Exception Handling
Introduction

• Common examples of exceptions
  – Failure of new to obtain memory
  – Out-of-bounds array subscript
  – Division by zero
  – Invalid function parameters

• Programs with exception handling
  – Clearer, more robust, more fault-tolerant
  – Can recover from errors
  – Can shut down “gracefully”
Basics of C++ Exception Handling

- Enclose the code that may generate an error in a `try` block
- Follow `try` block with `catch` blocks
- Function that detects an error is unable to deal with it
  - Just `throw` an exception
  - No guarantee the exception will be handled
  - The thrown object is typically a string or a class object
#include<iostream>
class DivBy0 {
    public:
        DivBy0():msg("attempted to divide by0"){}
        const char *what() const {return msg;}
    private:
        const char *msg;
};

double quotient(int oper1, int oper2){
    if (oper2==0)
        throw DivBy0();
    return static_cast<double>(oper1)/oper2;
}
void main() {
    int num1, num2;
    double result;
    while (cin >> num1 >> num2) {
        try{
            result = quotient(num1, num2);
            cout << "result is: " << result << endl;
        }
        catch (DivBy0 ex) {
            cout << "Exception: " << ex.what << "\n";
        }
    }
}
Throwing an Exception

- **Throw** indicates an exception has occurred – *throwing an exception*
  - Normally specifies one operand of any type
  - A temporary copy of the throw operand is created, it is destroyed when the exception handler completes execution
Catching an Exception

- Exception handlers are contained in `catch` blocks
  - `catch` is followed by parentheses containing a type and an optional parameter name
  - The first one after the try block that matches the type of thrown object catches exception
  - An exception that is not caught causes a call to `terminate`, by default `abort`
  - `catch(...)` means to catch all exceptions
    - Must be placed last
Match Occurs When...

- The type of the catch handler A matches the thrown object B
  - They are indeed of the same type
  - A is a public base class of the class of B
  - A is of a base-class pointer/reference and B is of a derived-class pointer/reference
  - The handler is catch(...)
Rethrowing an Exception

- Handler can rethrow the exception
  - Unable to process
  - Just release resources

```c++
void throwException()
{
    try {
        cout << "Function throwException\n";
        throw exception();
    } catch (exception e) {
        cout << "Exception handled\n";
        throw;
    }
    cout << "I am here 2\n";
}
```
Rethrowing an Exception

void main()
{
    try {
        throwException();
        cout << "I am here 1\n";
    } catch (exception e) {
        cout << "Exception handled in main\n";
    }
    cout << "Control continues\n";
}

Function throwException
Exception handled
Exception handled in main
Control continues
Exception Specification

- It lists the exception that can be thrown
  - A function may throw the indicated exceptions or derived types with public inheritance.
  - If an exception not listed is thrown, function `unexpected` is called.
  - `unexpected` calls the function specified with `set_unexpected` function. If no function specified, `terminate` is called.
  - Prototype of `set_unexpected`: `<exception>`
Exception Specification

// can only throw a, b, or c
void foo(double h) throw (a, b, c) {
  //function body
}

// an empty exception specification will not throw any exceptions
void foo(double h) throw () {
  //function body
}

// a function with no exception specification can throw any exceptions
void foo(double h) {
  //function body
}
Stack Unwinding

```cpp
#include <stdexcept>
void foo3() throw (runtime_error) {
    throw runtime_error("error in foo3");
}

void foo2() throw (runtime_error) {
    foo3();
}

void foo1() throw (runtime_error) {
    foo2();
}
```
void main()
{
    try{
        foo1();
    }
    catch (runtime_error e) {
        cout << "Exception: " << e.what() << endl;
    }
}

Exception: error in foo3
Standard Exception Hierarchy

- Base class `exception (<exception>)` contains `what()`
  - `runtime_error`: errors detected at execution time (`<stdexcept>`) - `overflow_error, underflow_error`
  - `logic_error`: error in logic that can be prevented by proper coding(`<stdexcept>`) - `Invalid_argument, out_of_range`
When Exception Handling Should be Used

- It can be used as an alternative to program control
- However, should be used to process only exceptional situations