UNIT-V
EXCEPTION HANDILING

INDRODUCTION

Common types of Errors

The common types of errors are logic errors and syntactic errors.

Logic Errors: These occur due to poor understanding of the problem and solution procedure.
Examples: Assigning a value to the wrong variable, multiplying 2 numbers instead of adding them etc.

Syntactic Errors: These occur due to poor understanding of the language itself.
Examples: Spelling mistakes, missing out quotes or brackets or semicolon etc.

Apart from these two, one more type of errors is **Exception**.

Definition of Exception: Exceptions are run time errors or unusual conditions that a program may encounter while executing.
Examples: Division by zero, access to an array outside of its bounds, running out of memory or disk space.

Exception Handling

It is a C++ built in language feature that allows us to manage run time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

BENEFITS OF EXCEPTION HANDLING

1. **Automation:** it automates much of the error-handling code that previously had to be coded "by hand" in any large 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 can handle any exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them.
4. **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception. We can group or categorize them according to types.
5. **Handles** the occurring of error and allows normal execution of the program.

EXCEPTION HANDLING FUNDAMENTALS
(THE TRY BLOCK, CATCHING AN EXCDEPTION, THROWING AN EXCCEPTION)

C++ exception handling is built upon three keywords: **try, catch, and throw**

- The program statements that you want to monitor for exceptions are contained in a **try block.**
- If an exception (i.e., an error) occurs within the try block, it is thrown using **throw**
- When an exception is thrown, it is caught by its corresponding **catch statement**, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception.

General form of try and catch

```
try {
        // try block
}
catch (type1 arg)
{
        // catch block
}
catch (type2 arg)
 {
        // catch block
 }
catch (type3 arg)
 {
        // catch block
 }
         . . .

catch (typeN arg)
{
        // catch block
 }
```

General form of the throw

```
        throw exception;
```

Program:

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                 // start a try block
    cout << "Inside try block\n";
    throw 100;                      // throw an error
    cout << "This will not execute";
  }
  catch (int i)
  {                                 // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
```

```
}
```

Output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Abnormal Termination

The type of the exception must match the type specified in a catch statement. Usually, the code within a catch statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the catch. However, often an error cannot be fixed i.e. throw an exception for which there is no applicable catch statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate ( )** to be invoked. By default, terminate ( ) calls **abort( )** to stop your program.

Program:

```
// This example will not work.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                              // start a try block
    cout << "Inside try block\n";
    throw 100;                   // throw an error
    cout << "This will not execute";
  }
  catch (double i)
  {                              // won't work for an int exception
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

Throwing an exception from outside the try block

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block.

Program:

```
/* Throwing an exception from a function outside the try block. */
#include <iostream>
using namespace std;
void
Xtest (int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if (test)
    throw test;
}

int main ()
{
  cout << "Start\n";
  try
  {                                   // start a try block
    cout << "Inside try block\n";
    Xtest (0);
    Xtest (1);
    Xtest (2);
  }
  catch (int i)
  {                                   // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

Localize a try/catch to a function

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

Program:

```
#include <iostream>
using namespace std;
// Localize a try/catch to a function.
void
Xhandler (int test)
{
  try
  {
    if (test)
      throw test;
  }
  catch (int i)
  {
    cout << "Caught Exception #: " << i << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

Catch statements when no exception is thrown

The code associated with a catch statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the catch altogether. When no exception is thrown, the catch statement does not execute.

Program:

```
#include <iostream>
using namespace std;
int main ()
{
```

```
 cout << "Start\n";
 try
 {                              // start a try block
  cout << "Inside try block\n";
  cout << "Still inside try block\n";
 }
 catch (int i)
 {                              // catch an error
  cout << "Caught an exception -- value is: ";
  cout << i << "\n";
 }
 cout << "End";
 return 0;
}
```

Output:

```
Start
Inside try block
Still inside try block
End
```

Using multiple catch Statements

There can be more than one catch associated with a try. However, each catch must catch a different type of exception. Which catch statement is used is determined by the type of the exception.

Program:

```
#include <iostream>
using namespace std;
// Different types of exceptions can be caught.
void
Xhandler (int test)
{
 try
 {
  if (test)
    throw test;
  else
    throw "Value is zero";
 }
 catch (int i)
 {
  cout << "Caught Exception #: " << i << '\n';
 }
 catch (const char *str)
 {
```

```
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

CATCHING ALL EXCEPTIONS

Using multiple catch statements we can write a separate catch statements for each type. But this is a complicated task. In these circumstances we want an exception handler to catch all exceptions instead of just a certain type.

General form:

```
catch(...)
{
// process all exceptions
}
```

Here, the ellipsis matches any type of data.

Program:

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void
Xhandler (int test)
{
  try
  {
    if (test == 0)
```

```
    throw test;             // throw int
   if (test == 1)
    throw 'a';              // throw char
   if (test == 2)
    throw 123.23;                   // throw double
  }
  catch ( ...)
  {                                 // catch all exceptions
   cout << "Caught One!\n";
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (0);
  Xhandler (1);
  Xhandler (2);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught One!
Caught One!
Caught One!
End
```

One very good use for catch (...) is as the last catch of a cluster of catches which catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

RETHROWING AN EXCEPTION

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

Reason: It allows multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement.

