Chapter 10

Objective

- Error Handling Techniques
- Exception Handling
- Advantages Of Using Exception Handling
- Simplest form of Exceptions
- Multiple Exceptions catches
- Matching Exception Type
- Ellipsis catch handler
- Destructor functions
- Throwing an Exception
- C++ Standard Exception Hierarchy Library
- Call to exception class member
- Calls to Exception Handlers
- Misuses of Exception Handling

Unit 1

Error Handling Techniques

- Early stages of C++ had no built-in facility for handling runtime errors.
- Traditional C methods were used for error checking purpose
    1. Return a status code with agreed-upon values to indicate either success or failure.
    2. Assign an error code to a global variable and have other functions examine it.
    3. Terminate the program altogether.

- C methods have significant drawbacks and limitations in an object-oriented environment.

Unit 1

Exception Handling

- Exceptions are a mechanism for error handling.
- Exception handling is designed for run time error processing so program does not suffer from malfunction.
- Purpose is to attempt to execute code and handle unexpected *exceptional* conditions.
- Exception handling improves the readability and maintainability of programs.
- C++ Exception Handling is centered around three keywords: **try**, **catch**, and **throw**.
- Any function that throws an exception must explicitly state the exception mechanism.
- Handle the exception in a function where it has occurred rather than passing it.
- Exception handling is a way to return control from a function or from an exiting block of code.
- Unknown throw will terminate the program by calling “**terminate()”** function.
- Exception handling should be used when system can recover from an error:
  - ❑ Attempt to divide by zero.
  - ❑ Out-of-bounds array subscript.
  - ❑ Arithmetic overflow. (Out of range values)
  - ❑ Memory exhaustion.

- Exception handling frees the programmer from writing tedious code that checks the success status of every function call.
- Exception handling is a very powerful and flexible tool for handling runtime errors effectively

Unit 1

Using Exception Handling

Advantages Of Using Exception Handling

- Exceptions can significantly increase applications robustness.
- With exceptions, error conditions can be recognized by application as well as with system.

Disadvantages of Not Using Exception Handling

- Devastating effects on general public.
- Mission-critical programs can damage or destroy things that are important for human race.

Unit 2

Simplest form of Exceptions

- Set up a try/catch block to catch an exception.
- Skip try block code if exception happens.
- Skip catch block(s) if exception does not happen.

```
try {

}
catch(ExceptionType e) {

}
```

Unit 2

Simplest form of Exceptions

```
#include <iostream>    // Example 9-1

using namespace std;


int main()
{
    try {
        throw 5;
    }
     catch (int num) {
         cout << "Exception for Number " << num << " occurred\n";
     }

     return 0;
 }

```

Unit 2

Matching Exception Type

- Type of an exception determines which handler can catch the exception.
- Type matching rules for exceptions are strict.

```
#include <iostream>    // Example 9-2

using namespace std;

int main()
{
    try {
        throw int();
    }
    catch (unsigned int) { // exception is not caught
        //will not catch the exception from the previous try-block
    }

     return 0;
 }
```

Unit 3

Multiple Exceptions catches

```
try {

}
catch (int n) {

}
catch (char *buf) {

}
```

Unit 3

```
#include <iostream>    // Example 9-3
using namespace std;
int main () // exceptions: multiple catch blocks
{
  try {
    char *mystring = new char [10];
     if (mystring == NULL) throw "Allocation failure";
       for (int n=0; n<=100; n++) {
         if (n > 9) throw n;
         mystring[n]='z';
       } // end of for loop
   } // end try block
   catch (int i) {
     cout << "Exception: ";
     cout << "index " << i << " is out of range" << endl;
   }
   catch (char * str) {
     cout << "Exception: " << str << endl;
   }
   return 0;
 }
```

Unit 4

Ellipsis catch handler

- To catch *all* exceptions use **catch(...)**
- Can't tell what type of exception has occurred because no argument is to reference.
- Ellipsis catch handler must be the last handler for its try block.
- Ellipsis handles following type of exceptions:
  - ❑ C exceptions
  - ❑ System generated and application generated exception.
  - ❑ Memory protection
  - ❑ Divided by zero
  - ❑ Floating point violations

Unit 4

Ellipsis catch handler

```
#include <iostream>    // Example 9-4

using namespace std;

int main()
{
    char *buf;
    int num = 0;

    try {
       buf = new char[512];
         if (buf == 0) throw "Memory allocation failure!";

         if (num == 0) throw num;
     }
     catch(char *str)
     {
         cout << "Excepion raised: " << str << "\n";
     }
```

Unit 4

Ellipsis catch handler

```
  catch(...)    // This must be the last handler for its
  {
    cout << "Handle following type of Exception:\n";
   cout << "C exceptions\nSystem generated ";
    cout << "and Applicatoin generated exceptions.\n";
    cout << "Memory protection, divided by zero, and ";
   cout << "floating point exceptions.\n";
  }

  return 0;
}
```

