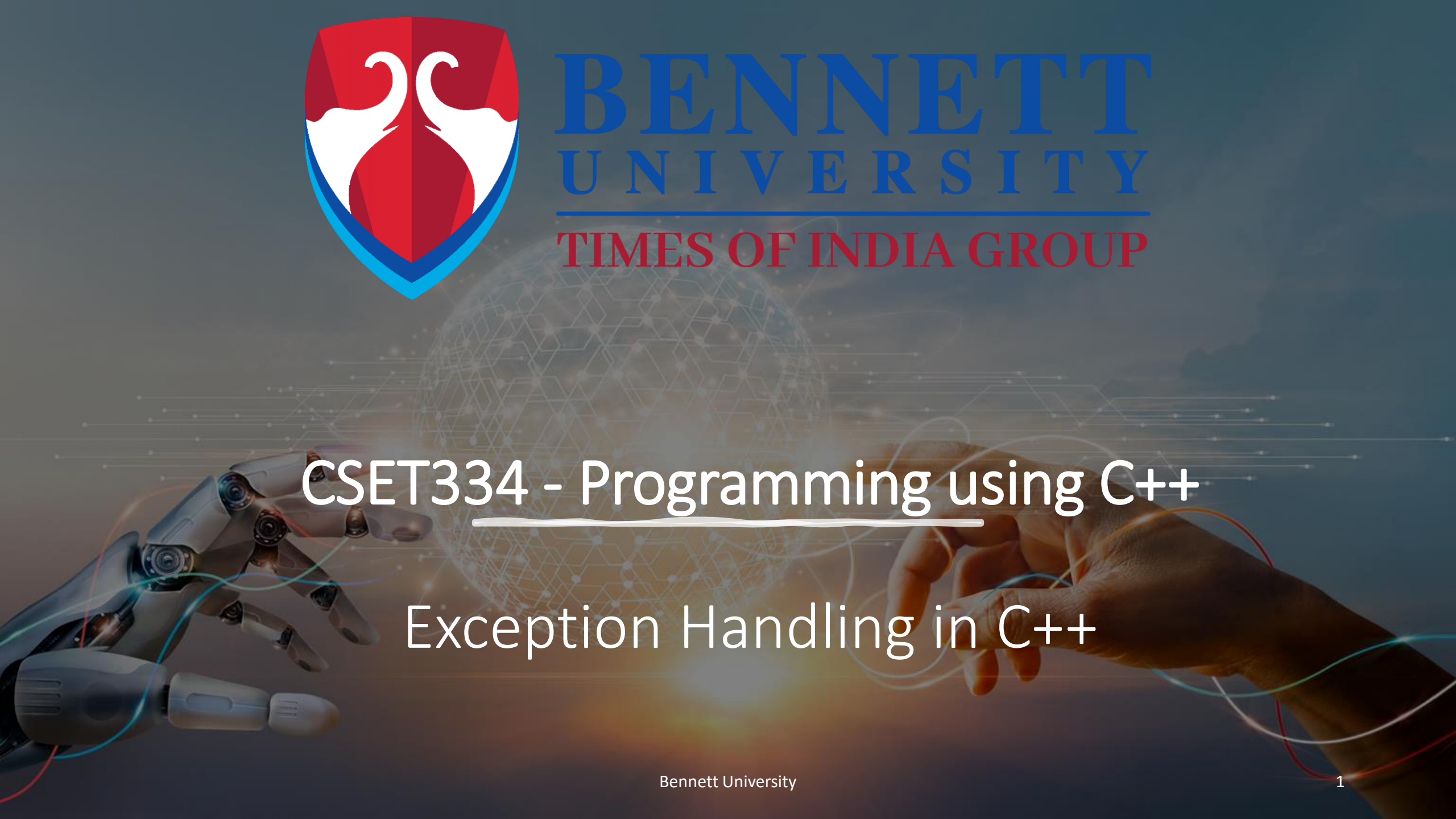


BENNETT UNIVERSITY

TIMES OF INDIA GROUP

CSET334 - Programming using C++

Exception Handling in C++

Course Specific Learning Outcomes

No.	Unit Learning Outcome
CO1	To explain the fundamental programming concepts and methodologies to building C++ programs.
CO2	To implement various OOPs concepts including memory allocation/deallocation procedures and member functions.

