

Error Handling with Exceptions

Sources of Problems

- Program logic.

Errors like exceeding array bounds and division by zero can be prevented by the programmer.

- Environmental (runtime) conditions.

Loss of a network connection or a full disk can be anticipated but not prevented.

Exceptions

- An exception is an object or primitive datum that signals an error condition and provides information about the error.
- When an exception is generated, control is passed up the call stack until a specific handler is found.
- Multiple handlers for different exceptions may be provided at multiple levels.
- In C++, all exceptions can be ignored. Their handling, however can be weakly enforced.

Basic Exceptions

- An exceptional condition is a problem that prevents the continuation of the current method or scope.
- To indicate such a condition, one throws an exception object:

```
if (d == 0)
    throw invalid_argument("Denominator is 0.");
```

Catching Exceptions

A thrown exception will cause immediate transfer from a **try** block to an appropriate **catch** clause.

```
try {  
    Code that might generate exceptions  
    but represents normal, expected behavior }  
catch (derived_class_exception d) {  
    Code to handle exception d }  
catch (base_class_exception b) {  
    Code to handle exception b }  
catch (...) {  
    Code to handle any other kind of exception }
```

The Exception Specification & Uncaught Exceptions

- Any exception thrown by a function may be declared in a **throw** clause. This is an exception specification.
- If there is an exception specification, a thrown exception that is not listed therein will result in a call to **unexpected()**, which by default calls **terminate()**.
- Uncaught exceptions cause a call to **terminate()** which by default calls **abort()**.

Changing the Handlers

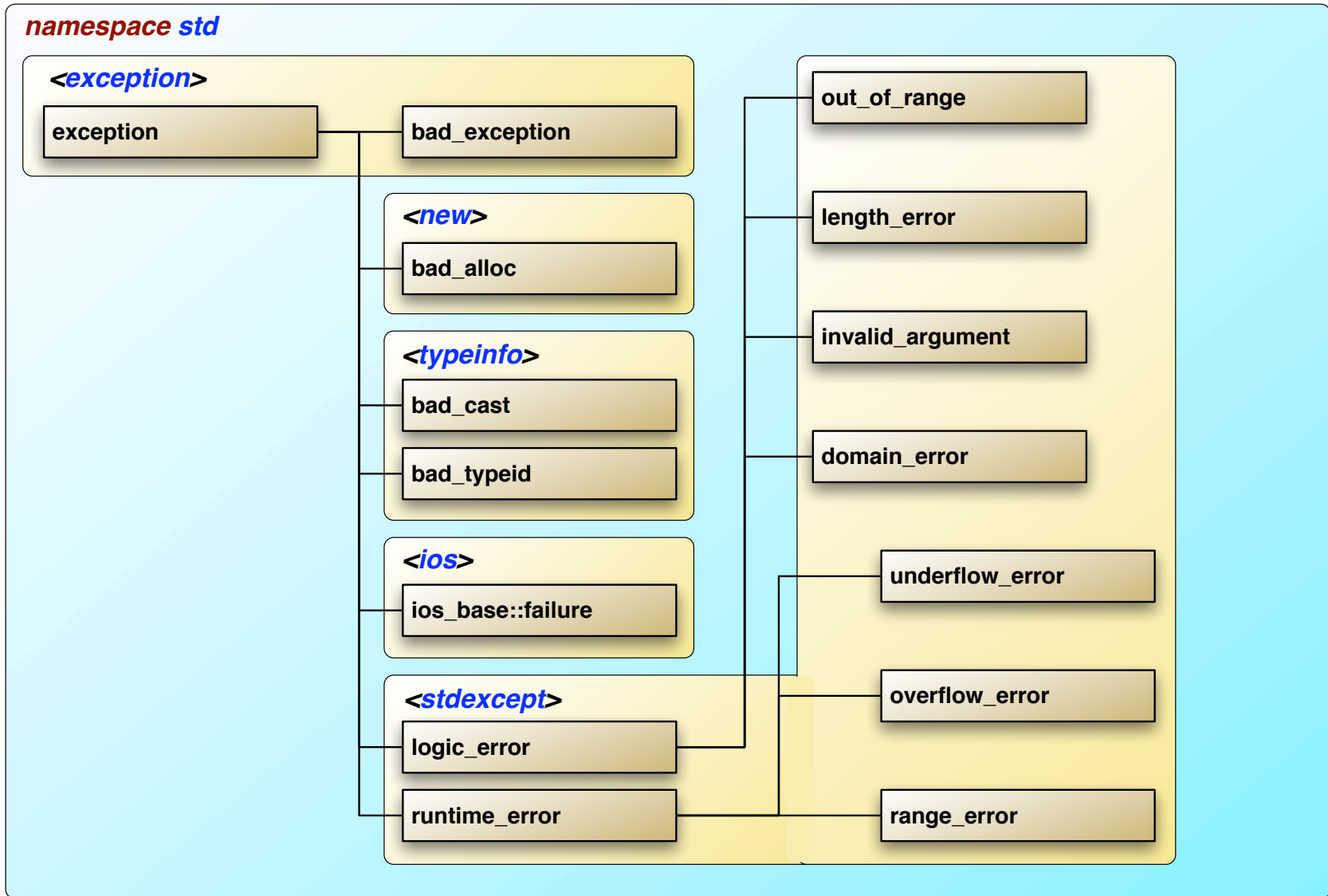
The `set_terminate` function saves `f` to be used by calls to `terminate`. The previous value of the terminate handler is returned.

```
typedef void (*terminate_handler)( );  
terminate_handler set_terminate(terminate_handler f) throw( );
```

