Chapter 16: Exception Handling

Instructor: Mark Edmonds

edmonds_mark@smc.edu

Exceptions

- Exceptions are a simple concept, but a powerful one.
- So far, if our program has runtime problem (error), we have no way to handle or correct it.
  - Imagine if every program you ran immediately crashed upon a problem (e.g. the internet was not connect, a hard drive was removed, etc). Very hard to use a computer!
  - You may remember older programs that would crash and say "Program exited with code 47" (or some other code) without providing much detail.
    * These were unhandled exceptions, and the program crashing was the way to "fix" the problem by not allowing more problems to occur in a bad program state.
- In the programs we've written, you can imagine having issues in a number of ways:
  - A user could pass the wrong parameters to a function
  - Data files that need to be opened for reading or writing could not exist
  - …just about anything you can imagine could go wrong, may go wrong

BankAccount Exceptions

- Suppose we want to use the + operator to add two `BankAccount`s together.
- This only makes sense if the accounts are owned by the same person
- But what if the user tries to add two bank accounts together that belong to different people?

```
BankAccount operator+ (const BankAccount& b1, const BankAccount& b2){
  BankAccount result;
  if (b1.my_Name == b2.my_Name) {
    result = BankAccount(b1.my_Name, b1.my_Balance + b2.my_Balance);
  }
  return result;
}
```

- If the user passes two bank accounts that match as arguments, this works great
- If the user passes two bank accounts that don't match as arguments, this doesn't work well
  - We return an uninitialized bank account, but is that the behavior we really want?
  - How can the user tell whether or not the operation (adding two bank accounts) succeeded?
    * What if both bank accounts were empty…?

- This is problematic, because it excepts the user to be able to interpret a default-initialized bank account as an error
    - The function still returns a value when we really encountered an error - probably not the behavior we want
    - What if we could inform the user of an error in a different way, that didn't require a special interpretation of an otherwise "normal" execution during an error?
        * This is what exceptions are for!

Caller-Callee Relationship revisited

- Remember our Caller-Callee relationships for functions:

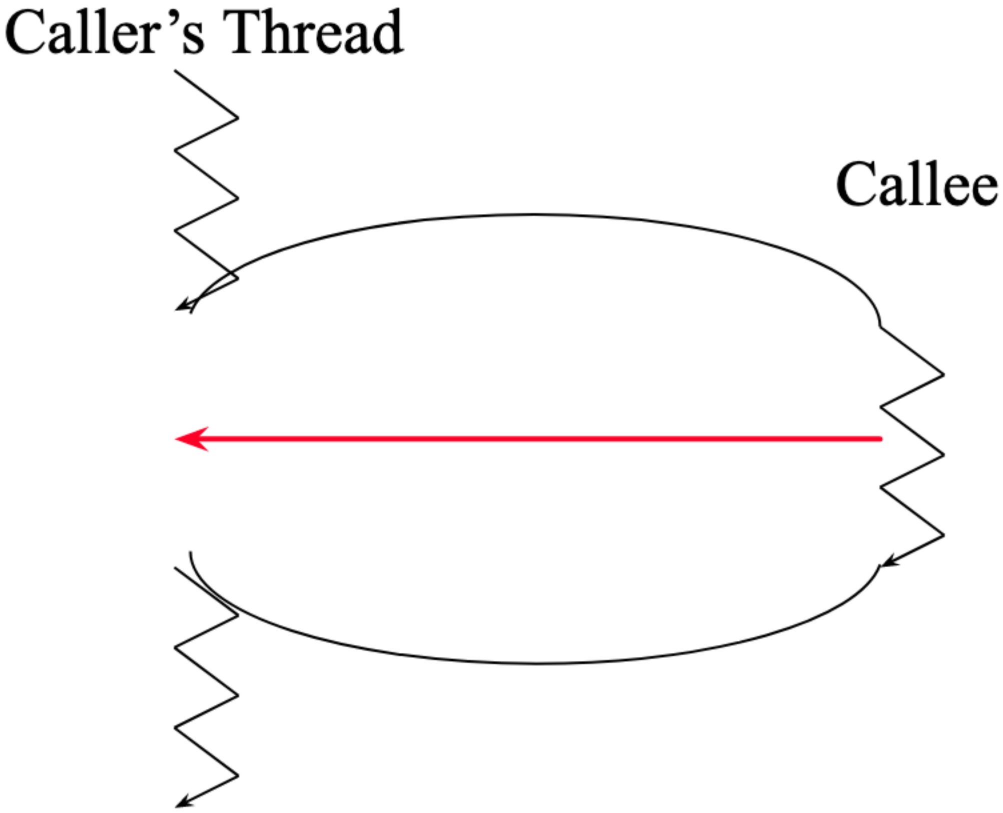

Figure 1: Caller-callee relationship

- The red line indicates that we could return to the Caller function during the Callee's execution if we encounter an error
- An exception is an "alternate" return mechanism to designate an error

- The caller then must handle the exception some way, or the program will crash
- Sending an exception to the caller is called "throwing" an exception
- Receiving and handling the exception in the caller is called "catching" an exception
- So the Callee can throw an exception, and if the Caller doesn't catch the exception, then the program crashes
  - The analogy is like playing catch with a ball, except if the ball is dropped, the program crashes.

