

Exceptions

Object-Oriented Programming with C++

Run-time error

- The basic philosophy of C++ is that "badly formed code will not be run."
- There's always something happens in run-time.
- It is very important to deal with all possible situation in the future running.

Read a file

open the file;

determine its size;

allocate that much memory;

read the file into memory;

close the file;

```
errorCodeType readFile {  
    initialize errorCode = 0;  
    open the file;  
    if ( theFilesOpen ) {  
        determine its size;  
        if ( gotTheFileLength ) {  
            allocate that much memory;  
            if ( gotEnoughMemory ) {  
                read the file into memory;  
                if ( readFailed ) {  
                    errorCode = -1;  
                }  
            } else {  
                errorCode = -2;  
            }  
        } else {  
            errorCode = -3;  
        }  
        close the file;  
        if ( theFILEDidntClose && errorCode == 0 ) {  
            errorCode = -4;  
        }  
    } else {  
        errorCode = -5;  
    }  
    return errorCode;  
}
```

Working with exception

```
try {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
} catch ( fileOpenFailed ) {  
    doSomething;  
}  
} catch ( sizeDeterminationFailed ) {  
    doSomething;  
}  
} catch ( memoryAllocationFailed ) {  
    doSomething;  
}  
} catch ( readFailed ) {  
    doSomething;  
}  
} catch ( fileCloseFailed ) {  
    doSomething;  
}  
}
```

Why exception?

- The significant benefit of exceptions is that they clean up error handling code.
- It separates the code that describes what you want to do from the code that is executed.

Example: Vector

```
template <class T> class Vector {  
private:  
    T* m_elements;  
    int m_size;  
public:  
    Vector (int size = 0) : m_size(size)  
    { ... }  
    ~Vector () { delete [] m_elements; }  
    void length(int);  
    int length() { return m_size; }  
    T& operator[](int);  
};
```

Problem

```
template <class T>  
T& Vector<T>::operator[] (int idx) {
```


Problem

```
template <class T>  
T& Vector<T>::operator[] (int idx) {
```

What should the [] operator do if the index is not valid?

Problem

```
template <class T>  
T& Vector<T>::operator[] (int idx) {
```

What should the [] operator do if the index is not valid?

1.) Return random memory object

```
return m_elements[idx];
```

More choices

2.) Return a special error value

```
if (idx < 0 || idx >= m_size) {  
    T error_marker("some magic value");  
    return error_marker;  
}  
return m_elements[idx];
```

This throws the baby out with the bath water!

```
x = v[2] + v[4]; // not safe code!
```

More choices ...

More choices ...

3.) Just die!

```
if (idx < 0 || idx >= m_size) {  
    exit(22);  
}  
return m_elements[idx];
```

4.) Die *gracefully* (with *autopsy*!)

```
assert(idx >= 0 && idx < m_size);  
return m_elements[idx];
```

When to use exceptions

- Many times, you don't know what should be done
- If you do *anything* you'll be wrong
- Solution: expose the problem

Make your caller (or its caller ...) responsible

How to raise an exception

```
template <class T>
T& Vector<T>::operator[] (int idx) {
    if (idx < 0 || idx >= m_size) {
        // throw is a keyword
        // exception is raised at this point
        throw <<something>>;
    }
    return m_elements[idx];
}
```

What do you throw?

// What do you have? Data!

// Define a class to represent the error

```
class VectorIndexError {  
public:  
    VectorIndexError(int v) : m_badValue(v) { }  
    ~VectorIndexError() { }  
    void diagnostic() {  
        cerr << "index " << m_badValue  
            << "out of range!"; }  
private:  
    int m_badValue;  
};
```


How to raise an exception

```
template <class T>
T& Vector<T>::operator[] (int idx) {
    if (idx < 0 || idx >= m_size) {
        throw VectorIndexError(idx);
    }
    return m_elements[idx];
}
```

What about your caller?

Case I) Doesn't care

–Code never even suspects a problem

```
int func() {  
    Vector<int> v(12);  
    v[3] = 5;  
    int i = v[42]; // out of range  
    // control never gets here!  
    return i * 5;  
}
```

What about your caller?

