Object-Oriented Programming

Errors and Exception Handling

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Importance of Error Handling

- Often ignored and more complex than anticipated
- Design for the "happy path": method naming, parameter types, etc.
- Design of error handling for the non-happy path (unhappy path) introduces further complexity to an already existing API
- Includes *what* indicates an error, e.g., int values, strings, and what errors map to which particular *values*

Error Handling Choices

- Error Return
- Error Parameter
- Global State

Error Return

```
int inc(int n) {
    if (n < 0)
        return -1;

    return n + 1;
}
// ...
auto result = inc(n);
if (result == -1) {
    std::cerr << "inc(n): invalid value\n";
    return;
}
```

- Return error code
- Typically, an `int` or `bool`
- Check value after each use (which is easy to forget)
- Return error may be mixed into normal processing return

Error Parameter

```
int inc(int n, int& error) {
    if (n < 0) {
        error = -1;
        return 0;
    }

    return n + 1;
}
// ...
int error = 0;
auto result = inc(n, error);
if (error == -1) {
    std::cerr << "inc(n): invalid value\n";
    return;
}
```

- A parameter contains an error code
- Similar to *error return* except not mixed with normal processing return
- Have to declare a variable before the call

Global Error Value

```
int fd = open("data.xml", O_RDONLY);
if (fd == -1) {
    perror("open");
    exit(1);
}
```

- A global variable contains the error results
- May be mixed with *error return* or *error parameter* for more information
- For `perror()`, the global `errno` contains the last error code
- `perror()` usage in the Linux kernel ...
- Code to reset or get the global, shared value may be confusing

Problems with Error Codes

- Do not work with constructors, as constructors do not return a value
- Error codes do not have consistent values across applications

  libarchive

  libcurl
- Numeric results are not obvious, and name forms are often used/needed
- Propagating an error code is tedious

Error Propagation Flow

```
int f(int n) {
    if (n < 0)
        return -1;

    return n + 1;
}

int g(int n) {
    if (n > 100)
        return -1;

    return f(n) * 2;
}

int h(int n) {
    if (n != 50)
        return -1;

    return g(n) / 2;
}
// ...
auto result = h(n);
if (result == -1) {
    std::cerr << "Error\n";
    return;
}
```

- Calls to h( ) only get h( ) errors
- Calls from h( ) to g( ) and g( ) to f( ) are not propagated
- Every call to a function with an error return or parameter must pass on that error to the calling function
- Disturbs normal processing
- Even with one layer missing, it breaks the error propagation
- Not easy to do correctly

Exception Handling by Keyword

```cpp
int f(int n) {
    if (n < 0)
        throw std::domain_error("Negative value");

    return n + 1;
}

int g(int n) {
    // limited length of result
    if (n > 100)
        throw std::length_error("Over 100");

    return f(n) * 2;
}

int h(int n) {
    if (n == 50)
        throw std::invalid_argument("50 is not allowed");

    return g(n) / 2;
}

int main(int argc, char* argv[]) {
    // ...
    try {
        int n = std::stoi(argv[1]);
        auto result = h(n);
    } catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << '\n';
        return 1;
    }
    // ...
}
```

- `throw`

  Where an error is detected, the function/method immediately returns from that point in the code

- `catch`

  Layered calls are returned (stack unwinding) until one that matches is found

- `try`

  The `catch` is associated with the code in the `try` block

Resumption vs. Termination

```
/* A */    int result = 0;
           try {

/* B */        result = f1();
/* C */        result += f2();
/* D */        if (result > 100)
/* E */            throw std::exception();

/* I */        result += f3();

/* F */    } catch (std::exception& e) {
/* G */        std::cerr << "Exception: Too large\n";
/* H */        result = 0;
           }

/* J */    std::cout << "Result: " << result << '\n';
```

```
/* A */    int result = 0;
           try {

/* B */        result = f1();
/* C */        result += f2();
/* D */        if (result > 100)
/* E */            throw std::exception();

               result += f3();

/* F */    } catch (std::exception& e) {
/* G */        std::cerr << "Exception: Too large\n";
/* H */        result = 0;
           }

/* I */    std::cout << "Result: " << result << '\n';
```

Resumption Semantics

```
/* A */    int result = 0;
           try {

/* B */        result = f1();
/* C */        result += f2();
/* D */        if (result > 100)
/* E */            throw std::exception();

/* I */        result += f3();

/* F */    } catch (std::exception& e) {
/* G */        std::cerr << "Exception: Too large\n";
/* H */        result = 0;
           }

/* J */    std::cout << "Result: " << result << '\n';
```

- When the code generates an exception, go to where caught, process catch, and then return to the original point in the code
- C++ does not suppor *resumption semantics* (but some language do)
- Was under consideration for C++ in the early days of C++ exception handling