Throwing Exceptions

- To throw an exception, we'll use the **`throw`** statement:

```
throw(std::logic_error("Always write a description of the problem as
    the argument to the logic_error constructor"))
```

- This is like a return statement, in the sense we "pass" a value back to the caller
- `std::logic_error` is a class
  - `#include <stdexcept>` to use it
  - We'll eventually learn how to write our own exceptions, but for now, we can use the ones defined by the Standard Library

Catching Exceptions

- The caller needs to **`try`** `{ }` to execute some code that may produce an error and **`catch`** `(){ }` any errors that occur
  - You can have as many **`catch`** statements as necessary (meaning you can put multiple, similar to multiple **`else if`** statements)

```
try {
  // execute code that could throw an exception inside of a "try" block
  some_function_that_may_throw_a_logic_error();
} catch (std::logic_error e) {
  // catch the exception, and do some error recovery procedure.
  // In this case, we just print out the exception message
  cout << e.what() << endl; // e.what() will return the message
      associated with the exception
}
```

Example: `BankAccount` with exceptions

- This example shows how to throw and use exceptions to process potentially invalid data in a loop.
- To cause an exception to be thrown, do the following:
    1. *Create* an account
    2. *Deposit* or *Withdraw* and use a different name than the name you used when you created the account

Example: `ExceptionBankAccount.h`

```
//------------------------------------------------------------------
// INTERFACE FILE: baccount.h
//
// Defines class BankAccount
//
//-------------------------------------------------------------------
// SAFEGUARDS AND INCLUDES
#ifndef BANKACCOUNT_H   // Avoid redeclaring class BankAccount.
#define BANKACCOUNT_H   // This code is compiled only once
#include <string>    // for class string

namespace cs52 {

///////////////////////////////////////////
/////// class BankAccount defintion ///////
///////////////////////////////////////////

class BankAccount {
public:  // class member functions

//--constructors
    BankAccount();

    BankAccount(std::string initName, double initBalance);
    // post: A BankAccount with two arguments when called like this:
    //       BankAccount anAcct("Hall", 100.00);

//--modifiers

    void deposit(double depositAmount);
    // post: depositAmount is credited to this object's balance

    void withdraw(double withdrawalAmount);
    // post: withdrawalAmount is debited from this object's balance

//--accessors

    double balance() const;
    // post: return this account's current balance

    std::string name() const;
    // post return the account name

    void setName( std::string initName );
    // post updates the member variable my_name

    // ADDED CODE BEGINS HERE
    friend std::ostream& operator << ( std::ostream& outs, const BankAccount& b );
    friend std::istream& operator >> ( std::istream& ins, BankAccount& b );
    friend BankAccount operator + ( const BankAccount& left, const BankAccount& right );
    friend BankAccount operator - ( const BankAccount& left, const BankAccount& right );
    friend bool operator ==( const BankAccount& left, const BankAccount& right );
    friend bool operator < ( const BankAccount& left, const BankAccount& right );
    friend bool operator > ( const BankAccount& left, const BankAccount& right );

private:
    std::string my_name;     // Uniquely identify an object
    double my_balance; // Store the current balance (non-persistent)
};

}

#endif   // ifndef BANKACCOUNT_H
```

Example: `ExceptionBankAccount.cpp`

```cpp
//--------------------------------------------------------------------
// IMPLEMENTATION FILE: baccount.cpp
//
// Implements 1. class BankAccount member functions
//
//--------------------------------------------------------------------
#include "ExceptionBankAccount.h"  // allows for separate compilation if you want
#include <iostream>    // for ostream << and istream >>
#include <string>      // for class string
#include <stdexcept>   // supports Linux exception classes

using namespace std;

namespace cs52 {

//--constructors

BankAccount::BankAccount()
{
  my_name = "?name?";
  my_balance = 0.0;
}

BankAccount::BankAccount(string initName, double initBalance)
{
  my_name = initName;
  my_balance = initBalance;
}

//--modifiers

void BankAccount::deposit(double depositAmount)
{
  my_balance = my_balance + depositAmount;
}

void BankAccount::withdraw(double withdrawalAmount)
{
  my_balance = my_balance - withdrawalAmount;
}

