Exceptions

• Problem: dealing with errors is a pain, even in a simple language like C.
• It’s even worse in C++, where some obvious approaches don’t work.
• Exceptions provide a new mechanism for dealing with errors.
• Make use of inheritance to make code more modular.
• Fundamentally a control construct
• “Unit of recovery” == “function”

Exceptions

• A function which detects an error might be able to deal with it, or might not.
• If not, it should notify someone who can. Obvious candidates are callers of the function.
• How do we notify callers of an error?
• If A calls B calls C calls D and only A can deal with the error detected by D, can we avoid involving B and C in the communication?
• What form should the communication take?
Exceptions

• A function which detects an error can throw an exception.
• A function which can deal with errors can catch an exception.
• When an exception is thrown, control “travels” up the call stack, “looking” for a function that can catch it.
• If none is found, some default behavior is invoked (typically termination)

What’s an exception?

• An object of any copy-able type can function as an exception.
• Avoid using native types
• Often new classes are created specifically for exceptions.
• The STL provides several exception types.
Throwing exceptions

struct AcctError
{
    AcctError( std::string p ) : problem(p) {}
    std::string problem;
};

bool Account::withdraw( unsigned int amount )
{
    if ( amount > m_cashBalance )
    {
        throw AcctError();
    }
    ...
}
Catching exceptions

```cpp
main()
{
    Account acct1;
    acct1.deposit(100000);

    try
    {
        acct1.withdraw(2000000);
    }
    catch (AcctError& e)
    {
        cout << "Error: " << e.problem << endl;
    }
    ...
}
```

throw syntax and semantics

- A throw statement is syntactically like a return statement.
- Execution leaves the function that throws.
- Local objects are all destroyed (like a return).
- Stack is “unwound,” destroying local objects in each frame until catch is found.
- Execution resumes with first statement of catch.
- When catch finished, execution continues with statements after catch.
try-catch syntax

• A try block must use {}s and is a scope.
• A try block must be followed by one or more catch blocks.
• A catch block must use {}s and is a scope.
• Each catch block must have a declaration list
• Declaration lists declare one type, with an optional identifier.
• The catch block can treat the identifier like a local variable.
• Catch block identifiers shadow other local vars.

try-catch syntax in constructors

• In constructors, the initializer list may be calling functions which can throw:

```cpp
Account::Account()
try
 : m_balance(might_throw())
 {
 ... 
 }
catch(std::exception)
 {
 // do something else (and don’t forget the 
 // rest of the constructor body here!
 }
```
Why not throw ints or enums?

```cpp
#define INSUFF_FUNDS 1
enum AcctError { InsuffFunds, InsuffShares };

bool Account::withdraw( unsigned int amount )
{
    if ( amount > m_cashBalance )
    {
        /* throw INSUFF_FUNDS; // blech */
        throw InsuffFunds; // better, but still bad
    }
    ...
}

try ...
/* catch ( int err ) */
catch ( AcctError err )
```

Different kinds of exceptions

```cpp
struct InsuffFunds
{
    InsuffFunds( int b, int r ):balance(b), request(r) {}
    int balance;
    int request;
};

struct InsuffShares
{
    InsuffShares( const Market::Symbol& sym, int num ):
        name(sym), nShares(num) {}
    Market::Symbol name;
    int nShares;
};
```
Catching a specific exception

```c++
main()
{
    ...
    try
    {
        // try something that could generate an exception
        acct1.withdraw(2000000);
    }
    catch (InsuffFunds f)
    {
        cout << "Account doesn't have enough cash for $" << f.request << " withdrawal." << endl;
    }
    ...
}
```

Catching more than one exception

```c++
main()
{
    ...
    try
    {
        // try something that could generate any one
        // of several exceptions
        acct1.complex_operation();
    }
    catch (InsuffShares e)
    {
        cout << ...
    }
    catch (InsuffFunds e)
    {
        cout << ...
    }
    ...
}
```
Using Inheritance

```cpp
struct AcctError {};
struct InsuffFunds : public AcctError
{
    InsuffFunds( int b, int r ):balance(b), request(r) {}
    int balance;
    int request;
};

struct InsuffShares : public AcctError
{
    InsuffShares( const Market::Symbol& sym, int num ):
    name(sym), nShares(num) {}
    Market::Symbol name;
    int nShares;
};
```

Catching base classes

```cpp
main()
{
    ...
    try
    {
        // try something that could generate any one
        // of several exceptions
        acct1.complex_operation();
    }
    // Catches AcctError or anything derived from it
    catch (AcctError& e )
    {
        cout << "...
    }
    ...
```
Virtual functions and exceptions

```cpp
struct AcctError
{
    virtual std::string description() = 0;
};

struct InsuffFunds : public AcctErr
{
    InsuffFunds( int b, int r ):balance(b), request(r) {}
    int balance;
    int request;
    virtual std::string description();
};

// likewise for InsuffShares
```

Catching base classes – pt. 2

```cpp
main()
{
    ...
    try
    {
        // try something that could generate any one
        // of several exceptions
        acct1.complex_operation();
    }
    // Catches AcctError or anything derived from it
    catch (AcctError& e )
    {
        cout << e.description() << endl;
    }
    ...
```
Rethrowing exceptions

```cpp
void Account::complex_operation()
{
    ...
    int* my_temp_array = new int[100];
    try
    {
        acct1.withdraw( 100000 );
    }
    // Catch the exception
    catch ( AcctError& e )
    {
        delete[] my_temp_array;
        // We can’t really handle this, so kick it up
        throw;
    }
    ...
}
```

Catching all exceptions

```cpp
void foo()
{
    ...
    int* my_temp_array = new int[100];
    try
    {
        // try something that could generate many
        // different exceptions
        acct1.complex_operation();
    }
    // Catch *any* exception
    catch (...)
    {
        delete[] my_temp_array;
        throw;
    }
    ...
}
Exception Specifications

• Makes exception behavior a part of function declaration.
• Keeps clients from having to read implementation (or doc).
• Compiler will try hard to hold you to your promises
  • … but not across compilation unit boundaries.

```cpp
void foo(); // might throw anything
define foo() throw(std::bad_alloc); // might throw 1 type
void foo() throw(); // must not throw at all
```

std::exception

```cpp
class exception {
class logic_error : public exception {
class bad_alloc : public exception {
```
The STL exception hierarchy

```cpp
#include<stdexcept>

exception
    bad_alloc
    bad_exception
    bad_cast
    bad_typeid
    ios_base::failure
    logic_error
        length_error
        domain_error
        out_of_range
        invalid_argument
    runtime_error
        range_error
        overflow_error
        underflow_error
```