Exception Handling

Exception handling is a mechanism to handle runtime errors such as division by zero, array out of bounds, etc. It is a way to transfer control from one part of a program to another.
In C++, exceptions are thrown using the **throw** keyword and caught using the **try-catch** block. The **try** block contains the code that may throw an exception, and the **catch** block contains the code that handles the exception.

```
try {
    // code that may throw an exception
    throw MyException();
}
catch (MyException& e) {
    // code to handle the exception
}
```

The **throw** statement can throw any type of object, including built-in types, standard library types, and user-defined types. The **catch** block can catch exceptions of a specific type or a base class type.

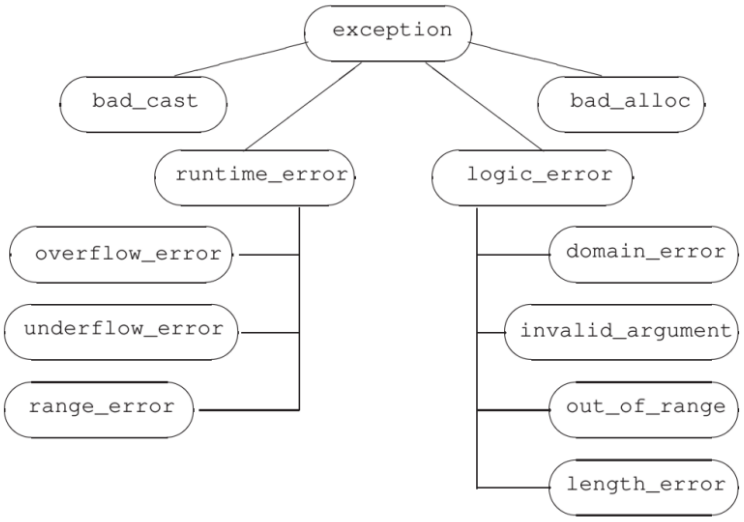
In the following example, the `safe_division()` function throws an exception of type `std::runtime_error` if the denominator is zero. The exception is caught in the `main` function and the error message is printed.

```
#include <iostream>
#include <stdexcept>
using std::cin, std::cout;

double safe_division(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero");
    }
    return a / b;
}
int main() {
    try {
        int x, y;
        cout << "Enter two numbers: ";
        cin >> x >> y;
        double result = safe_division(x, y);
        cout << "Result: " << result << "\n";
    }
    catch (std::runtime_error& e) {
        cout << "Caught an exception: " << e.what() << "\n";}}
```

The <stdexcept> Header

The <stdexcept> header file contains the following class hierarchy:



The base class `std::exception` has a virtual method `what()` that returns a string describing the exception. The `std::runtime_error` class is derived from `std::exception` and is used to report runtime errors. The `std::logic_error` class is used to report errors that are caused by the program logic, such as invalid arguments to a function. Further derived classes include `std::invalid_argument`, `std::domain_error`, `std::length_error`, etc.

Upcasting and Downcasting

1. ***Upcasting*** is the process of converting a pointer or reference of a derived class to a pointer or reference of a base class. It is done implicitly and does not require any explicit casting.

2. ***Downcasting*** is the process of converting a pointer or reference of a base class to a pointer or reference of a derived class. It is done explicitly using the `static_cast` or `dynamic_cast` operators.

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ };

void f() {
    Derived d;
    Base* pb = &d; // upcasting
    Derived* pd = static_cast<Derived*>(pb); // downcasting
}
```

static_cast and dynamic_cast

The `static_cast` and `dynamic_cast` operators are used for type conversion between different types of pointers and references. They can be used for upcasting, downcasting, and sideways conversions between pointers or references of classes in the inheritance hierarchy. They can also be used to convert between different types of pointers, such as converting a `void*` pointer to another type of pointer. The `static_cast` is done at compile time and does not perform any runtime type checking. While the `dynamic_cast` is done at runtime and performs runtime type checking to ensure that the conversion is valid.

```
class Base {
public:
    virtual ~Base() {} // Make Base polymorphic
    virtual void print() {
        std::cout << "Base class" << std::endl;
    }
};

class Derived : public Base {
public:
    void print() override {
        std::cout << "Derived class" << std::endl;
    }
    void derivedMethod() {
        std::cout << "Derived method called!" << std::endl;
    }
};

int main() {
    // Upcasting: Derived object treated as Base
    Derived derivedObj;
    Base* basePtr = &derivedObj; // Implicit upcasting

    // Downcasting using static_cast
    Derived* derivedPtrStatic = static_cast<Derived*>(basePtr);
    std::cout << "Using static_cast: ";
    derivedPtrStatic->print(); // Calls Derived::print()
    derivedPtrStatic->derivedMethod(); // Calls Derived::derivedMethod()

    // Downcasting using dynamic_cast
    Derived* derivedPtrDynamic = dynamic_cast<Derived*>(basePtr);
    if (derivedPtrDynamic) {
        std::cout << "Using dynamic_cast: ";
        derivedPtrDynamic->print(); // Calls Derived::print()
```

```
        derivedPtrDynamic->derivedMethod(); // Calls Derived::derivedMethod()
    } else {
        std::cout << "dynamic_cast failed!" << std::endl;
    }
    return 0;
}
```

When applied to pointers, the `dynamic_cast` returns a null pointer if the conversion is not valid. When applied to references, it throws an exception of type `std::bad_cast` if the conversion is not valid.