```

```
//--accessors

double BankAccount::balance() const
{
  return my_balance;
}

string BankAccount::name() const
{
  return my_name;
}

void BankAccount::setName( string initName )
{
  my_name = initName;
}


// NEW CODE STARTS HERE
std::ostream& operator << ( std::ostream& outs, const BankAccount& b ) {
    outs << b.my_name << " " << b.my_balance << endl;
    return( outs );
}

std::istream& operator >> ( std::istream& ins, BankAccount& b ) {
    ins >> b.my_name >> b.my_balance;
    return( ins );
}

BankAccount operator + ( const BankAccount& left, const BankAccount& right ) {
    BankAccount newB;
    if (left.my_name == right.my_name) {
        newB.deposit( left.my_balance );
        newB.deposit( right.my_balance );
    }
    else {
        cerr << "YIKES!  These two accounts can't be added together since the names differ!" << endl;
        throw logic_error( "Bad account names" );
    }
    return( newB );
}

BankAccount operator - ( const BankAccount& left, const BankAccount& right ) {
    BankAccount newB;
    if (left.my_name == right.my_name) {
        newB.deposit( left.my_balance );
        newB.withdraw( right.my_balance );
    }
    else {
        cerr << "YIKES!  These two accounts can't be subtracted together since the names differ!" << endl;
        throw logic_error( "Bad account names" );
    }
    return( newB );
}

bool operator ==( const BankAccount& left, const BankAccount& right ) {
    return( (left.my_balance == right.my_balance) && (left.my_name == right.my_name)  );
}

bool operator < ( const BankAccount& left, const BankAccount& right ) {
    return( left.my_balance < right.my_balance );
}

bool operator > ( const BankAccount& left, const BankAccount& right ) {
    return( left.my_balance > right.my_balance );
}

}
```

Example: ExceptionBanker.cpp

```
// This program demonstrates how to make use of existing objects.
// This program uses a BankAccount class with the interface described
// in class.

#include <iostream>                // for std::cout
#include <string>                            // for string class
#include "ExceptionBankAccount.h"              // for BankAccount class
#include <stdexcept>               // supports Linux exceptions
```

```

using namespace std;                // supports cout
using namespace cs52;                         // for BankAccount class

enum CHOICE { CREATE, DEPOSIT, WITHDRAW, PRINT, QUIT };

CHOICE menu();

int main( )
{
  CHOICE choice;
  BankAccount account, withdrawaccount, depositaccount;
  string name;
  double balance;

  cout.setf( ios::fixed );
  cout.setf( ios::showpoint );
  cout.precision( 2 );

  cout << endl << endl << "\t\tWelcome to the Bank of SMC!" << endl;
  do {
    choice = menu();
    try {
        switch (choice) {
        case CREATE:
            cout << "Please enter your name and opening bank balance: ";
            cin  >> name >> balance;
            account.setName(name);
            account.deposit(balance);
            break;
        case DEPOSIT:
            cout << "Please enter your name and amount to withdrawal: ";
            cin  >> name >> balance;
            depositaccount.setName(name);
            depositaccount.deposit(balance);
            account = account + depositaccount;
            break;
        case WITHDRAW:
            cout << "Please enter your name and amount to withdrawal: ";
            cin  >> name >> balance;
            withdrawaccount.setName(name);
          withdrawaccount.deposit(balance);
            account = account - withdrawaccount;
            break;
        case PRINT:
            cout << account;
            break;
        case QUIT:
            break;
        }
    } catch (logic_error le) {
        cout << "Caught logic_error" << endl;
        cout << "Transaction failed to process" << endl;
        cout << "Please try again!" << endl;
    }

  } while (choice != QUIT);

  return 0;
}

CHOICE menu() {
  CHOICE result;
  char   answer;
  cout << "(C)reate (D)eposit (W)ithdrawal (P)rint (Q)uit ";
  cin  >> answer;
  switch (answer) {
  case 'C':
  case 'c':
        result = CREATE;
        break;
  case 'D':
  case 'd':
        result = DEPOSIT;
        break;
  case 'W':
  case 'w':
        result = WITHDRAW;
        break;
  case 'P':
  case 'p':
        result = PRINT;
        break;
  case 'Q':
  case 'q':
        result = QUIT;
        break;
  }
  return( result );
}
```

- This is a good example because classes typically throw exceptions to indicate failure
- This is sense, the class is typically the callee and the user of the class is the caller

