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?

```
1 BankAccount operator+ (const BankAccount& b1, const BankAccount& b2){
2    BankAccount result;
3    if (b1.my_Name == b2.my_Name) {
4       result = BankAccount(b1.my_Name, b1.my_Balance + b2.my_Balance);
5    }
6    return result;
7 }
```

- 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:

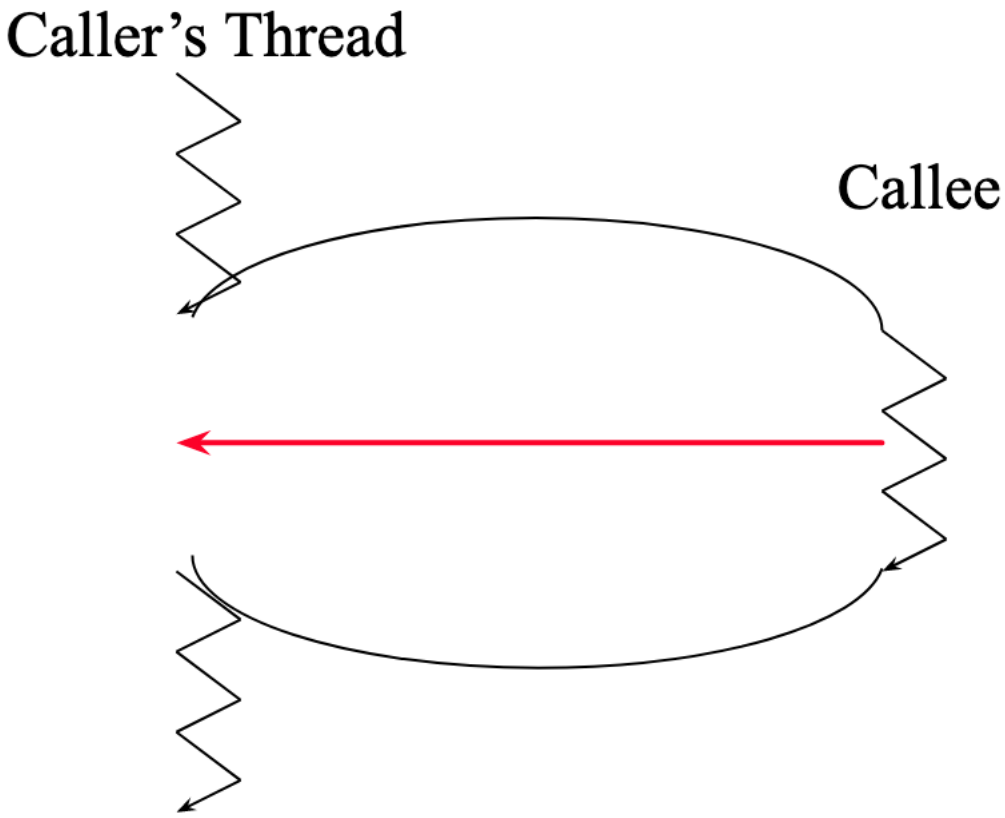

Figure 1: Caller-callee relationship

- 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`

```
1  //---------------------------------------------------------------------
2  // INTERFACE FILE: baccount.h
3  //
4  // Defines class BankAccount
5  //
6  //---------------------------------------------------------------------
7  // SAFEGUARDS AND INCLUDES
8  #ifndef BANKACCOUNT_H   // Avoid redeclaring class BankAccount.
9  #define BANKACCOUNT_H   // This code is compiled only once
10 #include <string>    // for class string
11
12 namespace cs52 {
13
14 ////////////////////////////////////////
15 /////// class BankAccount defintion ///////
16 ////////////////////////////////////////
17
18 class BankAccount {
19 public:  // class member functions
20
21 //--constructors
22     BankAccount();
23
24     BankAccount(std::string initName, double initBalance);
25     // post: A BankAccount with two arguments when called like this:
26     //       BankAccount anAcct("Hall", 100.00);
27
28 //--modifiers
29
30     void deposit(double depositAmount);
31     // post: depositAmount is credited to this object's balance
```

```
32
33      void withdraw(double withdrawalAmount);
34      // post: withdrawalAmount is debited from this object's balance
35
36  //--accessors
37
38      double balance() const;
39      // post: return this account's current balance
40
41      std::string name() const;
42      // post return the account name
43
44      void setName( std::string initName );
45      // post updates the member variable my_name
46
47      // ADDED CODE BEGINS HERE
48      friend std::ostream& operator << ( std::ostream& outs, const
            BankAccount& b );
49      friend std::istream& operator >> ( std::istream& ins, BankAccount&
            b );
50      friend BankAccount operator + ( const BankAccount& left, const
            BankAccount& right );
51      friend BankAccount operator - ( const BankAccount& left, const
            BankAccount& right );
52      friend bool operator ==( const BankAccount& left, const BankAccount
            & right );
53      friend bool operator < ( const BankAccount& left, const BankAccount
            & right );
54      friend bool operator > ( const BankAccount& left, const BankAccount
            & right );
55
56  private:
57      std::string my_name;    // Uniquely identify an object
58      double my_balance; // Store the current balance (non-persistent)
59  };
60
61  }
62
63  #endif   // ifndef BANKACCOUNT_H
```