Introduction

- Exceptions
  - Exception refers to unexpected condition in a program. The unusual conditions could be faults,
  - Causing an error which in turn causes the program to fail.
  - Occur infrequently
- Types of Exceptions
  - Synchronous
  - Asynchronous

Types of Exceptions

1. Synchronous
   - The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.

   Example:
   - Errors such as out of range, overflow, underflow and so on.

2. Asynchronous
   - The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.

   Example:
   - Errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

Exception Handling

- The error handling mechanism of c++ is generally referred to as exception handling.
- Can resolve exceptions
  - Allow a program to continue executing or
  - Notify the user of the problem and
  - Terminate the program in a controlled manner
- Makes programs robust and fault-tolerant

Traditional Exception Handling

- Intermixing program and error-handling logic
  - Pseudocode outline

    Perform a task
    If the preceding task did not execute correctly
      Perform error processing
    Perform next task
    If the preceding task did not execute correctly
      Perform error processing
    …
- Makes the program difficult to read, modify, maintain and debug
- Impacts performance

Note:– In most large systems, code to handle errors and exceptions represents >80% of the total code of the system

Fundamental Philosophy

- Remove error-handling code from the program execution's "main line"
- Programmers can handle any exceptions they choose
  - All exceptions
  - All exceptions of a certain type
  - All exceptions of a group of related types

Fundamental Philosophy (continued)

- Programs can
  - Recover from exceptions
  - Hide exceptions
  - Pass exceptions up the “chain of command”
  - Ignore certain exceptions and let someone else handle them

Fundamental Philosophy (continued)

- An *exception* is a class
  - Usually derived from one of the system's exception base classes
- If an exceptional or error situation occurs, program *throws* an object of that class
  - Object crawls up the call stack
- A calling program can choose to *catch* exceptions of certain classes
  - Take action based on the exception object

Example: Handling an Attempt to Divide by Zero

BENNETT UNIVERSITY
TIMES OF INDIA GROUP

```
1  // Fig. 27.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header file contains runtime_error
4  using std::runtime_error; // standard C++ library class runtime_error
5
6  // DivideByZeroException objects should be thrown by functions
7  // upon detecting division-by-zero exceptions
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // constructor specifies default error message
12    DivideByZeroException::DivideByZeroException()
13       : runtime_error( "attempted to divide by zero" ) {}
14 }; // end class DivideByZeroException
```

Zero Divide Example

BENNETT UNIVERSITY
TIMES OF INDIA GROUP

```
1  // Fig. 27.2: Fig27_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  #include "DivideByZeroException.h" // DivideByZeroException class
10
11 // perform division and throw DivideByZeroException object if
12 // divide-by-zero exception occurs
13 double quotient( int numerator, int denominator )
14 {
15    // throw DivideByZeroException if trying to divide by zero
16    if ( denominator == 0 )
17       throw DivideByZeroException(); // terminate function
18
19    // return division result
20    return static_cast< double >( numerator ) / denominator;
21 } // end function quotient
22
23 int main()
24 {
25    int number1; // user-specified numerator
26    int number2; // user-specified denominator
27    double result; // result of division
28
29    cout << "Enter two integers (end-of-file to end): ";
```

