Exception Handling

Dr. Anwar Shah, PhD, MBA(HR)
Assistant Professor in CS
FAST National University of Computer and Emerging Sciences CFD

Section Overview

Exception Handling

- What is an Exception?
- What is Exception Handling?
- What do we throw and catch exceptions?
- How does it affect flow of control?
- Defining our own exception classes
- The Standard Library Exception Hierarchy
  - `std::exception` and `what()`

Basic Concepts

Exception Handling

Basic concepts

- Exception handling
  - dealing with extraordinary situations
  - indicates that an extraordinary situation has been detected or has occurred
  - program can deal with the extraordinary situations in a suitable manner

- What causes exceptions?
  - insufficient resources
  - missing resources
  - invalid operations
  - range violations
  - underflows and overflows
  - Illegal data and many others

- Exception safe
  - when your code handles exceptions

Exception Handling

Terminology

- Exception
  - an object or primitive type that signals that an error has occurred

- Throwing an exception (raising an exception)
  - your code detects that an error has occurred or will occur
  - the place where the error occurred may not know how to handle the error
  - code can throw an exception describing the error to another part of the program that knows how to handle the error

- Catching an exception (handle the exception)
  - code that handles the exception
  - may or may not cause the program to terminate

Exception Handling

C++ Syntax

- `throw`
  - throws an exception
  - followed by an argument

- `try { code that may throw an exception }`
  - you place code that may throw an exception in a try block
  - if the code throws an exception the try block is exited
  - the thrown exception is handled by a catch handler
  - if no catch handler exists the program terminates

- `catch(Exception ex)  { code to handle the exception }`
  - code that handles the exception
  - can have multiple catch handlers
  - may or may not cause the program to terminate

Exception Handling

Divide by zero example

- What happens if `total` is zero?
  - crash, overflow?
  - it depends

```
double average {};
average = sum / total;
```

Exception Handling

Divide by zero example

- What happens if `total` is zero?
  - crash, overflow?
  - it depends

```
double average {};
if (total == 0)
    // what to do?
else
    average = sum / total;
```

Exception Handling

Divide by zero example

```
double average {};
try {                              // try block
   if (total == 0)
      throw 0;                     // throw the exception
   average = sum / total;          // won't execute if total == 0
   // use average here
}
catch (int &ex) {                  // exception handler
   std::cerr << "can't divide by zero" << std::endl;
}
std::cout << "program continues" << std::endl;
```

Throwing an Exception from a Function

Exception Handling

Throwing an exception from a function

What do we return if total is zero?

```
double calculate_avg(int sum, int total) {
    return static_cast<double>(sum) / total;
}
```

Exception Handling

Throwing an exception from a function

Throw an exception if we can't complete successfully

```
double calculate_avg(int sum, int total) {
    if (total == 0)
        throw 0;
    return static_cast<double>(sum) / total;
}
```

Exception Handling

Catching an exception thrown from a function

```
double average {};

try {
    average = calculate_avg(sum, total);
    std::cout << average << std::endl;
}
catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}

std::cout << "Bye" << std::endl;
```

Handling Multiple Exceptions

Exception Handling

Throwing multiple exceptions from a function

What if a function can fail in several ways

- `gallons` is zero
- `miles` or `gallons` is negative

```
double calculate_mpg(int miles, int gallons) {
    return static_cast<double>(miles) / gallons;
}
```

Exception Handling

Throwing an exception from a function

Throw different type exceptions for each condition

```
double calculate_mpg(int miles, int gallons) {
    if (gallons == 0)
        throw 0;
    if (miles < 0 || gallons < 0)
        throw std::string{"Negative value error"};

    return static_cast<double>(miles) / gallons;
}
```

Exception Handling

Catching an exception thrown from a function

```
double miles_per_gallon {};
try {
    miles_per_gallon = calculate_mpg(miles, gallons);
    std::cout << miles_per_gallon << std::endl;
}
catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}
catch (std::string &ex) {
    std::cerr << ex << std::endl;
}

std::cout << "Bye" << std::endl;
```

