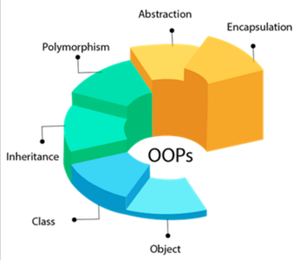
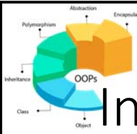


Exception Handling

Chapter 9

Object Oriented Programming

By DSBaral



Introduction

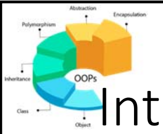
- The software systems must be developed to make them fault tolerant by handling possible error condition.
- These potential error conditions can be detected during the runtime and corrected, if possible, to prevent the system from software crash.
- The errors that occur at run time are called *exceptions*.
- They are caused by a wide variety of exceptional circumstance, such as running out of memory, not being able to open a file, trying to initialize an object to an impossible value, or using an out-of-bounds index to a vector.
- Detecting abnormal behavior of the software at the runtime and taking preventive measure is called *exception handling*.

Introduction (Contd…)

- Exception handling provides a way of transferring control and information to function caller that has expressed willingness to handle exceptions of a given type when the function itself cannot cope with the problem.
- The function that cannot handle problem *throws* the exception to its caller to handle it.
- A function that wants to handle that kind of problem can indicate that it is willing to *catch* that exception.
- Exceptions of any types can be “thrown and caught” and the set of exceptions a function may throw can be specified.
- Exception handling can be used to support ideas of error handling and fault tolerant computing.

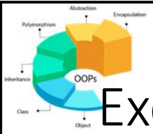
Introduction (Contd…)

- Traditionally, error handling was done by adding a piece of code in the program module for validating input data and prevent runtime errors in the module where error occurs.
  - If the user is using a library in which error handling is performed in this way, the user of such library has no ways to detect the errors and handle them in their code, even though they knew how to handle such errors.
- The exception handling technique is developed to deal with such problems.
- The error handling mechanism in C++ is called exception handling.
  - Exception handling allows us to take corrective measure before termination of the program or leave the program for termination.



Introduction (Contd…)

- There are two types of exceptions that can occur during runtime. They are
  - a) Synchronous exception
    - The exceptions which occur due to problem in input data or inappropriate way of handling data are called synchronous exceptions.
    - Some of the examples of synchronous exceptions are index out of range, overflow, underflow, not being able to open a file, I/O errors etc.
  - a) Asynchronous exception
    - The exceptions caused by events or faults unrelated to the program and beyond the control of program are called asynchronous exceptions.
    - Some of the examples of asynchronous exceptions are keyboard interrupt, hardware malfunction, disk failure etc.
- The exception handling mechanism in C++ allows us to handle only synchronous exceptions.



Exception Handling Construct

- When the program encounters abnormal situation, the program control can be transferred to another part of the program that is designed to handle the problem.
- The program control can be transferred to other location by throwing an exception.
- The other part that is designed to handle the exception catches the exception if its type matches with the thrown exception.
- The exception handling mechanism uses three keyword `try`, `throw` and `catch`.
- The part of the code that can generate the exception should be placed within the `try` block and the part of the code to handle appropriate exception should be placed within the `catch` block.

Exception Handling Construct (Contd…)

- The `try`, `throw` and `catch` construct has the following form

```
try
{
   //block that can raise exception as-> throw obj;
}
catch(type1)
{
   //Handle type1 exception
}
catch(type2)
{
   //Handle type2 exception
}
```

Exception Handling Construct (Contd…)

- The following figure shows the relationship between the `try`, `throw` and `catch` constructs in C++.