Auto example

- Exceptions are good because they allow you to greatly simplify your error checking using a consistent system that handles all error checking in one place
- To illustrate this, let’s consider the following example
- We’ll imagine we have a `Car` class that can fail for a number of reasons, each of which is specific to a reasonable real-world circumstance a car may face

Example: `auto_if.cpp`

```


WITH A C-MENTALITY AND NO EXCEPTION HANDLING....


///  Supposing I Have The Class Auto
///  I Am Going To Drive To Work...

Car c( "Honda", "Prelude" );
rv = c.openDoor();
if (rv == DOOR_LOCKED || rv == CAR_STOLEN || rv == WRONG_KEYS || rv == WRONG_CAR ) {
   // something bad happened...
}
else {
   rv = c.insertKey();
   if (rv == WRONG_KEYS || rv == KEY_UPSIDE_DOWN || rv == WRONG_CAR ) {
     // something bad happened...
   }
   else {
      rv = c.turnKey();
      if (rv == DEAD_BATTERY || rv == NO_GAS || rv == ASTEROID_HITS_ENGINE || rv == SADDAM_IN_ENGINE) {
         // something bad happened...
      }
      else {
         rv = c.intoReverse();
         if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == FLAT_TIRE || rv == NO_GAS || rv == PARKING_BRAKE_UP) {
        // something bad happened...
         }
     else {
        rv = c.drive();
            if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS || rv == NUCLEAR_WAR) {
           // something bad happened...
        }
        else {
        rv = c.intoFirst();
        if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS || rv == SPACE_SHUTTLE_DEBRIS_HITS_WINDSHIELD) {
           // something bad happened...
        }
        else {
           // Isn't this approach ridiculous???
           // I've literally spent so much time checking for errors, that I can't figure
           // out what my code was actually supposed to do...
        }
        }
     }
      }
   }
}



VERSUS


///  Supposing I Have The Class Auto
///  I Am Going To Drive To Work...

```

```
Car c( "Honda", "Prelude" );
try {
  c.openDoor();
  c.insertKey();
  c.turnKey();
  c.intoReverse();
  c.drive();
  c.intoFirst();
} catch( OutOfGasError ooge ) {
  // something bad happened...
} catch( WrongKeysError wke ) {
  // something bad happened...
} catch( ClutchDiedError cde ) {
  // something bad happened...
} catch( GearFailedError gfe ) {
  // something bad happened...
} catch( FlatTireError fte ) {
  // something bad happened...
}
```

Example: **`auto_exception.cpp`**

- The above is rather hard to read, hard to maintain, and hard to expand
- Consider the following similar approach using exceptions

```


WITH A C-MENTALITY AND NO EXCEPTION HANDLING....


///  Supposing I Have The Class Auto
///  I Am Going To Drive To Work...

Car c( "Honda", "Prelude" );
rv = c.openDoor();
if (rv == DOOR_LOCKED || rv == CAR_STOLEN || rv == WRONG_KEYS || rv ==
    WRONG_CAR ) {
   // something bad happened...
}
else {
   rv = c.insertKey();
   if (rv == WRONG_KEYS || rv == KEY_UPSIDE_DOWN || rv == WRONG_CAR ) {
     // something bad happened...
   }
   else {
      rv = c.turnKey();
      if (rv == DEAD_BATTERY || rv == NO_GAS || rv ==
         ASTEROID_HITS_ENGINE || rv == SADDAM_IN_ENGINE) {
         // something bad happened...
      }
      else {
         rv = c.intoReverse();
         if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == FLAT_TIRE
            || rv == NO_GAS || rv == PARKING_BRAKE_UP) {
        // something bad happened...
         }
     else {
        rv = c.drive();
            if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS
               || rv == NUCLEAR_WAR) {
           // something bad happened...
        }
        else {
        rv = c.intoFirst();
        if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS ||
           rv == SPACE_SHUTTLE_DEBRIS_HITS_WINDSHIELD) {
           // something bad happened...
        }
        else {
           // Isn't this approach ridiculous???
           // I've literally spent so much time checking for errors,
              that I can't figure
           // out what my code was actually supposed to do...
        }
        }
     }
      }
   }
}



VERSUS


```

```
///  Supposing I Have The Class Auto
///  I Am Going To Drive To Work...

Car c( "Honda", "Prelude" );
try {
  c.openDoor();
  c.insertKey();
  c.turnKey();
  c.intoReverse();
  c.drive();
  c.intoFirst();
} catch( OutOfGasError ooge ) {
  // something bad happened...
} catch( WrongKeysError wke ) {
  // something bad happened...
} catch( ClutchDiedError cde ) {
  // something bad happened...
} catch( GearFailedError gfe ) {
  // something bad happened...
} catch( FlatTireError fte ) {
  // something bad happened...
}
```