Exceptions

- General way to handle errors
- Error handling without exceptions:
  o Requires a lot of tedious and error prone programming
  o All functions must agree on “error” return values
  o Complicated flow control
  o Must remember to release resources
  o Hard to get out of deeply nested loops
  o Hard to deal with deeply nested function calls
- Constructors errors are especially problematic
  o No return value, so must use flags
  o Must remember to check flags
- Need flexible ways to deal with error conditions
  o Especially for library code

Syntax of C++ Exceptions

- Similar to Java exceptions
- No keyword exception, and no built-in exception type
  o Any object can be thrown as an exception
  o An exception is just a piece of data
- Two perspectives of exceptions
  o Generating the exception (throwing the exception)
  o Dealing with the event of the exception (catching the exception)
- Exceptions are generated with the reserved word throw
  o Indicates change of flow control, just like return
  o Usually requires an expression – the exception to be thrown
    ▪ May be any type of data
  o If no expression, then the current exception is re-thrown
    ▪ Run time error if there is no current exception
  o Other use of the reserved word throw is as an interface specification
    ▪ Indicates that a function will throw a particular exception
    ▪ throw() indicates function will not throw any exception
    ▪ May enclose comma separated list of exception types that will be thrown
    ▪ If another type is thrown, then std::unexpected() called, which terminates
    ▪ Default is that any exception type can be thrown
- Exceptions are handled by a combination of reserved words try and catch
  o The try block defines an area during which we want exceptions to be handled by our catching code
    ▪ try delineates the “normal” code to be executed, not the handling code
  o A try block is followed one or more catch blocks
    ▪ Each catch block indicates the exception type it is catching
    ▪ Exception value is like a function argument – type and identifier
    ▪ Delineates the code to be executed to handle the exception
Control Flow
- Each try block pushes another set of exception handlers on the runtime stack
- When an exception is thrown, the exception type is matched against the most recently pushed handler for that type
  - Inheritance relations hold for the matching of handlers
- If there is no matching handler, the default action is to terminate
- Think of the exception throw-catch as a long jump
- May be local within same function (e.g., throw could be nested inside of try block)
- May be transfer to calling function
- May be very long jump, unwinding an arbitrary number of function calls
- Housekeeping is done along the way
  - Stack frames are popped
    - Destructors for locals called
    - Basically all the housekeeping of an explicit return
    - But also applies to all intervening function calls
    - Exceptions do not violate consistent treatment expected from well defined constructors and destructors
    - Exception thrown from constructor means the object is not constructed (and if partially constructed, e.g., base classes, that work is undone, i.e., base destructors are called)
- No finally clause as in Java

Coding Conventions
- Exceptions thrown by a class can be nested definitions in throwing class
  - Allows name spaces to be simpler
- Use inheritance hierarchies to group related exceptions
  - Virtual functions to provide exception specific information or handling

Exception Example
- Class IntArray is the integer array, now with exception handling
- Throws two types of exceptions
  - OutOfSpace from constructors, assignment
  - OutOfBounds from array operator
  - Both derive from IntArray::Error
- The code:

```cpp
class IntArray {
public:
    IntArray(int size) : len(size) {
        try {
            data = new int[len];
        } catch(const exception &e) { throw OutOfSpace(len); }
    } 
    IntArray(const IntArray &a) : len(a.len) {
        try {
            data = new int[len];
        } catch(const exception &e) { throw OutOfSpace(len); }
        for (int i = 0; i < len; ++i) data[i] = a.data[i];
    } 
};
```
virtual int & operator [] (int i) { 
    if (i < 0 || i >= len) throw OutOfBounds(i, len); 
    else return data[i]; 
} 

virtual int operator [] (int i) const { 
    if (i < 0 || i >= len) throw OutOfBounds(i, len); 
    else return data[i]; 
} 

// Assignment (sizes need not match) 
IntArray & operator = (const IntArray & a) { 
    if (this == &a) return *this; 
    try { 
        delete [] data; data = new int[len = a.len]; 
    } catch(const exception &) { throw OutOfSpace(len); } 
    for (int i = 0; i < len; ++i) data[i] = a.data[i]; 
    return *this; 
} 

... 

public: 
// Exceptions 
class Error { 
public: 
    Error(int n) : _size(n) { } 
    virtual ~Error() { } 
    virtual void print(ostream & o) const = 0; 
    int size() const { return _size; } 
private: 
    const int _size; 
}; 

class OutOfSpace : public Error { 
public: 
    OutOfSpace(int n) : Error(n) { } 
    void print(ostream & o) const { 
        o << "IntArray out of space, size " << size(); 
    } 
private: 
}; 

class OutOfBounds : public Error { 
public: 
    OutOfBounds(int i, int n) : Error(n), _index(i) { } 
    void print(ostream & o) const { 
        o << "IntArray out of bounds access, index " << _index << ", size " << size(); 
    } 
private: 
    const int _index; 
};
inline
ostream & operator << (ostream & o, const IntArray::Error & e) {
    e.print(o);
    return o;
}

- And here's some of the driver code:

    try {
        IntArray a(size);
        for (int i = 0; i < a.length(); ++i)
            a[i] = i;
        IntArray b(2*size);
        for (int i = 0; i < b.length(); ++i)
            b[i] = 2*i + 1;
        a = b;
        a[index] = -1;
    } catch (const IntArray::Error & err) {
        // Report the caught exception
        cout << "* Caught array error: " << err << endl;
    }