negative no Exception";
        }
  return 0;
}
```

Catching All Exceptions: || Generalized catch block

⇒ In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This can be achieved by the following form of catch block.

```
catch (...)
  {
    . stmts;
    = =
  }
```

Ex:-

```
void xhandler (int test)
{
  try
  {
  }
```

```
int main()
{
  int n;
  cout<<"enter any no";
  cin>>n;
  try {
      if(n==0) throw n; // throw int
      if(n=1) throw 'K'  // throw char
      if(n=2) throw "Koushik" // throw double
  }
  catch(...)   // catch all Exceptions.
  {
    cout<<"I can caught all";
  }
}
```

need not to write the following individual catch blocks

```
catch (int ex)
{
  cout<<"handling int";
}
catch (char ch)
{
  cout<<"handling char";
}
catch (const char *ch)
{
  cout<<"handling string";
}
```

Exception specifications (throw() | noexcept ⑥
or
noexcept(true)

(or)

Restring Exceptions in C++

⇒ Exception specification is a C++ language feature that ~~we can~~ allows to specify the type of exceptions that can be propagated by a function.

⇒ with the help of the exception specification, you can restrict the type of exceptions that a function can through outside of itself.

Note:- In fact, we can also prevent a f° from throwing any exceptions whatsever.

Note:- The compiler can use this information to optimize calls to the function, and to terminate the pgm if an unexpected exceptions arises.

Note:- To accomplish these restrictions, you must add a throws clause to a function definition.

Syntax: is

```
ret-type f°name (arg-list) throw (type-list)
{
    ||
}
```

Note:- Here, only those data types contained in the type-list may be thrown by the function.

⇓

Throwing other type of exceptions will cause abnormal pgm termination.

Note: if you don't want a f^n to be able to throw any exceptions, then use an empty list.

Note: Attempting to throw an exception that is not supported by a f^n will cause the standard library f^n unexpected() to be called. by default, this causes abort() to be called. which causes abnormal termination.

```
void xhandler (int test) throw (int, char, double)
 {
    if (test==0) throw test;
    if (test==1) throw 'a';
    if (test==2) throw 12.23;
 }

int main()
 {
    try { xhandler(0); // also try by passing 1 & 2
        }
    catch ( int i)
     { cout<<" caught an integer";
     }
    catch (char c)
     { cout<<" caught char\n";
     }
    catch (double d)
     {
       cout<<"caught double";
     }
 }
```

```
void xhandler (int test) throw()
 {
    //
 }
```

is equivalent to

⇓

```
void xhandler (int test) noexcept;
```

or

```
void xhandler (int test) noexcept(true);
```

Rethrowing an Exception

⇒ if you wish to rethrow an exception from within an exception handler, you may do so by calling throw, by itself, with no exception.

⇒ This causes the current exception to be passed on to an outer try/catch sequence.

> Note:- The most reason for doing so is to allow multiple handlers access to the exception.

⇒ An exception can only be rethrown from within a catch block (or any f^n called from within that block)

Ex:-

```
void xhandler()
{
  try{
    throw "hello";
  }
  catch(const char*)
  {
    cout<<"caught char* inside xhandler\n";
    throw; //rethrow char* out of f^n.
  }
}
int main()
{
  try{
    xhandler();
  }
  catch(const char*)
  {
    cout<<"caught char* inside main\n";
  }
}
```

o/p