Termination Semantics

```
/* A */    int result = 0;
           try {

/* B */        result = f1();
/* C */        result += f2();
/* D */        if (result > 100)
/* E */            throw std::exception();

               result += f3();

/* F */    } catch (std::exception& e) {
/* G */        std::cerr << "Exception: Too large\n";
/* H */        result = 0;
           }

/* I */    std::cout << "Result: " << result << '\n';
```

- When the code generates an exception, go to where caught, process catch, and continue in the code at the catch point

- C++ only supports *termination semantics*

How to use Exceptions

Exception-handling code is error-handling code

- Separates the "good path" (or "happy path") from the "bad path"
- Errors where the code cannot handle the error locally, e.g., a constructor
- Do not use exception handling for normal processing
- Since they automatically deallocate, RAII is a crucial technique to use for all variables

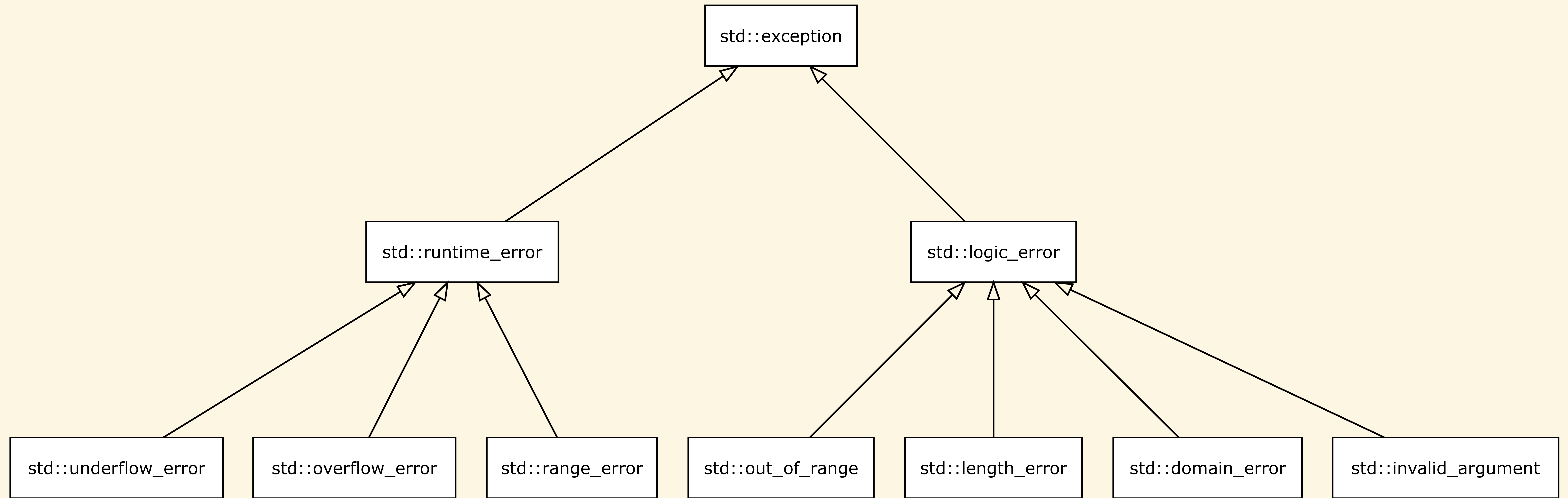
Why RAII is Important

```
void processFile(const char* s)  {
    // open the file named s
    FILE* f = fopen(s,"r");

    // ... processing that can throw an exception, or call code that can throw an exception

    // close the file named s
    fclose(f);
}
```

Partial std::exception Hierarchy



Full Set

`std::logic_error`

faulty logic within the program, such as violating logical preconditions or class invariants, and may be preventable

- `std::invalid_argument`

  Argument value is not accepted

- `std::domain_error`

  Inputs are outside of the domain defined for the operation

- `std::length_error`

  Exceeds implementation-defined length limits for some object

- `std::out_of_range`

  Access elements out of the defined range

std::runtime_error

errors due to events beyond the scope of the program, and where the program code cannot easily predict the error

- `std::range_error`

  The destination type cannot represent the result of a computation

- `std::overflow_error`

  Arithmetic overflow errors, i.e., the result of a computation is too large for the destination type

- `std::underflow_error`

  The result of a computation is a subnormal floating-point value

`catch` Handlers

```
try {
    int n = std::stoi(argv[1]);
    auto result = h(n);

} catch (std::domain_error& e) {
    std::cerr << "Domain Error: " << e.what() << '\n';
    return 1;
} catch (std::length_error& e) {
    std::cerr << "Length Error: " << e.what() << '\n';
    return 1;
} catch (std::invalid_argument& e) {
    std::cerr << "Invalid Argument: " << e.what() << '\n';
    return 1;
} catch (std::exception& e) {
    std::cerr << "Error: " << e.what() << '\n';
    return 1;
} catch (...) {
    std::cerr << "Error: " << '\n';
    return 1;
}
```

