School of Computing and Informatics

BCS 362 – Generic Programming with C++

Exception Handling

1 Introduction

An **exception** indicates a problem that occurs when a program is executing. The name exception suggests that the problem occurs infrequently. Exception handling enables you to create fault tolerant programs that can process or handle exceptions. In many cases, this allows the program to continue executing as if no problem occured. C++ handles exceptions using three keyword throw, try and catch

- throw – A program throws an exception when a problem shows up. This is done using a throw keyword. This stops the program from executing
- try – A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks. Any code that mighty throw an exception is placed in this section
- catch – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception. Catch specific exception in this section and provide code to be executed if the exception occurred in the try block

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.

If an ellipsis (...) is used as the parameter of catch, that handler will catch any exception no matter what the type of the exception thrown. This can be used as a default handler that catches all exceptions not caught by other handlers.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

1.1 C++ Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called **std::exception** and is defined in the **exception** header. This class has a virtual member function called **what()** that returns a null-terminated character sequence (of type char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

Figure 1 shows std exception hierarchy in C++ The **what()** function can be overridden as

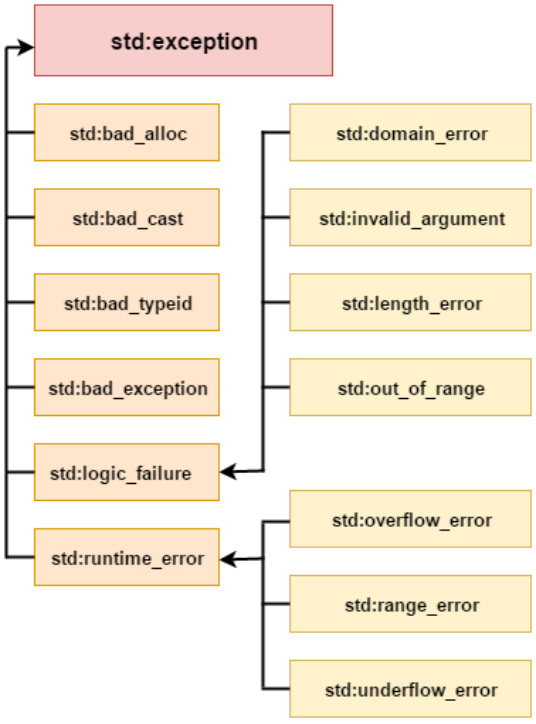

Figure 1: C++ std exceptions

```
#include <iostream>
#include <exception>
using namespace std;

class CustomException: public exception
{
```

```
  virtual const char* what() const throw() //you can override the what() function in exception
  {
    return "Customised message to be displayed";
  }
};

int main () {
  CustomException ce;
  try
  {
    throw ce;
  }
  catch (exception& e)
  {
    cout << e.what() << '\n'; //this line displays -- Customised message to be displayed
  }
  return 0;
}
```

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our **ce** object of type **CustomException**.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this exception class. These are Also deriving from exception, header exception defines two generic exception

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

types that can be inherited by custom exceptions to report errors. These are

exception	description
logic_error	error related to the internal logic of the program
runtime_error	error detected during runtime

1.2 Divide By Zero class

```
#include<iostream>
#include<stdexcept>
using namespace std;
class DivideByZero : public runtime_error
{
  public:
    DivideByZero() : runtime_error{"You attempted to divide by zero"}{}
    double divide(double a, double b)
    {
      if (b == 0)
        throw DivideByZero();
      return a / b;
    }
};
int main()
{
```

```
  DivideByZero dv;
  int a = 10;
  int b = 0;
  try
  {
    double q = dv.divide(a, b);
    cout << "Result = " << q << endl;
  }
  catch(DivideByZero &de)
  {
    cout << "Exception occurred: " << de.what() << endl;
  }
}
```

1.3 invalid_argument exception

```
#include<iostream>
#include<string>
#include<sstream>
#include<stdexcept>
using namespace std;
class LogicFailure : public invalid_argument
{
  public:
    LogicFailure() : invalid_argument{"Hour or minute or second is invalid. Invalid Time"}{}
};
class Time : public LogicFailure{
  private:
    int hour, min, sec;
  public:
    Time():hour{-1}, min{-1}, sec{-1}{}
    Time(int h, int m, int s) : hour{h}, min{m}, sec{s}{}
    void setHour(int h)
    {
      if (h >= 0 && h < 24)
        hour = h;
      else
        throw LogicFailure();
    }
    void setMin(int m)
    {
      if (m >= 0 && m < 60)
        min = m;
      else
        throw LogicFailure();
    }
    void setSec(int s)
    {
      if (s >= 0 && s < 60)
        sec = s;
      else
        throw LogicFailure();
    }
    void setTime(int h, int m, int s)
    {
      setHour(h);
      setMin(m);
      setSec(s);
    }
    string display()
    {
      ostringstream out;
```

```
47        out << hour << ":" << min << ":" << sec << endl;
48        return out.str();
49      }
50      bool validate()
51      {
52        return hour >= 0 && min >= 0 && sec >= 0;
53      }
54  };
55  int main()
56  {
57    Time t;
58    try
59    {
60      t.setTime(20, 45, 18);
61      cout << "Time set is : " << t.display();
62    }
63    catch(LogicFailure &lf)
64    {
65      t.setTime(0, 0, 0);
66      cout << "Exception :" << lf.what() << endl;
67      cout << "Default Time : " << t.display();
68    }
69  }
```