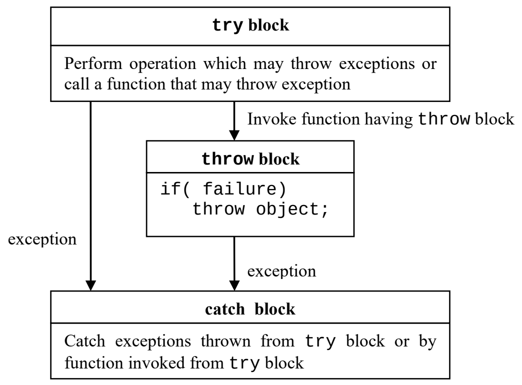

try block
Perform operation which may throw exceptions or call a function that may throw exception
Invoke function having throw block
throw block
if(failure)
throw object;
exception
exception
catch block
Catch exceptions thrown from try block or by function invoked from try block

Figure: Exception handling construct in C++

Exception Handling Construct (Contd…)

- The `try` block must immediately be followed by a `catch` block (handler).
- There can be multiple catch blocks in `try-catch` construct and each catch block must immediately be followed by previous `catch` block to handle exceptions of different types.
- If the exception is thrown in the `try` block, the program control is transferred to the appropriate exception handler that matches with the exception type.
- The program should try to handle (catch) every exceptions that are thrown.
  - If the thrown exceptions are not handled, the program terminates abnormally.
- It is better to throw exception as objects, however, basic data types can also be thrown. (With exception objects more information can be passed)

Advantage of Exception Handling over Conventional Error Handling

- When error is detected, the error could be handled locally or not locally.
- If the error is handled locally in a library then the error message may refer to library concept completely unknown to user such as “bad argument to asin()”.
- Even if the library is not generating error message, the error handled locally is not visible to the user.
- The user of the library should be aware of the error so that necessary measures could be taken.

Advantage of Exception Handling over Conventional Error Handling (Contd…)

- Traditionally, when the error is not handled locally the function could
  1. terminate the program
  2. return a value that indicates error
  3. return some value and set the program in illegal state
- All the traditional techniques have at least one problems
- The exception handling mechanism provides alternatives to these traditional techniques when they are unclear, insufficient and error prone.
- Exception handling separates the error handling code from the other code making the program more readable.
- Exception handling mechanism provides a more refined method when different program segments are written by different people.

Advantage of Exception Handling over Conventional Error Handling (Contd…)

- If the exception is not handled then the program terminates.
  - This is somewhat similar with one of traditional approach of error handling.
  - Programmer can decide whether to handle or not.
- Exception provides a way for code that detects a problem from which it cannot recover to the part of the code that might take necessary measure.
- The exception handling mechanism in C++ is designed to support the object oriented style of programming.
- The exception handling mechanism passes information about the error from the location where error occurs to the location where the error is handled.

