Module 37: Programming C++

Exceptions (Error handling in C++): Part 2

Sourangshu Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

sourangshu@cse.iitkgp.ac.in

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand the Error handling in C++

Module Outline

- Exception Fundamentals
  - Types of Exceptions
  - Exception Stages
- Exceptions in C
  - C Language Features
    - Return value & parameters
    - Local `goto`
  - C Standard Library Support
    - Global variables
    - Abnormal termination
    - Conditional termination
    - Non-local `goto`
    - Signal
  - Shortcomings
- Exceptions in C++
  - Exception Scope (`try`)
  - Exception Arguments (`catch`)
  - Exception Matching
  - Exception Raise (`throw`)
  - Advantages

Expectations

- Separate Error-Handling code from Ordinary code
- Language Mechanism rather than of the Library
- Compiler for Tracking Automatic Variables
- Schemes for Destruction of Dynamic Memory
- Less Overhead for the Designer
- Exception Propagation from the deepest of levels
- Various Exceptions handled by a single Handler

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- g() called

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- g() successfully returns

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- g() called and exception raised
- Stack frame of g() unwinds
- Remaining execution of g() and call of h() skipped
- Exception caught by catch clause
- Normal flow continues

Edited on 17-Feb-2021

Exception Flow


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

class MyException : public exception {};
class MyClass { ~MyClass() {} };

void h() { MyClass h_a;
    //throw 1;              // Line 1
    //throw 2.5;            // Line 2
    //throw MyException();  // Line 3
    //throw exception();    // Line 4
    //throw MyClass();      // Line 5
}   // Stack unwind, h_a.~MyClass() called
    // Passes on all exceptions

void g() { MyClass g_a;
    try {
        h();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 1
    catch (int)
        { cout << "int\n"; }
    // Catches exception from Line 2
    catch (double)
        { cout << "double\n"; }
    // Catches exception from Line 3-5
    // & passes on
    catch (...) { throw; }
}   // Stack unwind, g_a.~MyClass() called

void f() { MyClass f_a;
    try {
        g();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 3
    catch (MyException)
        { cout << "MyException\n"; }
    // Catches exception from Line 4
    catch (exception)
        { cout << "exception\n"; }
    // Catches exception from Line 5
    // & passes on
    catch (...) { throw; }
}   // Stack unwind, f_a.~MyClass() called

int main() {
    try {
        f();
        bool okay = true; // Not executed
    }
    // Catches exception from Line 5
    catch (...)
        { cout << "Unknown\n"; }

    cout << "End of main()\n";
    return 0;
}
```

try Block: Exception Scope

- try block
  - Consolidate areas that might throw exceptions
- function try block
  - Area for detection is the entire function body
- Nested try block
  - Semantically equivalent to nested function calls

Function try

```
void f()
    try {
        throw E();
    }
    catch (E& e) {
    }
```

Nested try

```
try {
    try { throw E(); }
    catch (E& e) { }
}
catch (E& e1) {
}
```

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- try Block

`catch` Block: Exception Arguments

- `catch` block
  - Name for the Exception Handler
  - Catching an Exception is like invoking a function
  - Immediately follows the try block
  - Unique Formal Parameter for each Handler
  - Can be simply a Type Name to distinguish its Handler from others

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- `catch` Block

try-catch: Exception Matching

- Exact Match
  - The catch argument type matches the type of the thrown object
    - No implicit conversion is allowed
- Generalization / Specialization
  - The catch argument is a public base class of the thrown class object
- Pointer
  - Pointer types – convertible by standard conversion

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- Expression Matching

`try-catch`: Exception Matching

- In the order of appearance with matching
- If Base Class `catch` block precedes Derived Class `catch` block
  - Compiler issues a warning and continues
  - Unreachable code (derived class handler) ignored
- `catch(...)` block must be the last `catch` block because it catches all exceptions
- If no matching Handler is found in the current scope, the search continues to find a matching handler in a dynamically surrounding `try` block
  - Stack Unwinds
- If eventually no handler is found, `terminate()` is called

`throw` *Expression*: Exception Raise

- *Expression* is treated the same way as
  - A function argument in a call or the operand of a return statement
- Exception Context
  - `class Exception ;`
- The *Expression*
  - Generate an Exception object to throw
    - `throw Exception();`
  - Or, Copies an existing Exception object to throw
    - `Exception ex;`
    - `...`
    - `throw ex; // Exception(ex);`
- Exception object is created on the Free Store

try-throw-catch