Exception Handling

Catching any type of exception

```
catch (int &ex) {
}
catch (std::string &ex) {
}
catch (...) {
    std::cerr << "Unknown exception" << std::endl;
}
```

Stack Unwinding

Exception Handling

Stack unwinding

If an exception is thrown but not caught in the current scope
C++ tries to find a handler for the exception by unwinding the stack

- Function in which the exception was not caught terminates and is removed from the call stack
- If a try block was used to then catch blocks are checked for a match
- If no try block was used or the catch handler doesn't match stack unwinding occurs again
- If the stack is unwound back to main and no catch handler handles the exception the program terminates

Creating User-defined Exception Classes

Exception Handling

User-defined exceptions

We can create exception classes and throw instances of those classes

Best practice:

- throw an object not a primitive type
- throw an object by value
- catch an object by reference (or const reference)

Exception Handling

Creating exception classes

```
class DivideByZeroException {
};

class NegativeValueException {
};
```

Exception Handling

Throwing user-defined exception classes

```
double calculate_mpg(int miles, int gallons) {

    if (gallons == 0)
        throw DivideByZeroException();
    if (miles < 0 || gallons < 0)
        throw NegativeValueException();

    return static_cast<double>(miles) / gallons;
}
```

Exception Handling

Catching user-defined exceptions

```
try {
   miles_per_gallon = calculate_mpg(miles, gallons);
   std::cout << miles_per_gallon << std::endl;
}
catch (const DivideByZeroException &ex) {
   std::cerr << "You can't divide by zero" << std::endl;
}
catch (const NegativeValueException &ex) {
   std::cerr << "Negative values aren't allowed" << std::endl;
}

std::cout << "Bye" << std::endl;
```

Class Level Exceptions

Exception Handling

Class-level exceptions

Exceptions can also be thrown from within a class:

- Method
  - These work the same way as they do for functions as we've seen
- Constructor
  - Constructors may fail
  - Constructors do not return any value
  - Throw an exception in the constructor if you cannot initialize an object
- Destructor
  - Do NOT throw exceptions from your destructor

Exception Handling

Class-level exceptions

```
Account::Account(std::string name, double balance)
    : name{name}, balance{balance} {

    if (balance < 0.0)
        throw IllegalBalanceException{};

}
```

Exception Handling

Class-level exceptions

```
try {
    std::unique_ptr<Account> moes_account =
          std::make_unique<Checking_Account>("Moe",-10.0);
    // use moes_account
}
catch (const IllegalBalanceException &ex) {
    std::cerr << "Couldn't create account" << std::endl;
}
```

Memory Leaks

Memory leaks during stack unwinding can occur in C++ when an exception is thrown, and during the process of unwinding the call stack to find an appropriate exception handler, the destructors of local objects are called. If memory allocated within those destructors is not properly released, it can lead to a memory leak.

Memory Leaks

Allocation in a Constructor:

If a constructor of an object allocates memory on the heap using new, and an exception is thrown before the destructor is called, the memory allocated in the constructor may not be deallocated.

```
class MyClass {
public:
    MyClass() {
        // Memory allocated in constructor
        data = new int[100];
        // ...
        // Some code that may throw an exception
        // ...
    }

    ~MyClass() {
        // Destructor where memory should be deallocated
        delete[] data;
    }

private:
    int* data;
};
```

Memory Leaks

Exception During Object Construction:

If an exception is thrown during the construction of an object, the destructors of the already constructed objects in the call stack will be called during stack unwinding. If any of these destructors fail to release allocated resources properly, a memory leak can occur.

```
void someFunction() {
    MyClass obj1;  // Constructor allocates memory

    // Exception thrown during construction of obj2
    throw SomeException();
}
```

Memory Leaks

Handling Memory in Destructors:

It's essential to handle memory properly in the destructors of C++ objects. If an object allocates resources (memory, file handles, etc.) during its lifetime, the destructor should be responsible for releasing those resources.

```
class ResourceHolder {
public:
    ResourceHolder() {
        data = new int[100];
    }

    ~ResourceHolder() {
        // Properly release allocated memory
        delete[] data;
    }

private:
    int* data;
};
```