- Parameter modifiers `const` and `volatile` are ignored
- `catch (...)` matches any type
- `catch (Foo& e)` matches any object of type Foo or any object of any class derived from Foo
- Order matters: Put derived class catch handlers before base class catch handlers

Goal: One `try` per Function

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

bool process(const std::string& arg) {

    try {

        int n = std::stoi(arg);
        f(n);
        g(n);
        h(n);

        return true;

    } catch (std::domain_error& e) {
        std::cerr << "Domain Error: " << e.what() << '\n';
        return false;
    } catch (std::length_error& e) {
        std::cerr << "Length Error: " << e.what() << '\n';
        return false;
    } catch (std::invalid_argument& e) {
        std::cerr << "Invalid Argument: " << e.what() << '\n';
        return false;
    } catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << '\n';
        return false;
    } catch (...) {
        std::cerr << "Error: " << '\n';
        return false;
    }
}
```

function try block

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

bool process(const std::string& arg) try {

    int n = std::stoi(arg);
    f(n);
    g(n);
    h(n);

    return true;

} catch (std::domain_error& e) {
    std::cerr << "Domain Error: " << e.what() << '\n';
    return false;
} catch (std::length_error& e) {
    std::cerr << "Length Error: " << e.what() << '\n';
    return false;
} catch (std::invalid_argument& e) {
    std::cerr << "Invalid Argument: " << e.what() << '\n';
    return false;
} catch (std::exception& e) {
    std::cerr << "Error: " << e.what() << '\n';
    return false;
} catch (...) {
    std::cerr << "Error: " << '\n';
    return false;
}
```

Recommendations

- Throw objects instead of scalar values

  It is possible to throw any type, including `int` and `char*`, but more challenging to determine what the error is

- Throw objects derived from `std::exception`

  That way, `catch` handlers for `std::exception` will catch your `throw`

- Catch by reference, i.e., &

  `const` is thrown away; not concerned about the advantages of const at this point

  Can also catch by pointer, but pointers cause the question: "*Who should free?*"

Constructors

```
class S {
    S(const std::string& arg) try : m(arg, 100) {

    } catch(std::exception& e) {
        std::cerr << "arg=" << arg << " failed: " << e.what() << '\n';
    }
private:
    const std::string m;
};
```

throw;

```
class MyException : public std::exception {
public:
    void addSize(int n);
};

void f() {

    try {
        // ...
    }
    catch (MyException& e) {
        e.addSize(n);
        throw;
    }
}
```

Exception Dispatcher: Re-Throw Idiom

```
void handleException() {
    try {
        throw;
    }
    catch (MyException1& e) {
        // ...code to handle MyException1
    }
    catch (MyException2& e) {
        // ...code to handle MyException2
    }
}

void f() {

    try {

        // ...something that might throw

    } catch (...) {
        handleException();
    }
}
```

`noexcept` specifier

- Current:

  non-throwing - `void f() noexcept;`

  potentially throwing - `void f();`

- Deprecated:

  non-throwing - `void f() throw();`

  potentially throwing - `void f() throw(std::exception);`

Other Issues

- C++ has no `finally`; use RAII instead

- Exception handling may or may not be slow, but shouldn't cause a slowdown of normal processing

- If a framework throws pointers, use pointers (e.g., MFC)

Overall Design

- Inherit from the standard exceptions, e.g., `std::exception` and family
- In general, one `try` block per function. It is a good idea to extract a function if more than one is needed.
- Organize around the logical reason for the exception, not the source (e.g., subsystem)

  E.g., Banking app, Reason: "insufficient funds", Subsystem: One of many
- Design for the entire system

  Avoid a layer-by-layer design

Scenario: Starting Code

```
#include <iostream>

class Class {
public:
    Class(int size) {
        data = new int[size];
        // ...
    }

    ~Class() {
        delete [] data;
    }

private:
    int* data;
};

int main(int argc, char* argv[]) {

    if (argc != 2) {
        std::cerr << "usage: " << argv[0] << " <size>\n";
        return 1;
    }

    int size = std::atoi(argv[1]);

    Class oop(size);

    return 0;
}
```

Scenario: Adding Exception Handling

Problem

Scenario: Summation

- In C++, the destructor cannot be called for a partially-constructed object
- When a constructor throws an exception, the destructor of the class cannot clean up fields/data members; the fields/data members have to clean up themselves
- Every data member should have RAII semantics and not rely on class destructor where they are fields
- True RAII classes have to be careful when they throw exceptions in the constructor