Unit 5

Destructor Behavior with Exception Handling

- Automatic call to destructor function occurs during stack unwinding for all local objects constructed before the exception was thrown.
- Exception handler does not have to terminate program, but it does terminate the block in which the exception occurred
- if operand is an object, it is called an *exception object*

```
#include <iostream>    // Example 9-5
using namespace std;
class CTest {
    public:
        CTest() { }
        ~CTest() { }
        const char *ShowReason() const { return "Exception in CTest."; }
};

class CDemo {
    public:
        CDemo() { cout << "Constructing CDemo.\n";}
        ~CDemo() { cout << "Destructing CDemo.\n";}
};
```

Unit 5

```
int main()
{
    cout << "In main.\n";

    try {
        cout << "In try block.\n";
        CDemo D;
        cout << "In main(). Throwing CTest exception\n";
        throw CTest();
    }

    catch(CTest& eTest) {
        cout << "In catch handler.\n";
        cout << "Caught CTest exception type: ";
        cout << eTest.ShowReason() << "\n";
    }
```

Unit 5

```

    catch(char *str) {
        cout << "Caught some other exception: ";
        cout  << str << "\n";
    }

        cout << "Back in main. Execution resumes here.\n";

    return 0;
}
```

Output:

```
In main.
In try block.
Constructing CDemo.
In main(). Throwing CTest exception
Destructing CDemo.
In catch handler.
Caught CTest exception type: Exception in CTest class.
Back in main. Execution resumes here.
```

Unit 5

Catching Divided By Zero Exception

```
#include <iostream>    // Example 9-6

using namespace std;

// Class DivideByZeroException to be used in exception
// handling for throwing an exception on a division by zero.

class DivideByZeroException
{
    public:
         DivideByZeroException() :
           message( "attempted to divide by zero" ) { }
         const char *what() const { return message; }

     private:
         const char *message;
 };

```

Unit 5

```
// Definition of function quotient. Demonstrates throwing
// an exception when a divide-by-zero exception is
// encountered.
double quotient( int numerator, int denominator )
{
    if ( denominator == 0 ) throw DivideByZeroException();

    return double( numerator ) / denominator;
}

// Driver program
int main()
{
    int number1, number2;
    double result;

    cout << "Enter two integers (end-of-file to end): ";

```

Unit 5

```
     while ( cin >> number1 >> number2 ) {
         // the try block wraps the code that may throw an exception
         // and the code that should not execute if an exception occurs.

         try {
             result = quotient( number1, number2 );
             cout << "The quotient is: " << result << endl;
         } // end try block

         // Exception is caught if argument type matches with throw type
         catch ( DivideByZeroException &ex ) {
            cout << "Exception occurred: " << ex.what() << '\n';
         }

         cout << "\nEnter two integers (end-of-file to end): ";
     }  // end while loop

     cout << endl;
     return 0;      // terminate normally
}  // end main
```

Unit 6

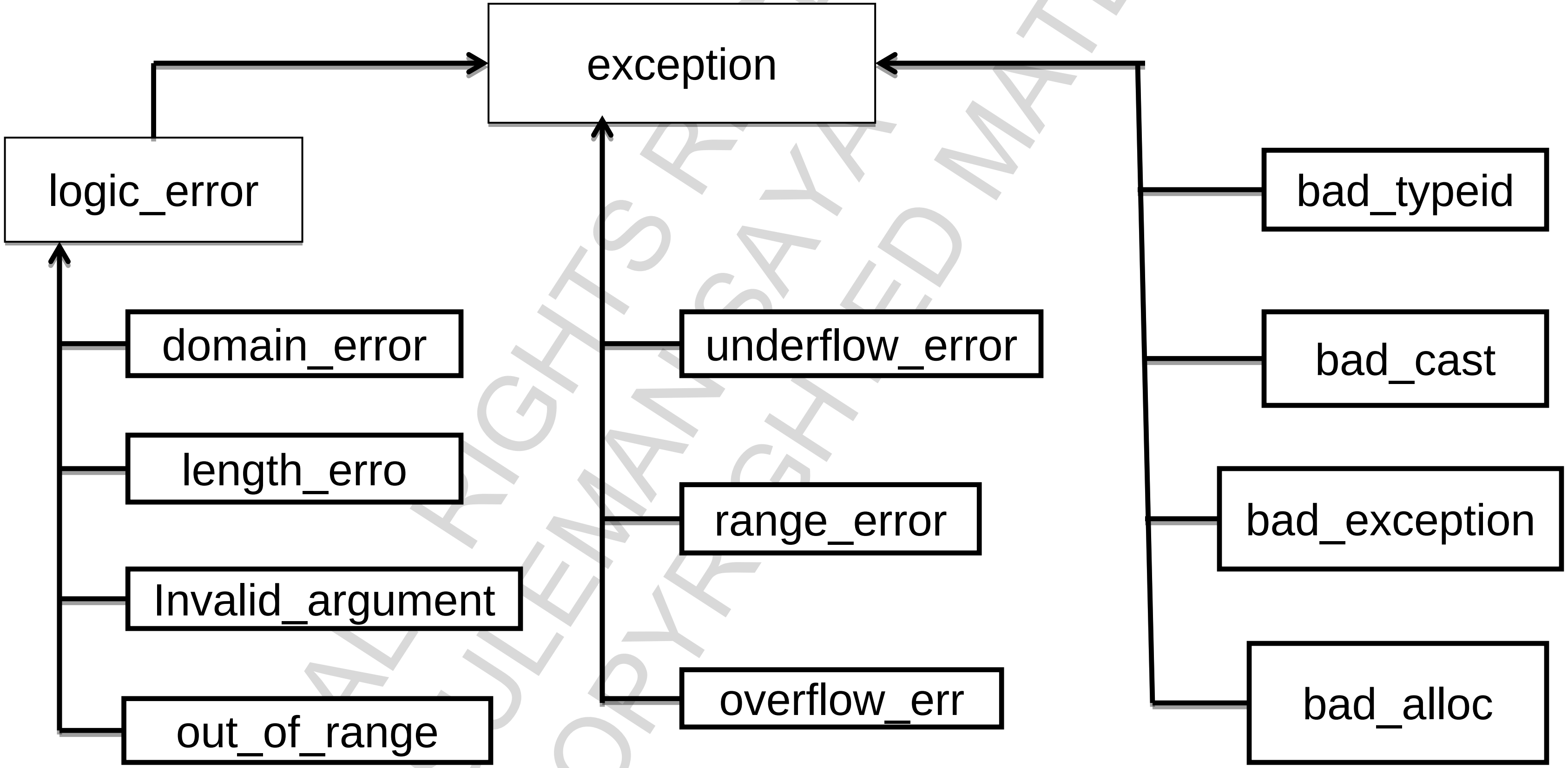
C++ Standard Exception Hierarchy Library