```
void f() {
        A a;
        try {
                B b;
                g();
                h();
        }
        catch (UsrExcp& ex) {
                cout <<
                ex.what();
        }

        return;
}
```

```
class UsrExcp:
    public exceptions {}

void g()
{
    A a;
    UsrExcp ex("From g()");

    throw ex;

    return;
}
```

- `throw` Expression

`throw` *Expression*: Restrictions

- For a UDT Expression
  - Copy Constructor and Destructor should be supported
- The type of Expression cannot be
  - An incomplete type (like `void`, array of unknown size or of elements of incomplete type, Declared but not Defined `struct` / `union` / `enum` / `class` Objects or Pointers to such Objects)
  - A pointer to an Incomplete type, except `void*`, `const void*`, `volatile void*`, `const volatile void*`

(re)-throw: Throwing Again?

- Re-throw
  - catch may pass on the exception after handling
  - Re-throw is not same as throwing again!

Throws again

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Raise again
    throw ex;
// ex copied
// ex destructed
}
```

Re-throw

```
try { ... }
catch (Exception& ex) {
    // Handle and
    ...
    // Pass-on
    throw;
    // No copy
// No Destruction
}
```

Advantages

- **Destructor-savvy**:
  - Stack unwinds; Orderly destruction of Local-objects
- **Unobtrusive**:
  - Exception Handling is implicit and automatic
  - No clutter of error checks
- **Precise**:
  - Exception Object Type designed using semantics
- **Native and Standard**:
  - EH is part of the C++ language
  - EH is available in all standard C++ compilers

Advantages

- **Scalable**:
  - Each function can have multiple try blocks
  - Each try block can have a single Handler or a group of Handlers
  - Each Handler can catch a single type, a group of types, or all types
- **Fault-tolerant**:
  - Functions can specify the exception types to throw; Handlers can specify the exception types to catch
  - Violation behavior of these specifications is predictable and user-configurable

Exceptions in Standard Library: `std::exception`

All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type by reference.

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
}
```

- `logic_error`: Faulty logic like violating logical preconditions or class invariants (may be preventable)
  - `invalid_argument`: An argument value has not been accepted
  - `domain_error`: Situations where the inputs are outside of the domain for an operation
  - `length_error`: Exceeding implementation defined length limits for some object
  - `out_of_range`: Attempt to access elements out of defined range
- `runtime_error`: Due to events beyond the scope of the program and can not be easily predicted
  - `range_error`: Result cannot be represented by the destination type
  - `overflow_error`: Arithmetic overflow errors (Result is too large for the destination type)
  - `underflow_error`: Arithmetic underflow errors (Result is a subnormal floating-point value)
- `bad_typeid`: Exception thrown on typeid of null pointer
- `bad_cast`: Exception thrown on failure to dynamic cast
- `bad_alloc`: Exception thrown on failure allocating memory
- `bad_exception`: Exception thrown by unexpected handler

Sources: `std::exception` and `std::exception in C++11, C++14, C++17 & C++20`

Exceptions in Standard Library: `std::exception`: C++98, C++11, C++14, C++17 & C++20

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
- bad_optional_access(C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
  - txtion(TM TS)
  - nonexistent_local_time(C++20)
  - ambiguous_local_time(C++20)
  - format_error(C++20)
- bad_typeid
- bad_cast
  - bad_any_cast(C++17)
- bad_weak_ptr(C++11)
- bad_function_call(C++11)
- bad_alloc
  - bad_array_new_length(C++11)
- bad_exception
- ios_base::failure(until C++11)
- bad_variant_access(C++17)

Module Summary

- Discussed exception (error) handling in C++
- Illustrated `try-throw-catch` feature in C++ for handling errors
- Demonstrated with examples