OOP345 – Exception Handling

Sukhbir Tatla

sukhbir.tatla@senecacollege.ca

Introduction

- Errors can be dealt with at place error occurs
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- Exception handling
  - Makes clear, robust, fault-tolerant programs
  - C++ removes error handling code from "main line" of program
- Common failures
  - **`new`** not allocating memory
  - Out of bounds array subscript
  - Division by zero
  - Invalid function parameters

Introduction

- Exception handling - catch errors before they occur
  - Deals with synchronous errors (ex., Divide by zero)
  - Does not deal with asynchronous errors - disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
  - Typically used when error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly
- Exception handling should not be used for program control
  - Not optimized, can harm program performance

Introduction

- Exception handling improves fault-tolerance
  - Easier to write error-processing code
  - Specify what type of exceptions are to be caught
- Most programs support only single threads
  - Techniques in this chapter apply for multithreaded OS as well (windows NT, OS/2, some UNIX)
- Exception handling is another way to return control from a function or block of code.

When Exception Handling Should Be Used

- Error handling should be used for
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Processing exceptions for widely used components (libraries, classes, functions) that should not process their own exceptions
  - Large projects that require uniform error processing

Exception Handling: *try, throw, catch*

- A function can **throw** an exception object if it detects an error
  - Object is typically a character string (error message) or class object
  - If exception handler exists, exception caught and handled
  - Otherwise, program terminates

Exception Handling: *try, throw, catch*

- Format
  - Enclose code that may have an error in **try** block
  - Follow with one or more **catch** blocks
    - Each **catch** block has an exception handler
  - If exception occurs and matches parameter in **catch** block, code in catch block executed
  - If no exception thrown, exception handlers skipped and control resumes after catch blocks
  - **throw** point - place where exception occurred
    - Control cannot return to **throw** point

A Simple Exception-Handling Example: Divide by Zero

- Look at the format of **try** and **catch** blocks
- Afterwards, we will cover specifics.

A Simple Exception-Handling Example: Divide by Zero

```
#include <iostream>
using namespace std;
class DivideByZeroException                    {
private:
        const char *message;
public:
        DivideByZeroException() : message("attempted to
divide by zero") { }
```

- `const char *what() const                    {`

```
                return message;
        }       };
```

A Simple Exception-Handling Example: Divide by Zero

```
double quotient(double numerator, double
denominator)
{
        if (denominator == 0)
        {
                throw DivideByZeroException();
        }
        return (numerator / denominator);
}
```

A Simple Exception-Handling Example: Divide by Zero

```
int main()       {
   double num1, num2;
   double result;  int answer=1;
   while (answer == 1) {
cout << "Enter two integers: ";
cin >> num1 >> num2;
try   {
    result = quotient(num1, num2);
cout << "The quotient is: " << result << endl; }
catch (DivideByZeroException ex)
{       cout << "Exception occurred: " << ex.what() << endl;   }
 cout << endl << "Do you want to contiue: Press 1 - Yes OR 2 - No …!!" << endl;
  cin >> answer;
  }
  cout << endl;
  return 0;       } //Main End
```

Throwing an Exception

- **throw** – indicates an exception has occurred
  - Usually has one operand (sometimes zero) of any type
    - If operand an object, called an exception object
    - Conditional expression can be thrown
  - Code referenced in a **try** block can throw an exception
  - Exception caught by closest exception handler
  - Control exits current try block and goes to **catch** handler
  - Example (inside function definition)

```
if ( denominator == 0 )
    throw DivideByZeroException();
```

    - Throws a **dividebyzeroexception** object
- Exception not required to terminate program
  - However, terminates block where exception occurred

Catching an Exception

- Exception handlers are in **catch** blocks
  - Format: **catch(** *exceptionType parameterName*)**{**
    exception handling code
    }
  - Caught if argument type matches **throw** type
  - If not caught then **terminate** called which (by default) calls **abort**
  - Example:

```
catch ( DivideByZeroException ex) {
  cout << "Exception occurred: " << ex.what() <<'\n'
}
```