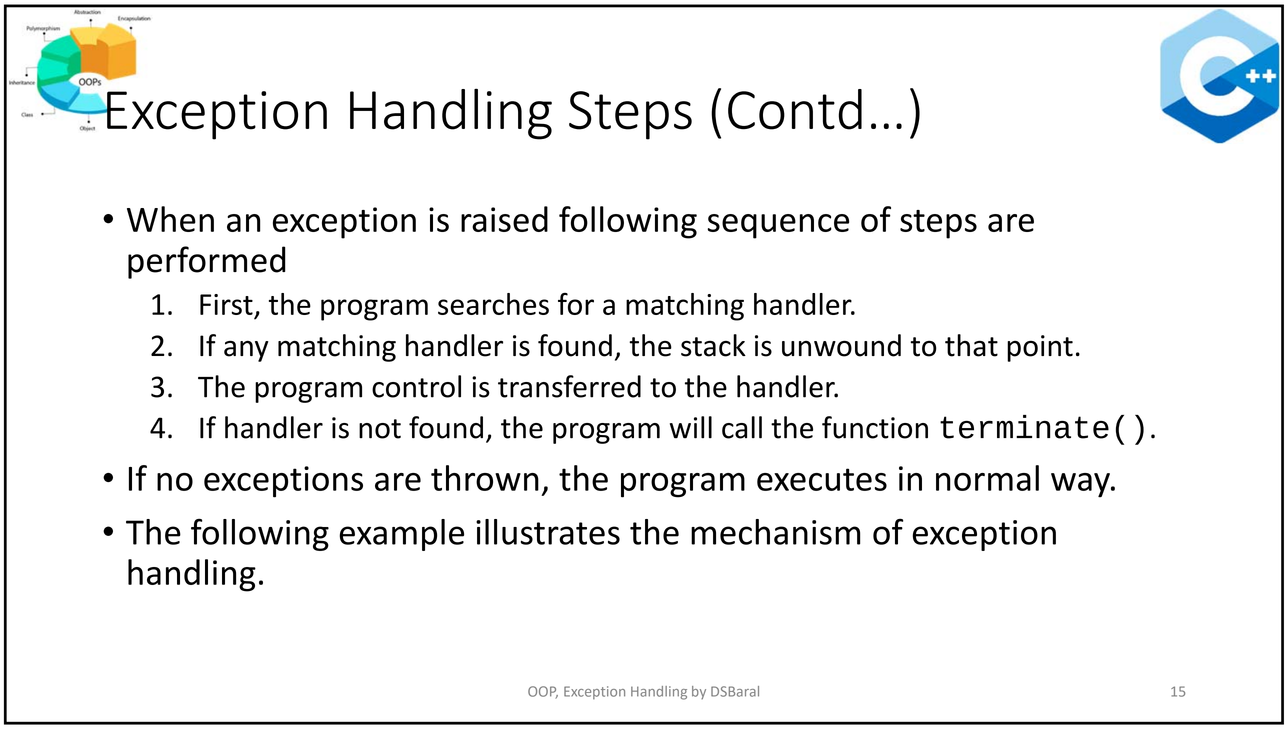
Exception Handling Steps

- To write the code that handles exception, the code should perform the following task.
  1. Hit the exception (detect the problem causing exception)
  2. Throw the exception (inform that an error has occurred)
  3. Catch the exception (receive the error information)
  4. Handle the exception (take appropriate actions)
- A typical exception handling code can have the following pattern.

```
function1()
{
   //......
   if(operation1_fails)
        throw obj1; //obj1 is of type1
}
```

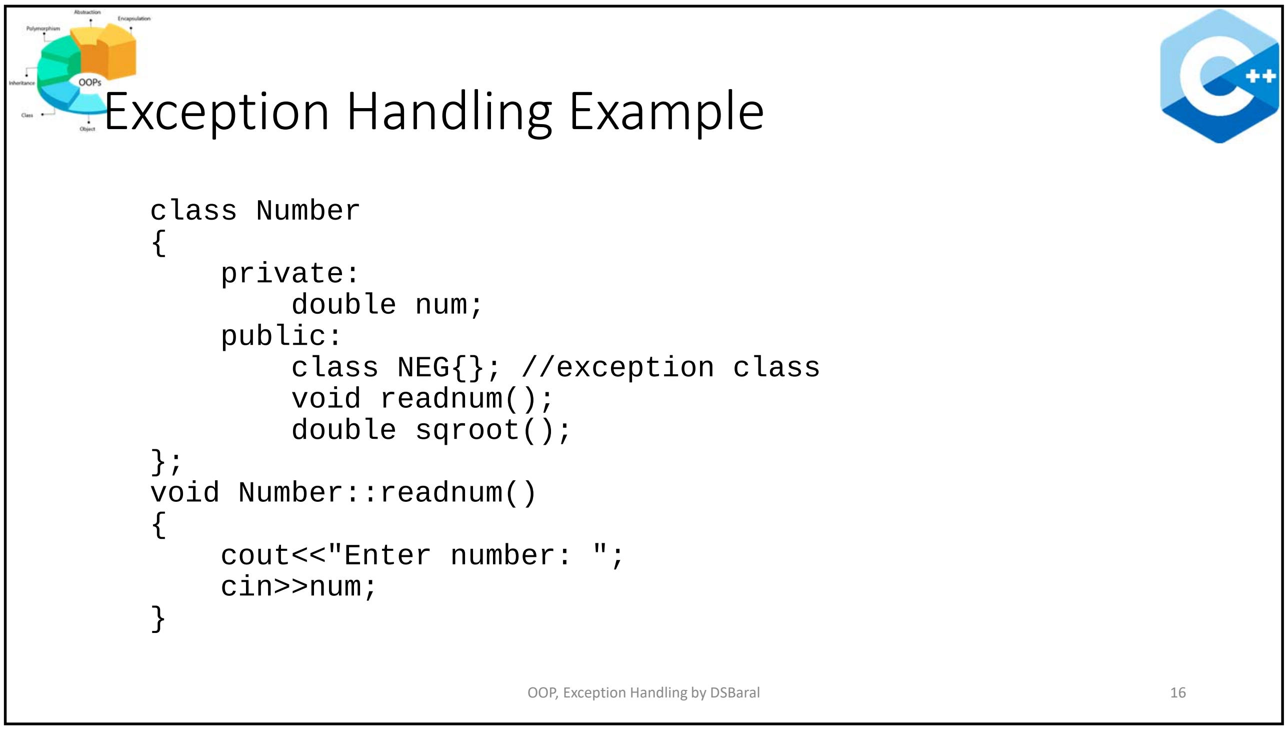
Exception Handling Steps (Contd…)

```
function2(){
   try {
         //......
         function1();
         //......
         if(operation2_fails)
               throw obj2; //obj2 is of type2
   }
   catch(type1 ob1){
         //handle appropriate actions
   }
   catch(type2 ob2)  {
         //handle appropriate actions
   }
   //......
}
```