Case 2) Cares deeply

```
void outer() {  
    try {  
        func();  
        func2();  
    } catch (VectorIndexError& e) {  
        e.diagnostic();  
        // This exception does not propagate  
    }  
    cout << "Control is here after exception";  
}
```

What about your caller?

Case 3) Mildly interested

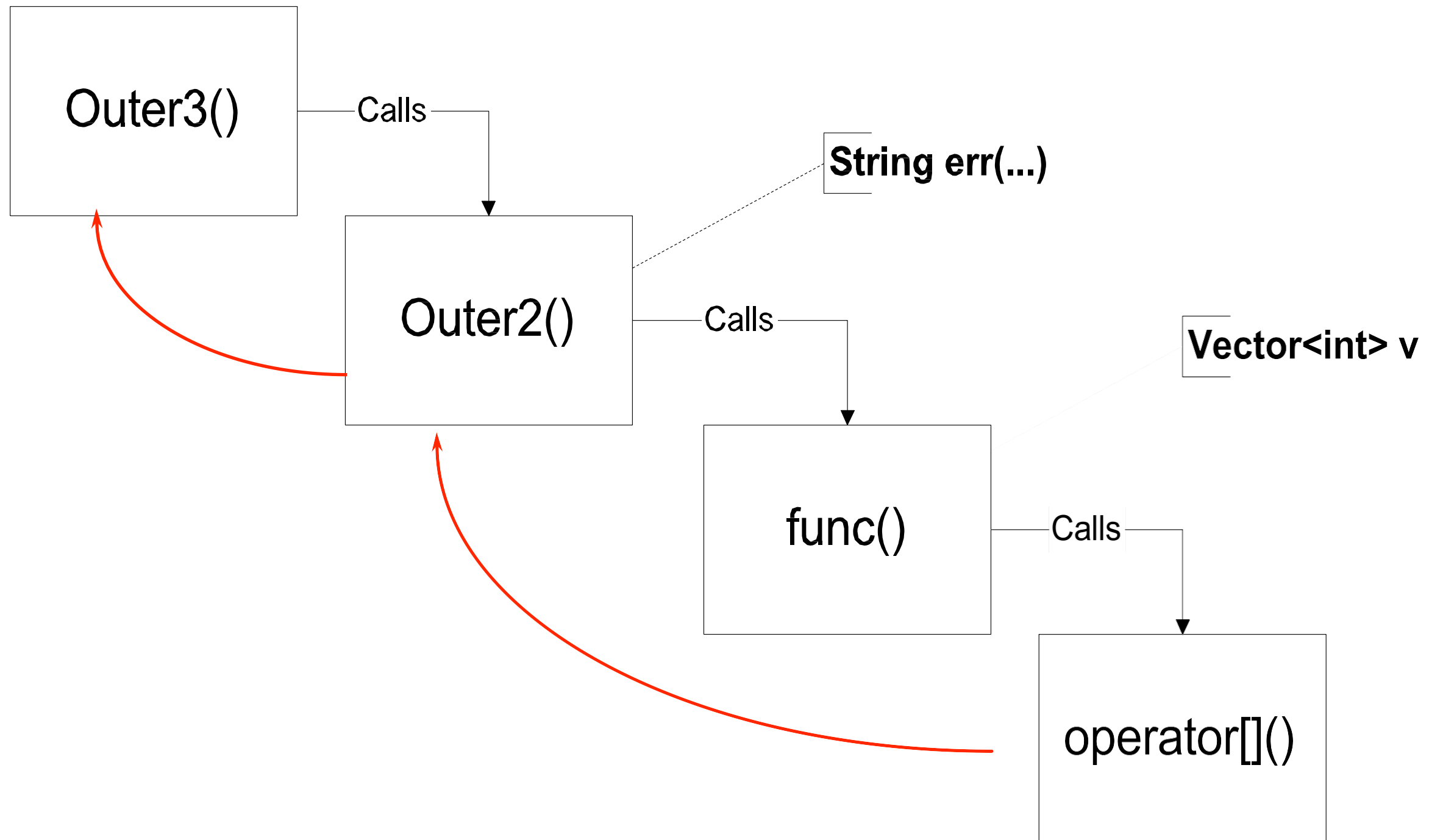
```
void outer2() {  
    String err("exception caught");  
    try {  
        func();  
    } catch (VectorIndexError) {  
        cout << err;  
        throw; // propagate the exception  
    }  
}
```

What about your caller?