    - Catches exceptions of type **DivideByZeroException**

Catching an Exception

- Catch all exceptions

  `catch(...)` - catches all exceptions

  - You do not know what type of exception occurred
  - There is no parameter name - cannot reference the object

- If no handler matches thrown object
  - Searches next enclosing **`try`** block
    - If none found, **`terminate`** called
  - If found, control resumes after last **`catch`** block
  - If several handlers match thrown object, first one found is executed

Catching an Exception

- **catch** parameter matches thrown object when
  - They are of the same type
    - Exact match required - no promotions/conversions allowed
  - The **catch** parameter is a **public** base class of the thrown object
  - The **catch** parameter is a base-class pointer/ reference type and the thrown object is a derived-class pointer/ reference type
  - The **catch** handler is **catch(...)**
  - Thrown **const** objects have **const** in the parameter type

Catching an Exception

- Unreleased resources
  - Resources may have been allocated when exception thrown
  - **catch** handler should **delete** space allocated by **new** and close any opened files

- **catch** handlers can throw exceptions
  - Exceptions can only be processed by outer **try** blocks

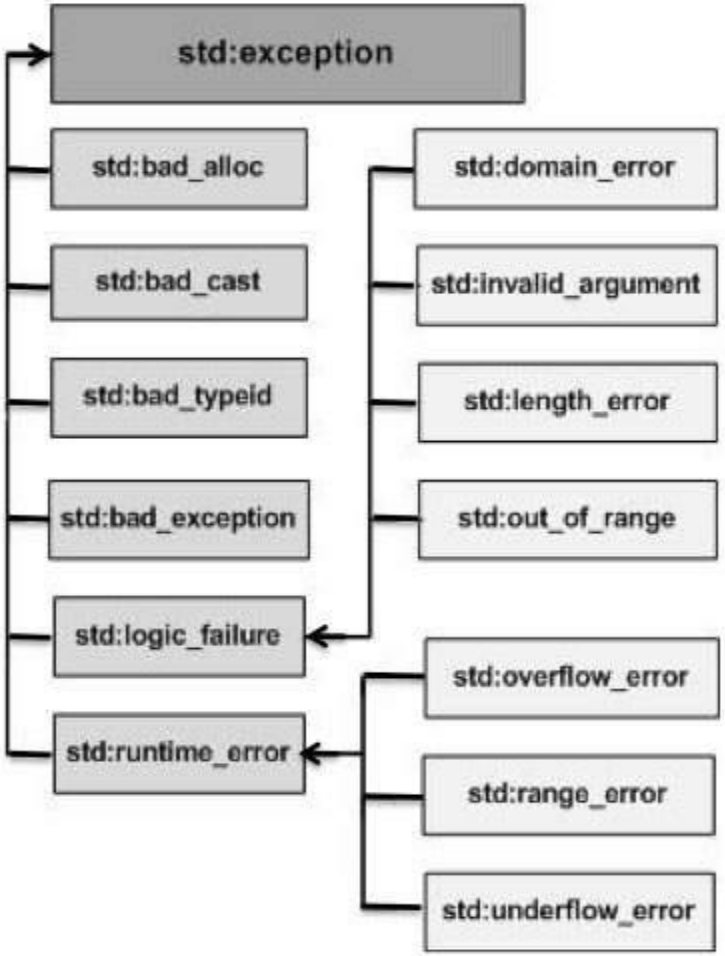
Rethrowing an Exception

- Rethrowing exceptions
  - Used when an exception handler cannot process an exception
  - Rethrow exception with the statement:

    throw;
    - No arguments
    - If no exception thrown in first place, calls **terminate**
  - Handler can always rethrow exception, even if it performed some processing
  - Rethrown exception detected by next enclosing **try** block

C++ Standard Exceptions

- C++ provides a list of standard exceptions defined in <exception> which we can use in our programs.