Example: `ExceptionBankAccount.cpp`

```
1  //--------------------------------------------------------------------
2  // IMPLEMENTATION FILE: baccount.cpp
3  //
4  // Implements 1. class BankAccount member functions
5  //
6  //--------------------------------------------------------------------
7  #include "ExceptionBankAccount.h"  // allows for separate compilation
       if you want
8  #include <iostream>    // for ostream << and istream >>
9  #include <string>      // for class string
10 #include <stdexcept>   // supports Linux exception classes
11
12 using namespace std;
13
14 namespace cs52 {
15
16 //--constructors
17
18 BankAccount::BankAccount()
19 {
20   my_name = "?name?";
21   my_balance = 0.0;
22 }
23
24 BankAccount::BankAccount(string initName, double initBalance)
25 {
26   my_name = initName;
27   my_balance = initBalance;
28 }
29
30 //--modifiers
31
32 void BankAccount::deposit(double depositAmount)
33 {
34   my_balance = my_balance + depositAmount;
35 }
36
37 void BankAccount::withdraw(double withdrawalAmount)
38 {
39   my_balance = my_balance - withdrawalAmount;
40 }
41
```

```
42 //--accessors
43
44 double BankAccount::balance() const
45 {
46   return my_balance;
47 }
48
49 string BankAccount::name() const
50 {
51   return my_name;
52 }
53
54 void BankAccount::setName( string initName )
55 {
56   my_name = initName;
57 }
58
59
60 // NEW CODE STARTS HERE
61 std::ostream& operator << ( std::ostream& outs, const BankAccount& b )
       {
62     outs << b.my_name << " " << b.my_balance << endl;
63     return( outs );
64 }
65
66 std::istream& operator >> ( std::istream& ins, BankAccount& b ) {
67     ins >> b.my_name >> b.my_balance;
68     return( ins );
69 }
70
71 BankAccount operator + ( const BankAccount& left, const BankAccount&
      right ) {
72     BankAccount newB;
73     if (left.my_name == right.my_name) {
74         newB.deposit( left.my_balance );
75         newB.deposit( right.my_balance );
76     }
77     else {
78         cerr << "YIKES!  These two accounts can't be added together
               since the names differ!" << endl;
79         throw logic_error( "Bad account names" );
80     }
81     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
```

```
9
10 using namespace std;                // supports cout
11 using namespace cs52;                          // for BankAccount class
12
13 enum CHOICE { CREATE, DEPOSIT, WITHDRAW, PRINT, QUIT };
14
15 CHOICE menu();
16
17 int main( )
18 {
19   CHOICE choice;
20   BankAccount account, withdrawaccount, depositaccount;
21   string name;
22   double balance;
23
24   cout.setf( ios::fixed );
25   cout.setf( ios::showpoint );
26   cout.precision( 2 );
27
28   cout << endl << endl << "\t\tWelcome to the Bank of SMC!" << endl;
29   do {
30     choice = menu();
31     try {
32         switch (choice) {
33         case CREATE:
34             cout << "Please enter your name and opening bank balance: "
                    ;
35             cin  >> name >> balance;
36             account.setName(name);
37             account.deposit(balance);
38             break;
39         case DEPOSIT:
40             cout << "Please enter your name and amount to withdrawal: "
                    ;
41             cin  >> name >> balance;
42             depositaccount.setName(name);
43             depositaccount.deposit(balance);
44             account = account + depositaccount;
45             break;
46         case WITHDRAW:
47             cout << "Please enter your name and amount to withdrawal: "
                    ;
48             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;
```

```
92    case 'Q':
93    case 'q':
94          result = QUIT;
95          break;
96    }
97    return( result );
98  }
```