Exception Handling Steps (Contd…)

- When an exception is raised following sequence of steps are performed
  1. First, the program searches for a matching handler.
  2. If any matching handler is found, the stack is unwound to that point.
  3. The program control is transferred to the handler.
  4. If handler is not found, the program will call the function `terminate()`.
- If no exceptions are thrown, the program executes in normal way.
- The following example illustrates the mechanism of exception handling.



Exception Handling Example

```
class Number
{
    private:
        double num;
    public:
        class NEG{}; //exception class
        void readnum();
        double sqroot();
};
void Number::readnum()
{
    cout<<"Enter number: ";
    cin>>num;
}
```

Exception Handling Example (Contd…)

```
double Number::sqroot()
{
    if(num<0)
        throw NEG();
    else
        return (sqrt(num));
}
int main()
{
    Number n1;
    double res;
    n1.readnum();
```

Exception Handling Example (Contd…)

```
    try
    {
        cout<<"\n Trying to find square root ...";
        res=n1.sqroot();
        cout<<"\nSquare root is: "<<res<<endl;
        cout<<"Success... Exception is not raised"<<endl;
    }
    catch(Number::NEG)
    {
        cout<<"\nSquare root of negative number"
              " not possible!"<<endl;
    }
    return 0;
}
```

Rethrowing Exception

- If we do not completely handle the exception that is caught, we can rethrow that exception by writing `throw` without argument.
- In that case the handler typically does what can be done locally (handle partially) and then throws the exception again (pass it to the caller).
- The throw construct without an explicit parameter raises the previous exception again (rethrown).
- When rethrowing same exception again, the function `terminate()` is called if an exception does not exist currently.
- An exception can only be rethrown from within catch block or from any function called from within catch block.

Rethrowing Exception (Contd…)

- The pattern of throwing same exception again may be as follows

```
function1(){
  //......
  if(operation1_fails)
      throw obj1; //obj1 is of type1
}
function2(){
  try{
      //......
      function1();
      //......
      if(operation2_fails)
          throw obj2; //obj2 is of type2
  }
```

Rethrowing Exception (Contd…)

```
    catch(type1 ob1)
    {
         //do what can be done here for exception of type1
         //and rethrow
         throw;  //rethrow type1 exception again
    }
    catch(type2 ob2)
    {
         //handle type2 exception completely
    }
    //......
}
```

Rethrowing Exception (Contd…)

```
function3()
{
    //......
    try
    {
         function2();
    }
    catch(type1 ob1)
    {
         //handle the exception of type1
         //......
    }
}
```