C++ Standard Exceptions

- Here is the small description of each exception:
- **std::exception:** An exception and parent class of all the standard C++ exceptions.
- **std::bad_alloc:** This can be thrown by **new**.
- **std::bad_cast:** This can be thrown by **dynamic_cast**.
- **std::bad_exception:** This is useful device to handle unexpected exceptions in a C++ program.
- **std::bad_typeid:** This can be thrown by **typeid**.
- **std::logic_error:** An exception that theoretically can be detected by reading the code.
- **std::domain_error:** This is an exception thrown when a mathematically invalid domain is used.

C++ Standard Exceptions

- Here is the small description of each exception:
- **std::invalid_argument:** This is thrown due to invalid arguments.
- **std::length_error:** This is thrown when a too big std::string is created.
- **std::out_of_range:** This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().
- **std::runtime_error:** An exception that theoretically cannot be detected by reading the code.
- **std::overflow_error:** This is thrown if a mathematical overflow occurs.
- **std::range_error:** This is occurred when you try to store a value which is out of range.

Program Example

```
#include <iostream>
#include <exception>
using namespace std;
void throwexception()                    {
        try {
        cout << "Function Throw Exception!!" << endl;
        throw exception();               }
        catch (exception e)              {
                cout << "Exception Handled in Function
                        throwexception()...!!!" << endl;
                throw;           }
        cout << "This also should not print..!!!" << endl;
}       //Function throwexception () Ends.
```

Program Example

```
int main()    {
        try {
           throwexception();
           cout << "This should not print..!!!" << endl;    }
        catch (exception e)          {
                cout << "Exception Handled in Function
main()...!!!" << endl;                    }
        cout << "Program control continues after catch in
main()" << endl;
        cout << endl;
        return 0;
}       //Function main() Ends
```

Exception Specifications

- Exception specification (**`throw`** list)
  - Lists exceptions that can be thrown by a function

  Example:

```
int g( double h ) throw ( a, b, c )  {
   // function body
}
```

  - Function can throw listed exceptions or derived types
  - If other type thrown, function **unexpected** called
  - **throw()** (i.e., no **throw** list) states that function will not throw any exceptions
    - In reality, function can still throw exceptions, but calls **unexpected** (more later)
  - If no **throw** list specified, function can **throw** any exception

Processing Unexpected Exceptions

- Function **unexpected**
  - Calls the function specified with **set_unexpected**
    - Default: **terminate**
- Function **terminate**
  - Calls function specified with **set_terminate**
    - Default: **abort**
- **set_terminate** and **set_unexpected**
  - Prototypes in **<exception>**
  - Take pointers to functions (i.E., Function name)
    - Function must return **void** and take no arguments
  - Returns pointer to last function called by **terminate** or **unexpected**

Constructors, Destructors and Exception Handling

- What to do with an error in a constructor?
  - A constructor cannot return a value - how do we let the outside world know of an error?
    - Keep defective object and hope someone tests it
    - Set some variable outside constructor
  - A thrown exception can tell outside world about a failed constructor
  - **`catch`** handler must have a copy constructor for thrown object

Constructors, Destructors and Exception Handling

- Thrown exceptions in constructors
  - Destructors called for all completed base-class objects and member objects before exception thrown
  - If the destructor that is originally called due to stack unwinding ends up throwing an exception, **`terminate`** called
  - If object has partially completed member objects when exception thrown, destructors called for completed objects

Constructors, Destructors and Exception Handling

- Resource leak
  - Exception comes before code that releases a resource
  - One solution: initialize local object when resource acquired
    - Destructor will be called before exception occurs

- **`catch`** exceptions from destructors
  - Enclose code that calls them in **`try`** block followed by appropriate **`catch`** block

THANK YOU!

Any Questions Please?