Zero Divide Example

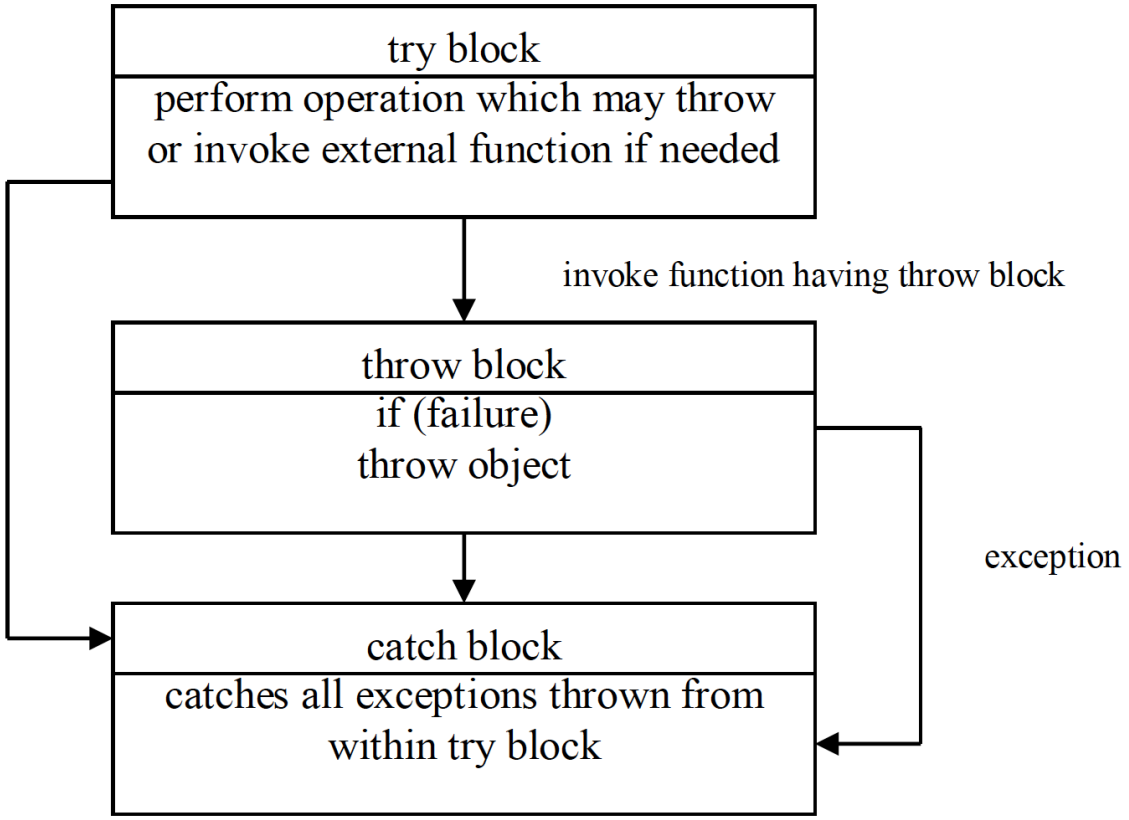
```
30
31    // enable user to enter two integers to divide
32    while ( cin >> number1 >> number2 )
33    {
34       // try block contains code that might throw exception
35       // and code that should not execute if an exception occurs
36       try
37       {
38          result = quotient( number1, number2 );
39          cout << "The quotient is: " << result << endl;
40       } // end try
41
42       // exception handler handles a divide-by-zero exception
43       catch ( DivideByZeroException &divideByZeroException )
44       {
45          cout << "Exception occurred: "
46             << divideByZeroException.what() << endl;
47       } // end catch
48
49       cout << "\nEnter two integers (end-of-file to end): ";
50    } // end while
51
52    cout << endl;
53    return 0; // terminate normally
54 } // end main
```

Exception Handling Model

- When a program encounters an abnormal situation for which it in not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception.
- The exception handling mechanism uses three blocks: try, throw, and catch.
- The try block must be followed immediately by a handler, which is a catch block.
- If an exception is thrown in the try block the program control is transferred to the appropriate exception handler.
- The program should attempt to catch any exception that is thrown by any function.