```
Base* pb = new Derived;
Derived* pd = dynamic_cast<Derived*>(pb);
if (pd != nullptr) {
    // conversion is valid
}
else {
    // conversion is not valid
}

Base& rb = *pb;
try {
    Derived& rd = dynamic_cast<Derived&>(rb);
    // conversion is valid
}
catch (std::bad_cast& e) {
    // conversion is not valid
}
```

The following example uses `dynamic_cast` to do runtime type checking and downcasting to count the number of students in a vector of `Person` pointers.

```
class Person { /* ... */ };
class Student : public Person { /* ... */ };
class Staff : public Person { /* ... */ };

int count_students(vector<Person*>& people) {
    int count = 0;
    for (Person* p : people) {
        if (dynamic_cast<Student*>(p) != nullptr) {
            count++;
        }
    }
    return count;
}
```

Custom Exception Classes in C++

In C++, exceptions are used to handle errors and other exceptional events. The standard library provides a set of predefined exception classes (e.g., `std::exception`, `std::runtime_error`, `std::logic_error`, etc.). However, sometimes you may need to create custom exception classes to handle specific errors in your application.

To create a custom exception class, you typically inherit from the `std::exception` class or one of its derived classes (like `std::runtime_error` or `std::logic_error`). The `std::exception` class has a virtual member function called `what()` that returns a C-style string describing the exception. By overriding this method, you can provide a custom error message for your exception class.

Steps to Create a Custom Exception Class

1. **Inherit from `std::exception` or its derived classes**: This allows your custom exception to be caught by `catch` blocks that handle `std::exception`.
2. **Override the `what()` method**: This method should return a C-style string (`const char*`) that describes the exception.
3. **Add any additional data members or methods**: You can add extra data members to store additional information about the exception.

Example 1: Basic Custom Exception Class

```
#include <iostream>
#include <exception>
#include <string>

class MyCustomException : public std::exception {
private:
    std::string message;
public:
    // Constructor to initialize the exception message
    MyCustomException(const std::string& msg) : message(msg) {}

    // Override the what() method to return the custom message
    const char* what() const noexcept override {
        return message.c_str();
    }
};
int main() {
    try {
        // Throw the custom exception
        throw MyCustomException("This is a custom exception!");
    }
```

```
    catch (const std::exception& e) {
        // Catch the exception and print the message
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Output:

```
Caught exception: This is a custom exception!
```

Explanation:

- **Inheritance**: `MyCustomException` inherits from `std::exception`.
- **Constructor**: The constructor takes a `std::string` as an argument and initializes the `message` member variable.
- **`what()` method**: The `what()` method is overridden to return the `message` as a C-style string (`const char*`).
- **`noexcept`**: The `what()` method is marked as `noexcept` to indicate that it does not throw exceptions.

Example 2: Custom Exception with Additional Data

You can also add additional data members to your custom exception class to provide more context about the error.

```
#include <iostream>
#include <exception>
#include <string>

class DivisionByZeroException : public std::exception {
private:
    std::string message;
    int dividend;

public:
    // Constructor to initialize the exception message and dividend
    DivisionByZeroException(const std::string& msg, int div) : message(msg),
       dividend(div) {}

    // Override the what() method to return the custom message
    const char* what() const noexcept override {
        return message.c_str();
    }
```

```
    // Additional method to get the dividend
    int getDividend() const {
        return dividend;
    }
};

int divide(int a, int b) {
    if (b == 0) {
        throw DivisionByZeroException("Division by zero occurred!", a);
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
    } catch (const DivisionByZeroException& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
        std::cerr << "Dividend was: " << e.getDividend() << std::endl;
    }

    return 0;
}
```

Output:

```
Caught exception: Division by zero occurred!
Dividend was: 10
```

Explanation:

- **Additional Data Member**: The `DivisionByZeroException` class has an additional data member `dividend` to store the value of the dividend when the exception occurs.
- **Additional Method**: The `getDividend()` method is added to retrieve the dividend value.
- **Exception Throwing**: The `divide()` function throws the `DivisionByZeroException` if a division by zero is attempted.