- 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`

```
1
2
3  WITH A C-MENTALITY AND NO EXCEPTION HANDLING....
4
5
6  ///  Supposing I Have The Class Auto
7  ///  I Am Going To Drive To Work...
8
9  Car c( "Honda", "Prelude" );
10 rv = c.openDoor();
11 if (rv == DOOR_LOCKED || rv == CAR_STOLEN || rv == WRONG_KEYS || rv ==
       WRONG_CAR ) {
12    // something bad happened...
13 }
14 else {
15    rv = c.insertKey();
16    if (rv == WRONG_KEYS || rv == KEY_UPSIDE_DOWN || rv == WRONG_CAR ) {
17      // something bad happened...
18    }
19    else {
```

```
20        rv = c.turnKey();
21        if (rv == DEAD_BATTERY || rv == NO_GAS || rv ==
              ASTEROID_HITS_ENGINE || rv == SADDAM_IN_ENGINE) {
22           // something bad happened...
23        }
24        else {
25           rv = c.intoReverse();
26           if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == FLAT_TIRE
                 || rv == NO_GAS || rv == PARKING_BRAKE_UP) {
27          // something bad happened...
28           }
29       else {
30          rv = c.drive();
31              if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS
                    || rv == NUCLEAR_WAR) {
32             // something bad happened...
33          }
34          else {
35          rv = c.intoFirst();
36          if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS ||
                rv == SPACE_SHUTTLE_DEBRIS_HITS_WINDSHIELD) {
37             // something bad happened...
38          }
39          else {
40             // Isn't this approach ridiculous???
41             // I've literally spent so much time checking for errors,
                   that I can't figure
42             // out what my code was actually supposed to do...
43          }
44          }
45       }
46        }
47    }
48 }
49
50
51
52 VERSUS
53
54
55 ///  Supposing I Have The Class Auto
56 ///  I Am Going To Drive To Work...
57
```

```
58 Car c( "Honda", "Prelude" );
59 try {
60   c.openDoor();
61   c.insertKey();
62   c.turnKey();
63   c.intoReverse();
64   c.drive();
65   c.intoFirst();
66 } catch( OutOfGasError ooge ) {
67   // something bad happened...
68 } catch( WrongKeysError wke ) {
69   // something bad happened...
70 } catch( ClutchDiedError cde ) {
71   // something bad happened...
72 } catch( GearFailedError gfe ) {
73   // something bad happened...
74 } catch( FlatTireError fte ) {
75   // something bad happened...
76 }
```

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

```
1
2
3  WITH A C-MENTALITY AND NO EXCEPTION HANDLING....
4
5
6  ///  Supposing I Have The Class Auto
7  ///  I Am Going To Drive To Work...
8
9  Car c( "Honda", "Prelude" );
10 rv = c.openDoor();
11 if (rv == DOOR_LOCKED || rv == CAR_STOLEN || rv == WRONG_KEYS || rv ==
       WRONG_CAR ) {
12    // something bad happened...
13 }
14 else {
15    rv = c.insertKey();
16    if (rv == WRONG_KEYS || rv == KEY_UPSIDE_DOWN || rv == WRONG_CAR ) {
```

```
17        // something bad happened...
18     }
19     else {
20        rv = c.turnKey();
21        if (rv == DEAD_BATTERY || rv == NO_GAS || rv ==
              ASTEROID_HITS_ENGINE || rv == SADDAM_IN_ENGINE) {
22           // something bad happened...
23        }
24        else {
25           rv = c.intoReverse();
26           if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == FLAT_TIRE
                 || rv == NO_GAS || rv == PARKING_BRAKE_UP) {
27          // something bad happened...
28           }
29       else {
30          rv = c.drive();
31              if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS
                    || rv == NUCLEAR_WAR) {
32             // something bad happened...
33          }
34          else {
35          rv = c.intoFirst();
36          if (rv == CLUTCH_DIED || rv == GEAR_FAILED || rv == NO_GAS ||
                rv == SPACE_SHUTTLE_DEBRIS_HITS_WINDSHIELD) {
37             // something bad happened...
38          }
39          else {
40             // Isn't this approach ridiculous???
41             // I've literally spent so much time checking for errors,
                   that I can't figure
42             // out what my code was actually supposed to do...
43          }
44          }
45       }
46        }
47     }
48  }
49
50
51
52  VERSUS
53
54
```

```
55 ///  Supposing I Have The Class Auto
56 ///  I Am Going To Drive To Work...
57
58 Car c( "Honda", "Prelude" );
59 try {
60   c.openDoor();
61   c.insertKey();
62   c.turnKey();
63   c.intoReverse();
64   c.drive();
65   c.intoFirst();
66 } catch( OutOfGasError ooge ) {
67   // something bad happened...
68 } catch( WrongKeysError wke ) {
69   // something bad happened...
70 } catch( ClutchDiedError cde ) {
71   // something bad happened...
72 } catch( GearFailedError gfe ) {
73   // something bad happened...
74 } catch( FlatTireError fte ) {
75   // something bad happened...
76 }
```