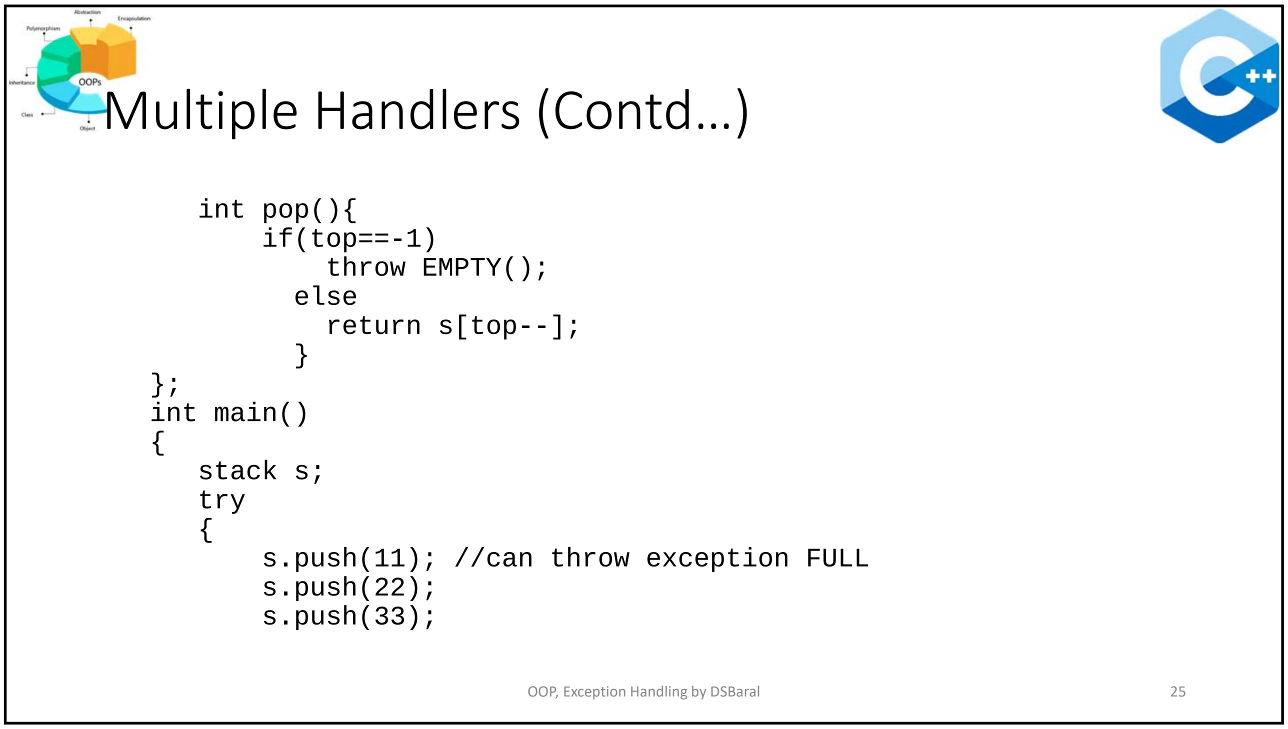
- Write a complete program for rethrowing.

Multiple Handlers

- In a program, there is a chance of occurrence of more than one error during runtime.
- In such a case, C++ permits to design functions to throw as many exceptions as needed.
- The implementation of stack class can be the best example of illustrating the need of multiple exceptions.
- The program has to throw separate exceptions while attempting to push data when the stack is full and while attempting to pop data from empty stack.