Exception Handling Model (continued)

The relationship of these three exceptions handling constructs called the exception handling model is shown as:



Throwing an Exception

- Use keyword **throw** followed by an operand representing the type of exception
  - The **throw** operand can be of any type
  - If the **throw** operand is an object, it is called an **exception** object
- The **throw** operand initializes the exception parameter in the matching **catch** handler, if one is found.
- The throw expression initialize a temporary object of the typeT used in throw (T arg).
- **Syntax:**

```
throw T;
```

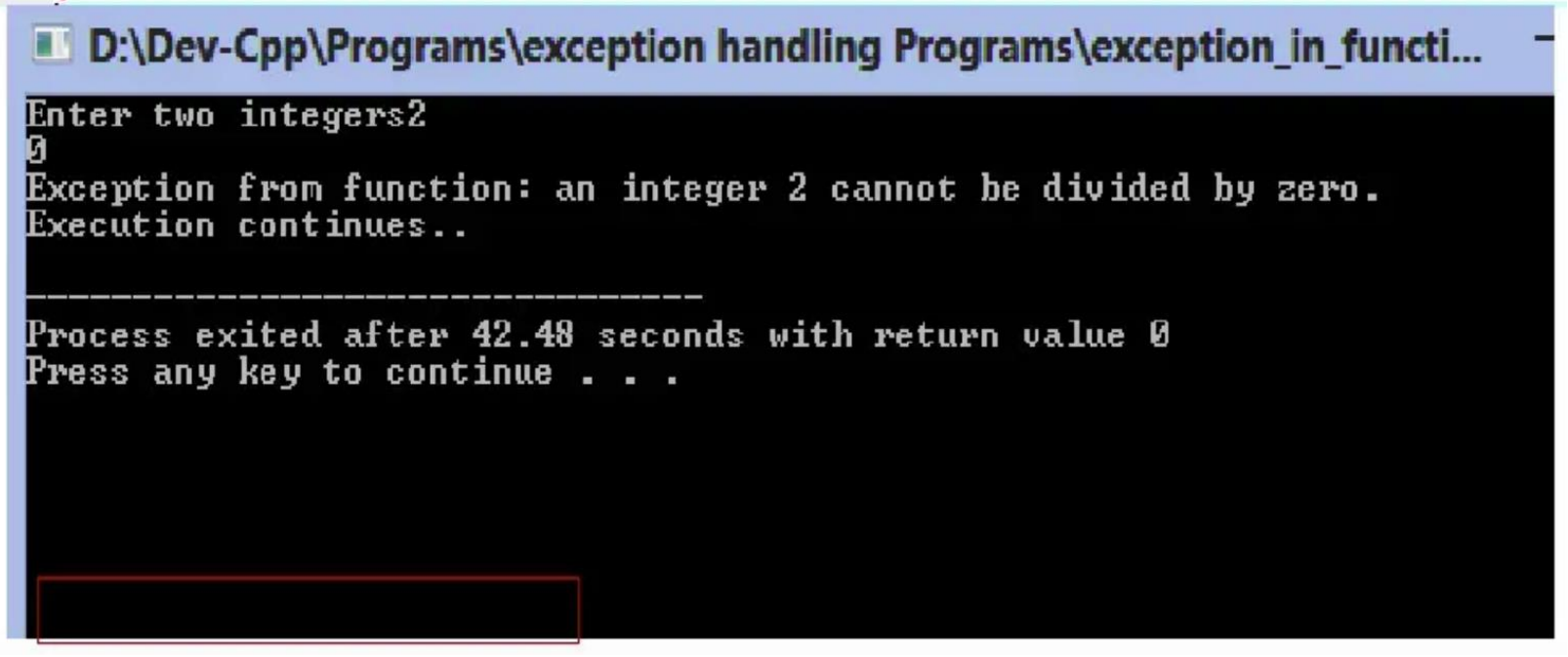
Throwing an Exception (continued)

```
1   #include<iostream>
2   using namespace std;
3   int division(int num1,int num2)    //division function
4   {
5       if(num2==0)
6       throw num1;                    // throw exception
7
8       return num1/num2;
9   }
10  int main()
11  {
12      //Read two integers
13      cout<<"Enter two integers";
14      int num1,num2;
15      cin>>num1>>num2;
16      try                                    //try block
17      {
18          int result=division(num1,num2);  //invoke the function
19          cout<<num1<<"/"<<num2<<" is:"<<result<<endl;
20      }
```