Case 4) Doesn't care about the particulars

```
void outer3 () {  
    try {  
        outer2 ();  
    } catch (...) {  
        // ... catches ALL exceptions!  
        cout << "The exception stops here!";  
    }  
}
```

What happened?



Review

- Throw statement **raises** the exception
 - Control propagates back to first handler for that exception
 - Propagation follows the **call** chain
 - Objects on **stack** are properly destroyed
- `throw exp;`
 - throws value for matching
- `throw;`
 - **re-raises** the exception being handled
 - valid only within a handler

Try blocks

- Try block

```
try { ... }  
catch ...  
catch ...
```

- Establishes any number of handlers
- Not needed if you don't use *any* handlers
- Shows where you expect to handle exceptions
- Costs cycles

Exception handlers

- Select exception by type
- Can re-raise exceptions
- Two forms

```
catch (SomeType v) { // handler code  
}  
catch (...) { // handler code  
}
```

- Take a single argument (like a formal parameter)

Selecting a handler

- Can have any number of handlers
- Handlers are checked in order of appearance
 1. Check for exact match
 2. Apply base class conversions
Reference and pointer types, only
 3. Catch-all handler (...)

Inheritance can be used to structure exceptions

Example: using inheritance

- Hierarchy of exception types

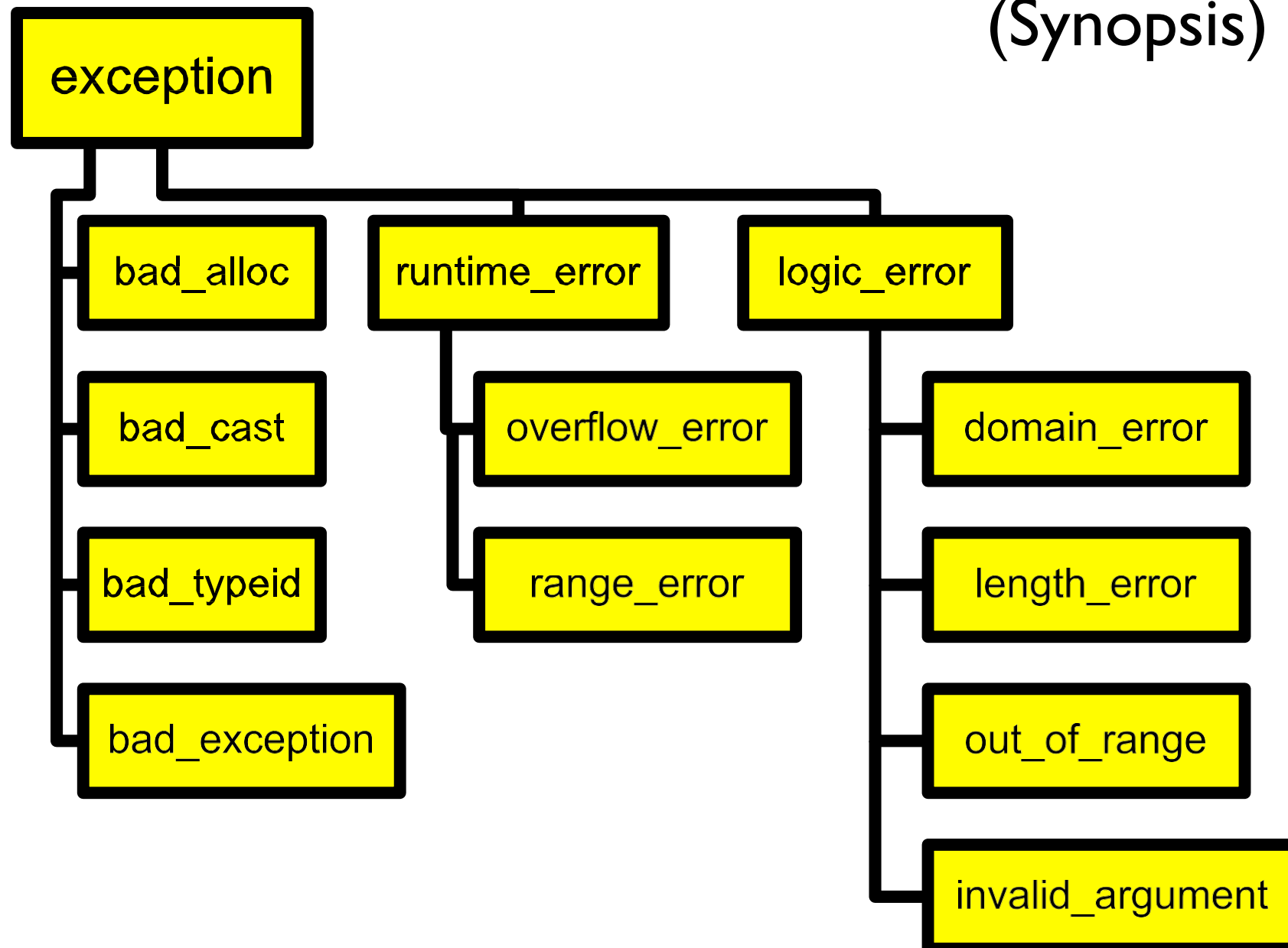
```
class MathErr {  
    ...  
    virtual void diagnostic();  
};  
  
class OverflowErr : public MathErr { ... }  
  
class UnderflowErr : public MathErr { ... }  
  
class ZeroDivideErr : public MathErr { ... }
```