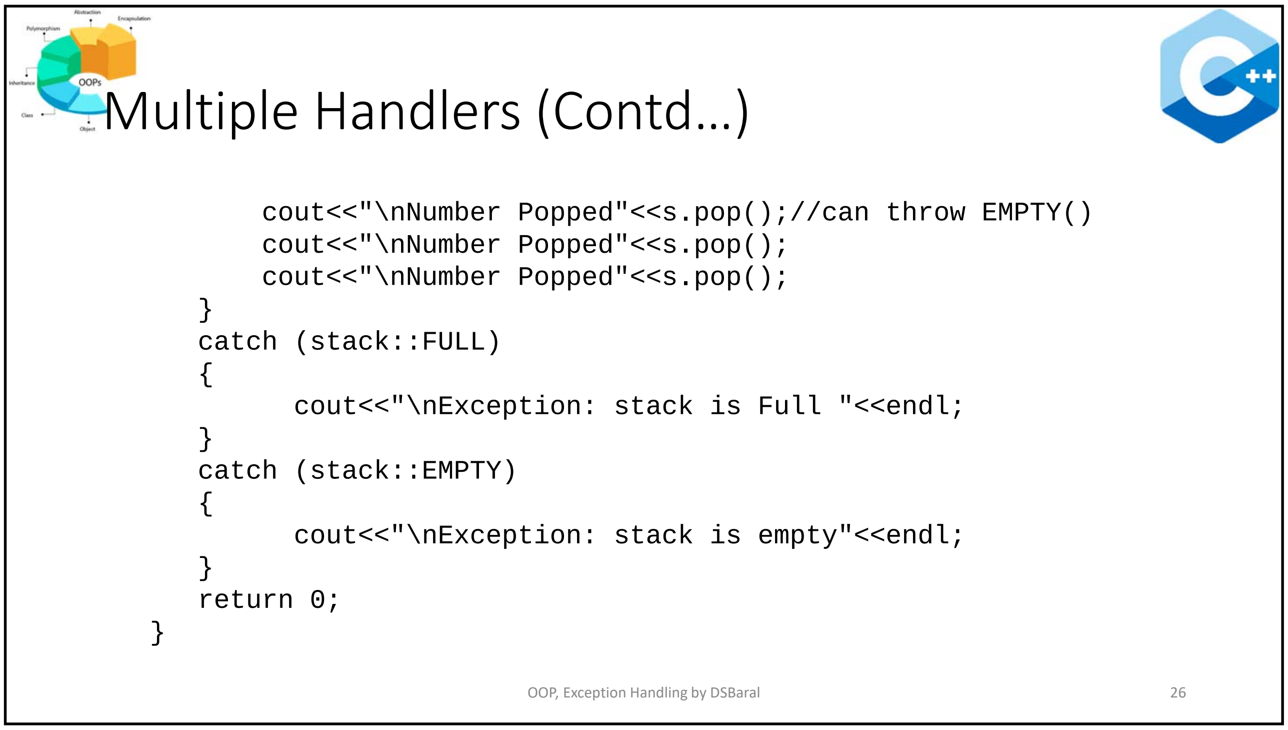
Multiple Handlers (Contd…)

```
const int MAX=25;
class stack
{
protected:
   int s[MAX];
   int top;
public:
   class FULL{}; //exception class for full stack
   class EMPTY{}; //exception class for empty stack
   stack() { top=-1;}   //constructor
   void push(int x){
       if(top==MAX-1)   //if stack full
           throw FULL(); // throw exception
       else
           s[++top]=x;
   }
```



Multiple Handlers (Contd…)

```
      int pop(){
          if(top==-1)
              throw EMPTY();
            else
              return s[top--];
            }
   };
   int main()
   {
      stack s;
      try
      {
          s.push(11); //can throw exception FULL
          s.push(22);
          s.push(33);
```



Multiple Handlers (Contd…)

```
          cout<<"\nNumber Popped"<<s.pop();//can throw EMPTY()
          cout<<"\nNumber Popped"<<s.pop();
          cout<<"\nNumber Popped"<<s.pop();
      }
      catch (stack::FULL)
      {
            cout<<"\nException: stack is Full "<<endl;
      }
      catch (stack::EMPTY)
      {
            cout<<"\nException: stack is empty"<<endl;
      }
      return 0;
   }
```

Catching All Exceptions

- Sometimes it would be useful to have a handler that handles the exception without knowing the type or a handler that handles any type of exception.
- There is a provision in C++ for catching all exceptions raised in the try block without knowing the type.
- In order to catch all the exceptions the syntax of the catch construct is as follows:

```
catch(...)
{
   //handle exceptions here
}
```

Catching All Exceptions (Contd…)

- The “catch all” exception handler is to be kept at the last catch block if multiple catch block follows the try block as

```
try{
   //code that raise exception
}
catch(type1 arg){
   //handle type1
}
catch(...){
   //handle rest
}
```

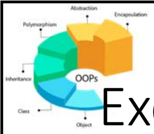
- If the catch all block is placed at first before other catch blocks, rest of the catch blocks are not executed in any case.