Throwing an Exception (continued)

Cont..

```
21      catch( int e)
22      {
23      cout<<"Exception from function: an integer "<<e<<" cannot be divided by zero."
24      cout<<endl;
25      }
26      cout<<"Execution continues.."<<endl;
27      return 0;
28  }
```



Catch Handlers

- The exception handler is indicated by the catch keyword.
- It must be used immediately after the statements marked by the try keyword.
- The catch handler can also occur immediately after another catch Each handler will only evaluate an exception that matches.
- Keyword **catch**
- **Syntax**:

```
catch(T)
{
// error messages
}
```

Catch Handlers (continued)

```
try {
  // code to try
}
catch (exceptionClass1 &name1) {
  // handle exceptions of exceptionClass1
}
catch (exceptionClass2 &name2) {
  // handle exceptions of exceptionClass2
}
catch (exceptionClass3 &name3) {
  // handle exceptions of exceptionClass3
}
...
/* code to execute if
       no exception or
       catch handler handled exception*/
```

All other classes of exceptions are not handled here

`catch` clauses attempted in order; first match wins!

try Blocks

- The try keyboard defines a boundary within which an exception can occur.
- A block of code in which an exception can occur must be prefixed by the keyword try.
- Following the try keyword is a block of code enclosed by braces.
- This indicates that the prepared to test for the existence of exceptions.
- If an exception occurs, the program flow is interrupted.

try Blocks (continued)

- **Syntax**:

```
try
{
……..
if (failure)
      throw T;
}
catch(T)
{
……….
}
```

```
try
 {        // code
          if ( x )
            throw 10;
          // code
          if (y)
            throw 20;
          //code
 }
```

```
try { // code here }
catch (int param) {
   cout << "int exception";
}
catch (char param) {
   cout << "char exception";
}
catch (...) {
   cout << "default exception";
}
```

When to Use Exception Handling

- To process synchronous errors
  - Occur when a statement executes
- Not to process asynchronous errors
  - Occur in parallel with, and independent of, program execution
- To process problems arising in predefined software elements
  - Such as predefined functions and classes
  - Error handling can be performed by the program code to be customized based on the application's needs

Don't use for routine stuff such as end-of-file or null string checking

Error Note

- The compiler will not generate a compilation error if a function contains a **throw** expression for an exception not listed in the function's exception specification.
- Error occurs only when that function attempts to **throw** that exception at run time.
- To avoid surprises at execution time, carefully check your code to ensure that functions do not **throw** exceptions not listed in their exception specifications

Constructors and Destructors

- Exceptions and constructors
  - Exceptions enable constructors to report errors
    - Unable to return values
  - Exceptions thrown by constructors cause any already-constructed component objects to call their destructors
    - Only those objects that have already been constructed will be destructed
- Exceptions and destructors
  - Destructors are called for all automatic objects in the terminated **try** block when an exception is thrown
    - Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
  - If a destructor invoked by stack unwinding throws an exception, function **terminate** is called

Exceptions and Inheritance

- New exception classes can be defined to inherit from existing exception classes
- A **catch** handler for a particular exception class can also catch exceptions of classes derived from that class
  - Enables **catching** related errors with a concise notation

Exception Handling Summary

- Exceptions are derived from class **exception**
- Exceptional or error condition is indicated by **throwing** an object of that class
  - Created by constructor in **throw** statement
- Calling programs can check for exceptions with **try...catch** construct
- Unified method of handling exceptions
  - Far superior to coding exception handling in long hand
- No performance impact when no exceptions


THANK YOU!