Using handlers

```
try {  
    // code to exercise math options  
    throw UnderFlowErr();  
} catch (ZeroDivideErr& e) {  
    // handle zero divide case  
} catch (MathErr& e) {  
    // handle other math errors  
} catch (UnderFlowErr& e) {  
    // handle underflow errors  
} catch (...) {  
    // any other exceptions  
}
```

Standard library exceptions

(Synopsis)



Exceptions and new

- `new` does NOT returned 0 on failure
- `new` raises a `bad_alloc()` exception

```
void func() {  
    try {  
        while(1) {  
            char *p = new char[10000];  
        }  
    } catch (std::bad_alloc& e) {  
        // ...  
    }  
}
```

Exception specifications

- Specifies whether a function could throw exceptions.
- Part of function type, but not part of signature.

```
void abc(int a) noexcept {  
    ...  
}
```

- May not be checked at compile time, but utilized by the compiler to enable certain optimizations.
- At run time,
 - if an exception is thrown out, the `std::terminate` is called.

Design considerations

- Exceptions should indicate errors
- Here is an inappropriate use:

```
try {  
    for (;;) {  
        p = list.next()  
        // ...  
    } catch (List::end_of_list) {  
        // handle end of list here  
    }
```


Design considerations...

- Don't use exceptions in place of good design

```
void func() {  
    File f;  
    if (f.open("somefile")) {  
        try {  
            // work with f  
        } catch (...) {  
            f.close()  
        }  
    }  
}
```

- This is a good place to use the destructor

```
void func() {  
    File f("some file");  
    // assume destructor closes f  
    // will still be closed if exception  
    // is raised!  
    if (f.ok()) {  
        }  
}
```

Summary

Summary

- Error recovery is a hard design problem
- All subsystems need help from their clients to handle exceptional cases

Summary

- Error recovery is a hard design problem
- All subsystems need help from their clients to handle exceptional cases
- Exceptions provide the mechanism
 - Propagated dynamically
 - Objects on stack destroyed properly

More exceptions

- Exceptions and constructors
- Exceptions and destructors
- Design and usage with exceptions
- Handlers

Failure in constructors

- No return value is possible
- Use an “uninitialized flag”
- Defer work to an `init()` function

Better: Throw an exception

Failure in constructors...