Exception with Arguments

- Sometimes, application may demand more information about the cause of exception.
- When exception is thrown, the control is transferred to the handler along with object as arguments to the handlers from the throw point.
- If we have to throw objects with information, the exception classes are to be declared with members to provide detail information about the error from the throw point.
- In the catch block, instead of just mentioning type to catch, we can mention type and object parameter which gets information of the thrown object as in function call.
- The following example illustrates this concept

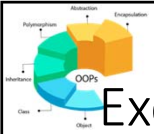
Exception with Arguments (Contd...)

```
class myException
{
public:
   int number;
   char description[20];
   myException(){
         number=0;
         description[0]='\0';
   }
   myException(int n,char *desc){
         number=n;
         strcpy(description,desc);
   }
};
```



Exception with Arguments (Contd…)

- This exception class can be used to throw object with information of the cause of the error as
```
double sqroot(double n){
   if(n<0)
      throw myException(1,"Error: negative number");
   else
      return sqrt(n);
}
```
- The `try-catch` construct is written as
```
try{
   double srt=sqroot(dn);
   cout<<"Square root= "<<srt;
}
```



Exception with Arguments (Contd…)

```
   catch(myException e)
   {
     cout<<"Error occurred in performing square root";
     cout<<"\nError number: "<<e.number;
     cout<<"\nError description: "<<e.description;
   }
   //......
```
- So, this method helps the programmer to know the cause of the exception (what bad value caused the exception).
- If the same exception is thrown from different function, it would be easy to know the actual cause of the exception
- The thrown exception object is caught as according to its type in the catch block and the value of the object provides the detail information

Exceptions Specification for Function

- We can specify the list of exception that could be thrown from the function.
- In C++, there is a provision of specifying a list of exceptions as a part of the function definition.
- The syntax for declaring list of exceptions for function is as follows:

```
return_type function_name(parameter_list) throw(type1,type2,...)
{
   //body of function
}
```

- This specification of list of exceptions is to mean that, this function may throw the exceptions specified in the exception list. For example

```
void func(int num)throw(x,y); //can only throw x and y exception
```

Exceptions Specification for Function (Contd…)

- When specifying the exception list with function, the function guarantees to its caller what exceptions it might throw.
- If the function, directly or indirectly, throws exception not mentioned in the list, the standard library function `unexpected()` is called.
- The default meaning of `unexpected()` is to call the other standard library function `terminate()`, which by default calls `abort()`.
- The function `abort()` terminates the program abnormally, however, we can specify our own unexpected handler if necessary.

Exceptions Specification for Function (Contd…)

- A function that does not specify the exception list can throw any exception. For example
```
void func1();//can throw any exception
```
- A function that does not throw any exception is declared with empty exception list as
```
void func2() throw();//cannot throw any exception
```
- Specific exception that is thrown by the function is specified as
```
void func3() throw(x);//throws only x exception and
                    //exception derived from x
void func4() throw(x,y);//throws only x and y exception
                   //and exception derived from x and y
```