The `set_unexpected` function saves `f` to be used by calls to `unexpected`. The previous value of the unexpected handler is returned.

```
typedef void (*unexpected_handler)( );  
unexpected_handler set_unexpected(unexpected_handler f) throw( );
```

Standard Exceptions



The Top exception Hierarchy

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

The Exception Class

The `exception` class is the base class for all exception objects thrown by the standard library or by code generated by the compiler.

- By convention, user-defined exception classes also derive from `exception` or from one of its derived classes.
- `what()` returns a message that describes the nature of the exception. The exact contents of the string are implementation-defined.

The `exception` Subclasses

- The `logic_error` class is a base class for logic-error exceptions. A logic error is a violation of the preconditions or other requirements for a function.
- The `runtime_error` class is the base class for runtime errors, which are errors that cannot reasonably be detected by a static analysis of the code but can be revealed only at runtime.

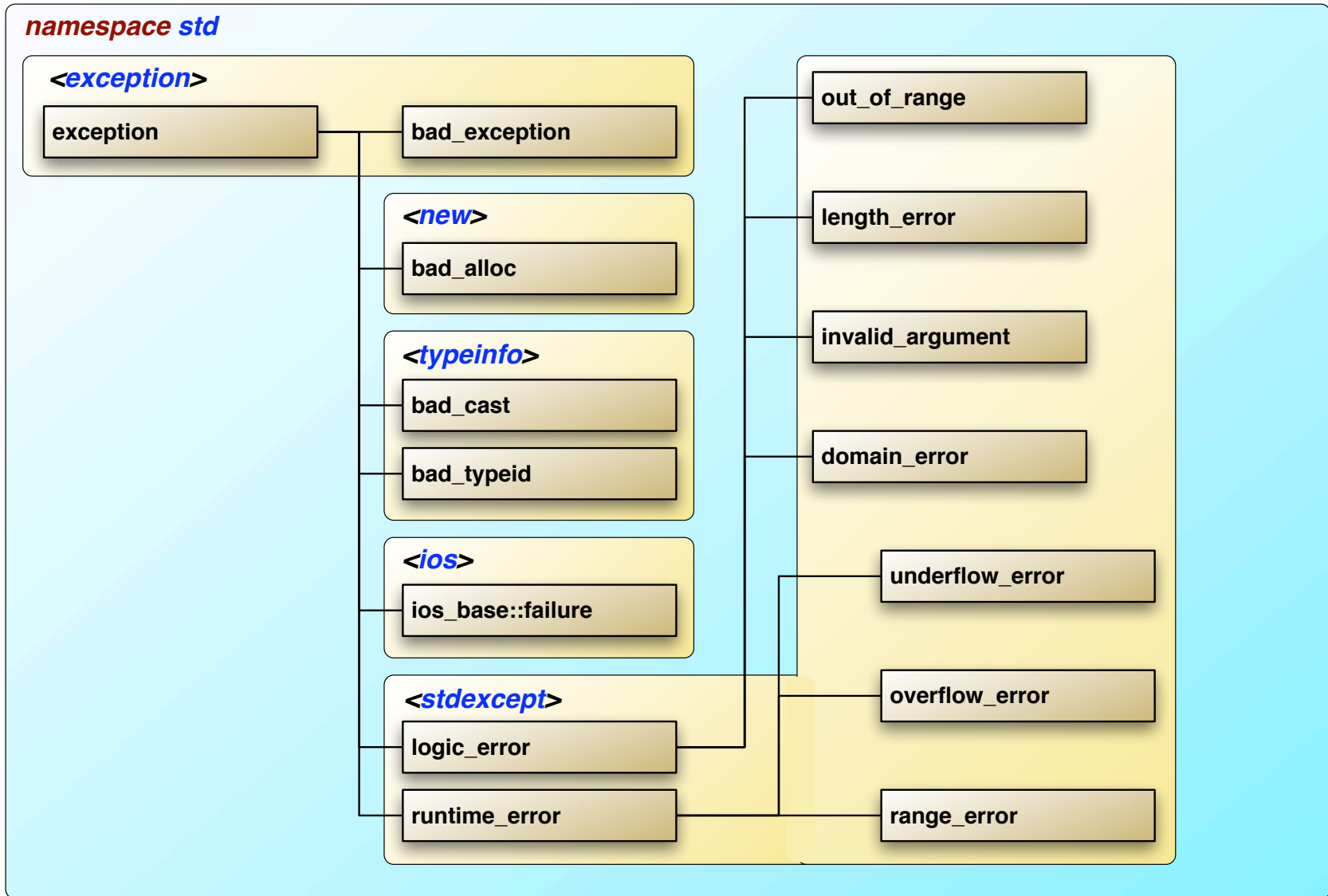
The `logic_error` Subclasses

- The `out_of_range` class is used when an index or similar value is out of its expected or allowed range.
- The `length_error` class is used to attempt to set or change the size of an object that exceeds the maximum size.
- The `invalid_argument` class is thrown to report invalid arguments to functions. Specific kinds of invalid arguments are covered by the other logic errors; use `invalid_argument` for any other situations.
- The `domain_error` class is used to report domain errors, that is, arguments to functions that are outside the valid domain for input to the functions.

The `runtime_error` Subclasses

- The `underflow_error` class can be used for arithmetic underflow. Note that underflow in most arithmetic expressions has undefined behavior.
- The `overflow_error` class can be used for arithmetic overflow.
- The `range_error` class can be used when a function's results would fall outside its valid range. Note that the `<cmath>` functions do not throw any exceptions, but a third-party math library might throw `range_error` for, say, computing a power that exceeds the limits of its return type.

Standard Exceptions



Custom Exceptions

```
class bad_data : public std::out_of_range {  
  public:  
    bad_data(int value, int min, int max)  
      : std::out_of_range(make_what(value, min, max)) {}  
  private:  
    std::string make_what(int value, int min, int max);  
};  
  
std::string bad_data::make_what(int value, int min, int max) {  
  std::ostringstream out;  
  out << "Invalid datum, " << value << ", must be in [" <<  
  min << ", " << max << "];  
  return out.str( );  
}
```

Exception Demonstration

```
// File: exceptions.cpp
// Author: Gary D. Brown
// Date: February 18, 2005
// Version: 1.0
// Purpose: To contrast three approaches to error handling

#define NDEBUG

#include <iostream>
#include <cassert>
#include <stdexcept>

using namespace std;