If your constructor throws an exception:

- Dtors for the object *won't* be called.
- Manually clean up allocated resources before throwing, otherwise memory leak happens.

demo

Two stages construction

- Do normal work in ctor
 - Initialize all member objects
 - Initialize all primitive members
 - Initialize all pointers to 0
 - NEVER request any resource
 - File
 - Network connection
 - Memory
- Do additional initialization work in `Init()`

Using smart pointers

- `std::unique_ptr`
- `std::shared_ptr`
- ...
- The destructor will delete the native pointer when it dies.

Exceptions and destructors

Destructors are called when:

- Normal call ended: object exits from scope
- Exceptions throwed: “stack unwinding” invokes dtors on objects as they exit from scope.

What happens if an exception is thrown in a destructor?

Exceptions and destructors...

Throwing an exception in a destructor that is itself being called as the result of an exception will invoke `std::terminate()`.

- Allowing exceptions to escape from destructors should be avoided, never throw it!

Programming with exceptions

Prefer catching exceptions by reference

- throwing/catching by value involves slicing:

```
struct X {};  
struct Y : public X {};  
try {  
    throw Y();  
} catch(X x) {  
    // was it X or Y?  
}
```

Programming with exceptions...

- throwing/catching by pointer introduces coupling between normal and handler code:

```
try {  
    throw new Y();  
} catch(Y* p) {  
    // whoops, forgot to delete..  
}
```

Catch exceptions by reference:

```
struct B {  
    virtual void print() { /* ... */ }  
};  
struct D : public B { /* ... */ };  
  
try {  
    throw D("D error");  
}  
catch(B& b) {  
    b.print(); // print D's error.  
}
```

Exceptions wrap-up

Exceptions wrap-up

- Develop an error-handling strategy early in design.

Exceptions wrap-up

- Develop an error-handling strategy early in design.
- Avoid over-use of try/catch blocks. Use objects to acquire/release resources.

Exceptions wrap-up

- Develop an error-handling strategy early in design.
- Avoid over-use of try/catch blocks. Use objects to acquire/release resources.
- Don't use exceptions where local control structures would suffice.

Exceptions wrap-up

- Develop an error-handling strategy early in design.
- Avoid over-use of try/catch blocks. Use objects to acquire/release resources.
- Don't use exceptions where local control structures would suffice.
- Not every function can handle every error.

Exceptions wrap-up...

Exceptions wrap-up...

- Use exception-specifications for major interfaces.

Exceptions wrap-up...

- Use exception-specifications for major interfaces.
- Library code should not decide to terminate a program. Throw exceptions and let the caller decide.

Uncaught exceptions

- If an exception is thrown but not caught `std::terminate()` will be called.
- The `std::terminate()` can also be intercepted.

```
void my_terminate() {  
    /* ... */  
}  
set_terminate(my_terminate);
```


Write exception-safe code





Write exception-safe code

```
class BankAccount {  
    // ...  
  
    void withdrawMoney(int amount) {  
        // ...  
  
        reduceBalance(amount);  
        prepareCash();  
        releaseCash();  
  
        // ...  
    }  
  
    // ...  
};
```

Write exception-safe code

```
class BankAccount {  
    // ...  
  
    void withdrawMoney(int amount) {  
        // ...  
  
        reduceBalance(amount);  
        prepareCash();           ← Throws an exception  
        releaseCash();  
  
        // ...  
    }  
  
    // ...  
};
```

Write exception-safe code

```
class BankAccount {  
    // ...  
  
    void withdrawMoney(int amount) {  
        // ...  
  
        reduceBalance(amount) ;  Balance already reduced...  
        prepareCash() ;  Throws an exception  
        releaseCash() ;  
  
        // ...  
    }  
  
    // ...  
};
```