Handling Uncaught Exceptions

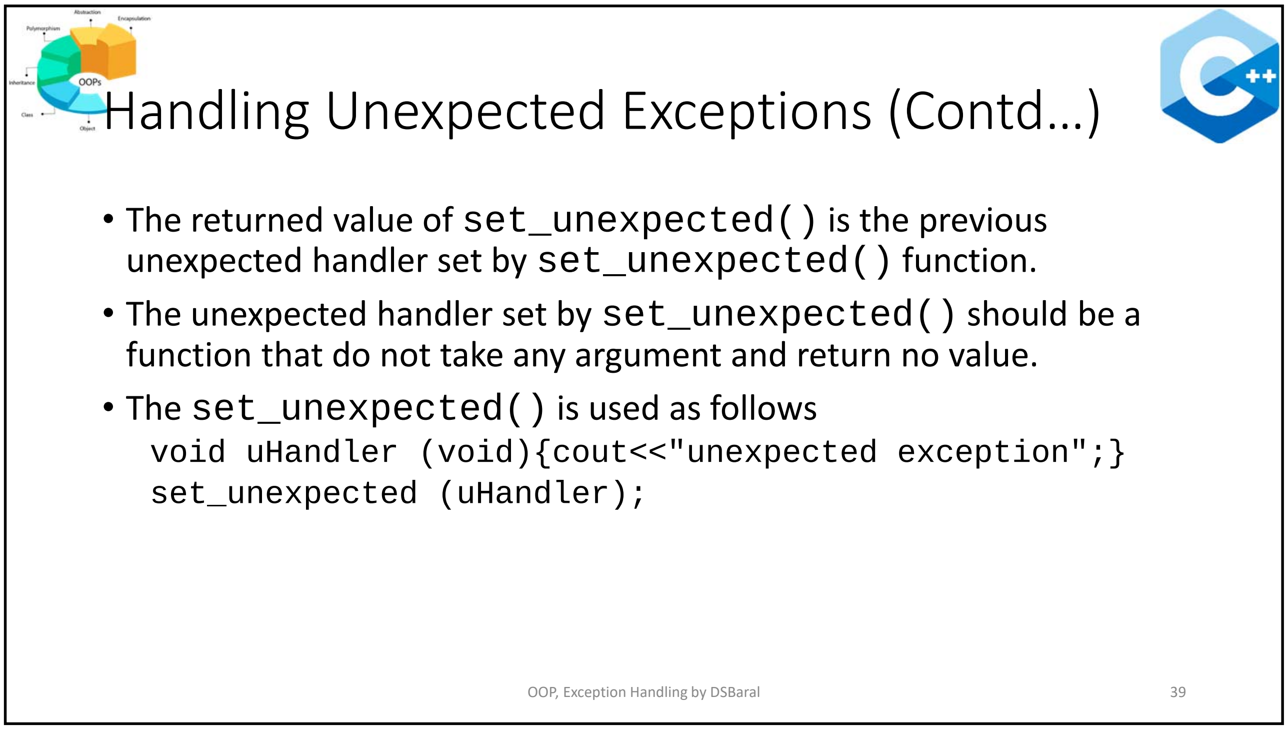
- When the exception is thrown but the handler is not found, the standard library function `terminate()` is called.
- The `terminate()` function is also called when the exception handling mechanism finds a corrupted stack, or a destructor called during stack unwinding raises exception and not handled.
- The function `terminate()` by default calls `abort()` which in turn terminates the program, indicating the system that program is terminated abnormally.
- The default behavior of `terminate()` can be changed to call a different function when handler for thrown exception is not found.
- The standard library function `set_terminate()` changes the function to be called by `terminate()` when thrown exception is not handled.

Handling Uncaught Exceptions (Contd...)

- By using `set_terminate()` function, the programmer can define the program's action to be taken to terminate the program when handler for exception cannot be found.
- The `set_terminate()` function is declared in `<exception>` as
  ```
  terminate_handler set_terminate(terminate_handler);
  ```
- The argument of the `set_terminate()` is the new terminate handler function to be called by `terminate()`.
- The return value of `set_terminate()` is the previous terminate handler set by the `set_terminate()` function.
- The terminate handler set by `set_terminate()` should be a function that takes no argument and return no value. For example
  ```
  void tHandler (void){cout<<"uncaught exception";}
  set_terminate (tHandler);
  ```

Handling Unexpected Exceptions

- When the function raises exception that is not in the exception list, the standard library function `unexpected()` is called.
- By default `unexpected()` calls another standard library `terminate()` which in turn normally calls `abort()`.
- The default behavior of function `unexpected()` can be modified to call some other unexpected handler function instead of calling `terminate()` by using the function `set_unexpected()`.
- The function `set_unexpected()` is declared in `<exception>` as
  ```
  unexpected_handler set_unexpected(unexpected_handler);
  ```



Handling Unexpected Exceptions (Contd…)

- The returned value of `set_unexpected()` is the previous unexpected handler set by `set_unexpected()` function.
- The unexpected handler set by `set_unexpected()` should be a function that do not take any argument and return no value.
- The `set_unexpected()` is used as follows

```
void uHandler (void){cout<<"unexpected exception";}
set_unexpected (uHandler);
```