double quotient(double numerator, double denominator)
    throw (invalid_argument);
// Error Handling: Flexible.
// Precondition: denominator is nonzero.
// Postcondition: numerator is divided by denominator and the quotient
// is returned. If the denominator is zero, either an invalid_argument
// exception is thrown or execution is terminated with a failed
// assertion.
```

Three Approaches

```
void computeWithAssert();  
// Error Handling: OK for development only. Clean style.  
// Precondition: none.  
// Postcondition: After the user is prompted for values of a numerator  
// and denominator the quotient is displayed. If a zero is entered  
// for the denominator, execution is terminated with a failed  
// assertion.  
  
void computeWithErrorChecking();  
// Error Handling: OK for deployment. Clumsy style.  
// Precondition: none.  
// Postcondition: After the user is prompted for values of a numerator  
// and denominator the quotient is displayed. If a zero is entered  
// for the denominator, the user is prompted to try again.  
  
void computeWithExceptionHandling();  
// Error Handling: OK for deployment. Clean style.  
// Precondition: none.  
// Postcondition: After the user is prompted for values of a numerator  
// and denominator the quotient is displayed. If a zero is entered  
// for the denominator, the user is prompted to try again.
```

Assert or Throw

```
int main() {  
    #ifndef NDEBUG  
        computeWithAssert();  
    #else  
        computeWithExceptionHandling();  
    #endif  
    return EXIT_SUCCESS;  
}  
  
double quotient(double n, double d) throw (invalid_argument) {  
    assert(d != 0);  
    if (d == 0)  
        throw invalid_argument("Denominator is 0.");  
    return n/d;  
}
```

Clean and Dangerous

```
void computeWithAssert() {  
    double n, d, q;  
    cout << "Numerator: ";  
    cin >> n;  
    cout << "Denominator: ";  
    cin >> d;  
    q = quotient(n, d);  
    cout << endl << "The quotient is " << q << endl;  
}
```

Clumsy and Effective

```
void computeWithErrorChecking() {
    bool retry;
    do {
        double n, d, q;
        cout << "Numerator: ";
        cin >> n;
        cout << "Denominator: ";
        cin >> d;
        if (d != 0) {
            q = quotient(n, d);
            cout << endl << "The quotient is " << q << endl;
            retry = false; }
        else {
            cout << endl << "Error: Denominator is 0." << endl
                << "Please try again." << endl << endl;
            retry = true; }}
    while (retry);
}
```

Clean and Effective

```
void computeWithExceptionHandling() {
    bool retry;
    do {
        try {
            double n, d, q;
            cout << "Numerator: ";
            cin >> n;
            cout << "Denominator: ";
            cin >> d;
            q = quotient(n, d);
            cout << endl << "The quotient is " << q << endl;
            retry = false; }
        catch (invalid_argument ia) {
            cout << endl << "Error: " << ia.what() << endl
                << "Please try again." << endl << endl;
            retry = true; }}
    while (retry);
}
```