Example 3: Using `std::runtime_error` as Base Class

Instead of inheriting directly from `std::exception`, you can inherit from `std::runtime_error` or `std::logic_error`, which already provide a constructor to set the error message.

```
#include <iostream>
#include <stdexcept>
#include <string>

class FileNotFoundException : public std::runtime_error {
public:
    // Constructor to initialize the exception message
    FileNotFoundException(const std::string& filename)
        : std::runtime_error("File not found: " + filename) {}
};

int main() {
    try {
        // Simulate a file not found scenario
        throw FileNotFoundException("example.txt");
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }

    return 0;
}
```

Output:

```
Caught exception: File not found: example.txt
```

Explanation:

- **Inheritance**: `FileNotFoundException` inherits from `std::runtime_error`.
- **Constructor**: The constructor takes a filename and passes a formatted message to the base class constructor.
- **`what()` method**: The `what()` method is already implemented in `std::runtime_error`, so you don't need to override it unless you want to customize it further.

Creating custom exception classes in C++ allows you to handle specific error conditions in a more structured and meaningful way. By overriding the `what()` method, you can provide detailed error messages, and by adding additional data members, you can include more context about the error. This makes your code more robust and easier to debug.

Lab exercises

Exercise 1

Age Verification for a Movie Ticket: You are developing a program for a movie theater that sells tickets. The theater has a rule that only people aged 13 or older can watch a particular movie. If someone under 13 tries to buy a ticket, the program should throw an exception.

Write a function `void buyTicket(int age)` that:

- Throws a `std::invalid_argument` exception if the age is less than 13.
- Displays `"Ticket purchased successfully!  Enjoy the movie."` otherwise.

In `main()`, prompt the user for their age, call `buyTicket`, and handle any exceptions.

Exercise 2

Password Strength Checker: You are building a program to check the strength of a password. The program should enforce the following rules:

- The password must be at least 8 characters long.
- The password must contain at least one digit.
- If the password does not meet these requirements, the program should throw an exception.

Write a function `void checkPassword(const std::string& password)` that:

- Throws a `std::runtime_error` if the password is less than 8 characters long.
- Throws a `std::runtime_error` if the password does not contain at least one digit.
- Prints `"Password is strong and valid."` if both conditions are met.

In `main()`, prompt the user to enter a password, call `checkPassword`, and handle any exceptions.

Example Output:

```
Enter your password: abc
Caught exception: Password is too short. It must be at least 8 characters long.

Enter your password: password
Caught exception: Password must contain at least one digit.

Enter your password: password123
Password is strong and valid.
```

Exercise 3

Shape Drawing System: You are building a system to draw different shapes. The system has a base class `Shape` and two derived classes: `Circle` and `Rectangle`. Each shape has a method `void draw()` that prints the shape being drawn. Additionally, `Circle` has a method `void setRadius(double radius)` to set its radius.

Task:

1. Create a base class `Shape` with a virtual method `void draw()` that prints `"Drawing a shape."`.
2. Create two derived classes:
   - **Circle**: Overrides `draw()` to print `"Drawing a circle."` and adds a method `void setRadius(double radius)` that prints `"Circle radius set to <radius>."`.
   - **Rectangle**: Overrides `draw()` to print `"Drawing a rectangle."`.

Implementation in `main()`:

- Create a `Circle` object and a `Rectangle` object.
- Store them in a **vector of `Shape*` pointers**.
- Iterate through the vector and call `draw()` for each shape.
- For circles only, call `setRadius()`. (Hint: You'll need to determine if a `Shape*` pointer actually points to a `Circle` object.)

Exercise 4

Custom Exception for Banking System: You are building a simple banking system where users can perform transactions such as withdrawing money, depositing money, and transferring funds. The system should handle specific errors, including:

- **InsufficientBalanceException**: Thrown when a user tries to withdraw more money than their account balance.
- **InvalidAmountException**: Thrown when a user tries to deposit or withdraw a negative or zero amount.
- **AccountNotFoundException**: Thrown when a user tries to transfer funds to an account that does not exist.

Task:

1. Create custom exception classes for the above scenarios by inheriting from `std::exception` or `std::runtime_error`.
2. Write a class `BankAccount` with the following methods:
   - `void withdraw(double amount)`: Throws `InsufficientBalanceException` if the amount exceeds the balance and `InvalidAmountException` if the amount is invalid.
   - `void deposit(double amount)`: Throws `InvalidAmountException` if the amount is invalid.

- `void transfer(BankAccount& target, double amount)`: Throws `InsufficientBalanceException` if the amount exceeds the balance, `InvalidAmountException` if the amount is invalid, and `AccountNotFoundException` if the target account does not exist (simulated by checking if the target account is valid).

Implementation in `main()`:

- Demonstrate exception handling for withdrawals, deposits, and transfers.
- Display appropriate error messages when an exception is thrown.

Expected Output:

```
Attempting to withdraw $200 from account 123...
Caught exception: Insufficient balance in account 123.

Attempting to deposit $-50 into account 123...
Caught exception: Invalid amount: $-50.

Attempting to transfer $100 to account 456...
Caught exception: Account not found: 456.

Attempting to transfer $50 to account 789...
Transfer successful!
```