```
caught char* inside xhandler
caught char* inside main.
```

Stack unwinding in C++

⇒ The process of removing f^n entries from function call stack at runtime is called stack unwinding.

⇒ stack unwinding is used in exception handling.

Note: in C++, when an exception occurs, the f^n call stack is linearly searched for the exception handler and all the entries before the function with exception handler are removed from the f^n call stack.

Note:- So exception handling involves stack unwinding if exception is not handled in same function (where it is thrown)

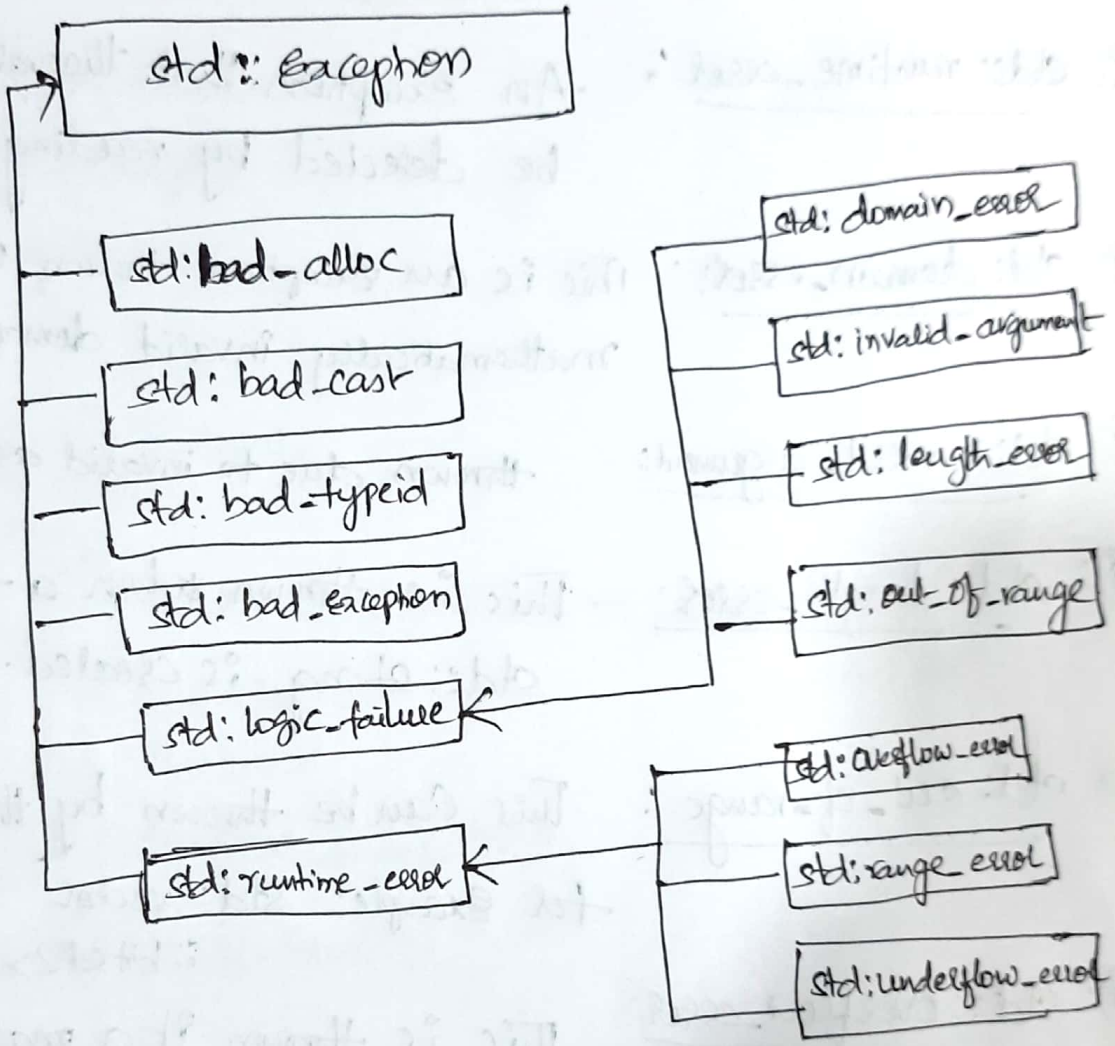
Ex:-
```
void f1() throw (int)  // f1 that throws an int exception.
{
    cout<<"\n f1() start";
    throw 100;
    cout<<"\n f1() end";
}

void f2() throw (int)
{
    cout<<"\n f2() start";
    f1();
    cout<<"\n f2() end";
}

void f3()
{
    cout<<"\n f3() start";
    try{
        f2();
    }
    catch(int i)
    {
        cout<<"\n caught Exception: "<<i;
    }
    cout<<"\n f3() end";
}

int main()
{
    f3();
    return 0;
}
```

O/P:
```
f3() start
f2() start
f1() start
Caught Exception: 100
f3() end.
```

C++ Exception classes

⇒ in C++ standard exceptions are defined in <exception> class that we can use inside our programs.

⇒ The hirarchy of stand Exception classes in C++ is as follows.



① std::exception: it is an exception and parent class of all the standard C++ exceptions.

② std::bad_alloc: This exception is generally thrown by new

③ std::bad_cast: Generally thrown by dynamic_cast.

④ std: bad_typeid : Thrown by typeid.

⑤ std: bad_exception: it is used to handle unexpected exceptions in a c++ pgm.

⑥ std:: logic_failure: it is an exception that can be detected by reading a code.

⑥ std:: runtime_error: An exception that theoretically can not be detected by reading the code.

⑦ std: domain_error: This is an exception thrown when a mathematically invalid domain is used.

⑧ std: invalid_argument: thrown due to invalid arguments

⑨: std:: length_error: This is thrown when a too big std:: string is created.

⑩ std: out_of_range: This can be thrown by the 'at' method, for Example: std:: vector
:: bitset<>

⑪ std:: overflow_error: This is thrown if a mathematical overflow occurs.

⑫ std:: range_error: This is occured when you try to store a value which is out of range.

⑬ std:: underflow_error: thrown if a mathematical underflow occurs.

⇒ <exception> is the headerfile where exception is available.

⇒ There are two main derived clases from the the exception clas (base clas for all standad exceptions) are <logic_failure> clas & <runtime_error> clas.

⇓ These are defined in other header file <stdexcept> header file.

```
# include<iostream>
# include<stdexcept>
using namespace std;
int main()
{
    int a=5, b=0;
    float k;
    try
    {
        if(b==0) throw runtime_error("divide by zero cant do");
        k=a/b;
        cout<<k;
    }
    catch(runtime_error re)
    {
        cout<<re.what();
    }
    return 0;
}
```

O/P: divide by zero cant do.

C++ User-Defined Exceptions

→ The new exception can be defined by overriding and inheriting exception class functionality.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception
{
public:
    const char* what() throw()
    {
        return "Attempt to divide by zero";
    }
};
int main()
{
    int x, y;
    cout<<"enter two nos";
    cin>>x>>y;
    try {
        if(y==0)
        {
            myException obj;
            throw obj;
        }
        else
        {
            cout<<"x/y is:"<<float(x)/y;
        }
    }
    catch(exception e)
    {
        cout<<e.what();
    }
    return 0;
}
```

it will return an exception string should have return type char*

Note:- what() is a public method provided by exception class. it is used to return the cause of an exception.