- C++ defines a hierarchy of standard exceptions that are thrown at runtime when abnormal conditions arise.
- The standard exception classes are derived from **std::exception**
- The class **exception** is defined in the <**stdexcept**> header file.
- Use the classes of standard hierarchy of exceptions to throw your exceptions or derive new classes from them.
- Hierarchy enables the application to catch these exceptions in a single catch statement.

```
catch (std::exception& exc)
{
  // Handle exception of type std::exception as well as
  //any exception derived from it
}
```

Unit 6

Exception Hierarchy



Unit 6

Standard Exception Classes

- Standard exceptions that can be thrown by built-in operators of C++ language:
- You must include <**stdexcept**> header file.

EXCEPTION	THROWN BY
bad_alloc	new()
bad_cast	dynamic_cast()
bad_typeid	typeid()
bad_exception	exception specification
out_of_range	at() and [] in bitset
invalid_argument	bitset constructor
overflow_error	to_ulong() in bitset
ios_base::failure	ios_base::clear ()

Unit 7

Call to Exception Class Member Function

- Standard exception class have provided the member function what().
- A what( ) member function issue's error messages when exceptions are thrown.

```
// bad_alloc class example
#include <iostream>    // Example 9-7
#include <stdexcept>
using namespace std;
int main()
{
    try
    {
        char * buff = new char[10000000000000000000];
         strcpy(buff,"Suleman");
         cout << buff << "\n";
         if (buff) delete [] buff;
    }
```

Unit 7

```
// Handlers of derived objects must appear before the handlers of base classes,
// otherwise  corresponding base class will never execute derived class handler.
     catch(bad_alloc& alloc_failure)
     {  // bad_alloc is derived from exception
        // handle exception thrown by operator new
         cout << "memory allocation failure\n";
         cout << alloc_failure.what();
     }
     catch(exception& std_ex)
     {
         Cout << std_ex.what() << endl;
     }
       // exceptions that are not handled elsewhere are caught here
     catch(...)
     {
         cout << "unrecognized exception" << endl;
     }
   return 0;
 }
```

Unit 7

Applying Polymorphism With Exception Handling

- Object of an **exception** class is used for referring to a **bad_typeid** class.

```
#include <iostream>    // Example 9-8
#include <exception>
#include <typeinfo>
using namespace std;
class Fruit { virtual f() {}; };
int main()
 {
     try {
         Fruit *a = NULL;
         typeid (*a); // Standard exceptions to show bad_typeid
     }
     catch (exception& e) {
         cout << "Exception: " << e.what() << "\n";
         // what() msg: Attempted a typeid of NULL pointer
     }
   return 0;
}
```

Unit 8

Misuses of Exception Handling

- Exception handling should not be confused with regular error checking.
- It should not be used as an alternative for control structures such as for, while and do loops.
- Do not use exception handling to prompt a user to enter data until certain condition has been fulfilled.
- Use of exception handling as an alternative control structure imposes a significant performance overhead.

```
#include <iostream>    // Example 9-9
using namespace std;

class Exit{}; //used as exception object

int main()
{
    int num;

     cout<< "enter a number; 99 to exit" <<endl;
```

Unit 8

```
    try
    {
        while (true) //infinitely
        {
            cin >> num;
            // Throw statement breaks the loop and transfers
            // control to the following catch statement.
            if (num == 99) throw Exit(); //exit the loop

            cout<< "you entered: " << num << "\n";
            cout << "enter another number " <<endl;
        } // end while loop
    }
    catch (Exit& )
    {
        cout<< "time to go home!" << endl;
    }

    return 0;
}
```