Memory Leaks

Handling Memory in Destructors:

If an object is allocated with automatic storage duration (i.e., it is not dynamically allocated with new), its destructor will be automatically called when it goes out of scope. However, if an exception is thrown before the object's destructor is called, proper cleanup might be skipped.

```
void someFunction() {
    MyClass obj;  // Destructor may not be called if an exception occurs here
    // Some code that may throw an exception
}
```

Memory Leaks

To avoid memory leaks during stack unwinding, it's crucial to follow good resource management practices:

- Use smart pointers (std::shared_ptr, std::unique_ptr) to manage dynamic memory whenever possible.
- Avoid using raw pointers and manual memory management when it's not necessary.
- Implement proper resource cleanup in destructors.
- Be cautious with operations that can throw exceptions, especially within constructors. If an exception occurs, destructors of already constructed objects in the call stack will be called during stack unwinding.

The C++ std__exception Class Hierarchy

Exception Handling

The C++ standard library exception class hierarchy

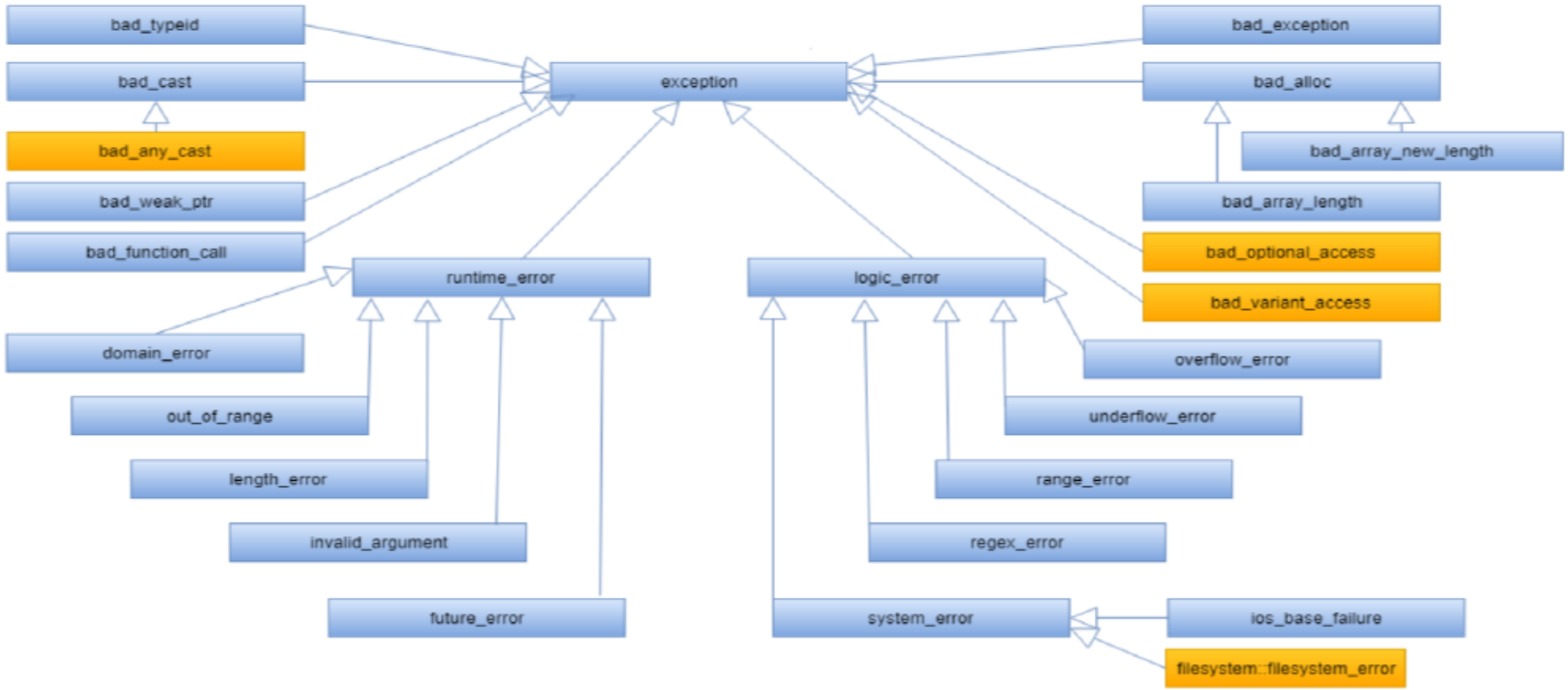
C++ provides a class hierarchy of exception classes