Program:

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
```

```
void Xhandler ()
{
  try
  {
    throw "hello";                  // throw a char *
  }
  catch (const char *)
  {                                 // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw;                          // rethrow char * out of function
  }
}

int main ()
{
  cout << "Start\n";
  try
  {
    Xhandler ();
  }
  catch (const char *)
  {
    cout << "Caught char * inside main\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

EXCEPTION SPECICATION (RESTRICTING EXCEPTIONS)

You can restrict the type of exceptions that a function can throw outside of itself i.e. we are restricting a function to throw only certain specified exceptions .To accomplish these restrictions, we must add a throw clause to a function definition.

General form

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

only those data types contained in the comma-separated *type-list* may be thrown by the function. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected ( )** to be called. By default, this causes **abort( )** to be called, which causes abnormal program termination.

Program:

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler (int test)
throw (int, char, double)
{
  if (test == 0)
    throw test;  // throw int
  if (test == 1)
    throw 'a';  // throw char
  if (test == 2)
    throw 123.23;    // throw double
}

int main ()
{
  cout << "start\n";
  try
  {
    Xhandler (0);       // also, try passing 1 and 2 to Xhandler()
  }
  catch (int i)
  {
    cout << "Caught an integer\n";
  }
  catch (char c)
  {
    cout << "Caught char\n";
  }
  catch (double d)
  {
    cout << "Caught double\n";
  }
  cout << "end";
  return 0;
}
```

Output:

```
start
Caught an integer
```

end

A function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a function may throw any type of exception so long as it is caught *within* that function.The restriction applies only when throwing an exception outside of the function.

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
/* The following statements no longer work. Instead,
they will cause an abnormal program termination. */
if(test==0) throw test;
if(test==1) throw 'a';
if(test==2) throw 123.23;
}
```

STACK UNWINDING

Stack unwinding is a process of calling all destructors for all automatic objects constructed at run time when an exception is thrown. The objects are destroyed in the reverse order of their formation.

When an exception is thrown, the runtime mechanism first searches for an appropriate matching handler (catch) in the current scope. If no such handler exists, control is transferred from the current scope to a higher block in the calling chain or in outward manner. - Iteratively, it continues until an appropriate handler has been found. At this point, the stack has been unwound and all the local objects that were constructed on the path from a try block to a throw expression have been destroyed. - The run-time environment invokes destructors for all automatic objects constructed after execution entered the try block. This process of destroying automatic variables on the way to an exception handler is called stack unwinding.

Program:

```
#include <iostream>
#include <string>

using namespace std;

class MyClass
{
private:
  string name;
public:
  MyClass (string s):name (s)
  {
  }
   ~MyClass ()
```

```
  {
    cout << "Destroying " << name << endl;
  }
};

void fa ();
void fb ();
void fc ();
void fd ();

int main ()
{
  try
  {
    MyClass mainObj ("M");
    fa ();
    cout << "Mission accomplished!\n";
  }
  catch (const char *e)
  {
    cout << "exception: " << e << endl;
    cout << "Mission impossible!\n";
  }
  return 0;
}

void fa ()
{
  MyClass a ("A");
  fb ();
  cout << "return from fa()\n";
  return;
}

void fb ()
{
  MyClass b ("B");
  fc ();
  cout << "return from fb()\n";
  return;
}

void fc ()
{
  MyClass c ("C");
  fd ();
```

```
  cout << "return from fc()\n";
  return;

}

void fd ()
{
  MyClass d ("D");
  // throw "in fd(), something weird happened.";
  cout << "return from fd()\n";
  return;
}
```

Output:

```
return from fd()
Destroying D
return from fc()
Destroying C
return from fb()
Destroying B
return from fa()
Destroying A
Mission accomplished!
Destroying M
```

EXCEPTION OBJECT

The exception object holds the error information about the exception that had occurred. The information includes the type errors i.e. logic errors or run time error and state of the program when the error occurred.

An exception object is created as soon as exception occurs and it is passed to the corresponding catch block as a parameter. The catch block contains the code to catch the occurred exception.

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error.

General Form:

```
try
{
Throw exception object;
}
catch(Exception &exceptionobject)
{
```

…
}

When a throw expression is evaluated, an exception object is initialized from the value of the expression. The exception object which is thrown gets its type from the static type of the throw expression.

Inside a catch block, the name initialized with the caught exception object is initialized with this exception object

The exception object is available only in catch block. You cannot use the exception object outside the catch block.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
class MyException
{
public:
  char str_what[80];
  int what;
    MyException ()
  {
    *str_what = 0;
    what = 0;
  }
  MyException (char *s, int e)
  {
    strcpy (str_what, s);
    what = e;
  }
};

int main ()
{
  int i;
  try
  {
    cout << "Enter a positive number: ";
    cin >> i;
    if (i < 0)
      throw MyException ("Not Positive", i);
  }
  catch (MyException e)
  {                             // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
```

```
  }
  return 0;
}
```

Output:

```
Enter a positive number: -1
Not Positive: -1
```