- `std::exception` is the base class
- all subclasses implement the `what()` virtual function
- we can create our own user-defined exception subclasses

```
virtual const char *what() const noexcept;
```

Exception Handling

The C++ standard library exception class hierarchy



Exception Handling

Deriving our class from std::exception

```
class IllegalBalanceException: public std::exception
{
public:
    IllegalBalanceException() noexcept = default;
    ~IllegalBalanceException() = default;
    virtual const char* what() const noexcept {
        return "Illegal balance exception";
    }
};
```

Exception Handling

Our modified Account class constructor

```
Account::Account(std::string name, double balance)
    : name{name}, balance{balance} {
    if (balance < 0.0)
        throw IllegalBalanceException{};
}
```

Exception Handling

Creating an Account object

```
try {
    std::unique_ptr<Account> moes_account =
        std::make_unique<Checking_Account>("Moe", -100.0);

    std::cout << "Use moes_account" << std::endl;
}
catch (const IllegalBalanceException &ex)
{
    std::cerr << ex.what() << std::endl;
          // displays "Illegal balance exception"
}
```

More Explanations

Memory Leaks

Exception During Construction:

Memory leaks during object construction can occur when resources are allocated, and an exception is thrown before those resources are properly released. Here are some scenarios that can lead to memory leaks during object construction:

Memory Leaks

Exception During Construction:

If an exception occurs during the construction of an object, and the constructor has already allocated resources (e.g., memory), those resources might not be properly released.

This is particularly problematic if the constructor uses new to allocate memory. In this example, if an exception is thrown during the construction of MyClass, the destructor will not be called, leading to a memory leak.

```
class MyClass {
public:
    MyClass() {
        // Memory allocated during construction
        data = new int[100];
        // Some code that may throw an exception
        throw std::runtime_error("Exception during construction");
    }

    ~MyClass() {
        // Destructor should clean up allocated memory
        delete[] data;
    }

private:
    int* data;
};
```

Memory Leaks

Objects with Automatic Storage Duration:

If an object is created with automatic storage duration and an exception is thrown before the constructor completes, the destructor for that object may not be called, leading to potential resource leaks.

```
void someFunction() {
    MyClass obj;  // If an exception occurs during construction, destructor may not be called
    // Some code that may throw an exception
}
```

Memory Leaks

To mitigate memory leaks during object construction:

Use smart pointers (std::shared_ptr, std::unique_ptr) to manage dynamic memory whenever possible. Smart pointers automatically handle memory deallocation when an exception occurs during construction.

```
class MyClass {
public:
    MyClass() {
        // Memory allocated during construction
        data = std::make_unique<int[]>(100);
        // Some code that may throw an exception
        throw std::runtime_error("Exception during construction");
    }

private:
    std::unique_ptr<int[]> data;
};
```

Memory Leaks

To mitigate memory leaks during object construction:

If using raw pointers and manual memory management, ensure proper cleanup in the destructor, and catch and handle exceptions within the constructor to release resources before propagating the exception.

```
class MyClass {
public:
    MyClass() {
        // Memory allocated during construction
        data = new int[100];
        try {
            // Some code that may throw an exception
            throw std::runtime_error("Exception during construction");
        } catch (...) {
            // Cleanup resources before propagating the exception
            delete[] data;
            throw;  // Re-throw the exception
        }
    }

    ~MyClass() {
        // Destructor should clean up allocated memory
        delete[] data;
    }

private:
    int* data;
};
```

Good Luck!