CSCE 121

Exceptions

Dr. Tim McGuire

Some of the material and images from Michael Main, University of Colorado, and Tony Gaddis, Haywood Community College

Exceptions

Exception Handling Basics

- It is often easier to write a program by first assuming that nothing incorrect will happen
- Once it works correctly for the expected cases, add code to handle the exceptional cases
- Exception handling is commonly used to handle error situations
  - Once an error is handled, it is no longer an error

Functions and Exception Handling

- A common use of exception handling:
  - Functions with a special case that is handled in different ways depending on how the function is used
  - If the function is used in different programs, each program may require a different action when the special case occurs

Exception Handling Mechanism

- In C++, exception handling proceeds by:
  - Some library software or your code signals that something unusual has happened
  - This is called throwing an exception
  - At some other place in your program you place the code that deals with the exceptional case
  - This is called handling the exception

A Toy Example

- Exception handling is meant to be used sparingly in situations that are generally not reasonable introductory examples
- For this example:
  - Suppose coffee is so important that we almost never run out
  - We still would like our program to handle the situation of running out of coffee

The Coffee Example (cont.)

- Code to handle the normal situations involving coffee, might be:

```
cout << "Enter number of kolaches:\n";
 cin >> kolaches;
 cout << "Enter number of mugs of coffee:\n";
 cin >> coffee;
 kpm = kolaches /static_cast<double>(coffee);
 cout << kolaches << " kolaches.\n"
           << coffee << " mugs of coffee.\n"
           << "You have " << kpm
           << " kolaches per mug of coffee.\n";
```

The No Coffee Problem

- If there is no coffee, the code on the previous slide results in a division by zero
  - We could add a test case for this situation
  - `NoCoffee-1.cpp` shows the program with the test case
  - `NoCoffee-2.cpp` shows the program rewritten using an exception

The try Block

- `NoCoffee-2.cpp` replaces the test case in the if-else statement with:

```
if(coffee <= 0)
        throw kolaches;
```

- This code is found in the try block

```
try
{
        Some_Code
}
```

which encloses the code to handle the normal situations

The Try Block Outline

- The try block encloses code that you want to "try" but that could cause a problem
- The basic outline of a try block is:

```
try
{
    Code_To_Try
    Possibly_Throw_An_Exception
    More_Code
}
```

The Exception

- To throw an exception, a `throw`-statement is used to throw a value
  - In the coffee example:

    ```
    throw kolaches;
    ```

    throws an integer value.
  - The value thrown is sometimes called an *exception*
  - In C++ you can throw a value of any type

The catch-block

- Something that is thrown goes from one place to another
- In C++ `throw` causes the flow of control to go to another place
    - When an exception is thrown, the try block stops executing and the catch-block begins execution
    - This is *catching* or *handling* the exception

The Coffee catch-block

- The catch-block from the coffee example looks like, but is not, a function definition with a parameter:

```
catch(int e)
{
    cout << e << kolaches, and no coffee!\n"
         << "Go buy some coffee.\n";
}
```

- If no exception is thrown, the catch-block is ignored during program execution

The catch-block Parameter

- The catch-block parameter, (recall that the catch-block is not a function) does two things:
  - The type of the catch-block parameter identifies the kind of value the catch-block can catch
  - The catch-block parameter provides a name for the value caught so you can write code using the value that is caught

try-blocks and if-else

- try-blocks are very similar to if-else statements
  - If everything is normal, the entire try-block is executed
  - else, if an exception is thrown, the catch-block is executed
- A big difference between try-blocks and if-else statements is the try-block's ability to send a message to one of its branches

try-throw-catch Review

- This is the basic mechanism for throwing and catching exceptions
  - The `try`-block includes a throw-statement
  - If an exception is thrown, the `try`-block ends and the `catch`-block is executed
  - If no exception is thrown, then after the `try`-block is completed, execution continues with the code following the `catch`-block(s)

Defining an Exception Class

- Because a `throw`-statement can throw a value of any type, it is common to define a class whose objects can carry the kind of information you want thrown to the `catch`-block
- A more important reason for a specialized exception class is so you can have a different type to identify each possible kind of exceptional situation

A NoCoffee Class

```
class NoCoffee
{
public:
    NoCoffee();
    NoCoffee(int howMany);
    int getDonuts();
private:
    int count;
};
```

```
NoCoffee::NoCoffee(){
   count = 0;
}
NoCoffee::NoCoffee(int
howMany) {     count =
howMany)
}
int NoCoffee::getDonuts() {
    return count;
}
```

The Exception Class

- An exception class is just a class that happens to be used as an exception class
- An example of a program with a programmer defined exception class is in `NoCoffee-3.cpp`

Throwing a Class Type

- The program in `NoCoffee-3.cpp` uses the `throw`-statement

```
throw NoCoffee(kolaches);
```

  - This invokes a constructor for the class `NoCoffee`
  - The constructor takes a single argument of type **int**
  - The `NoCoffee` object is what is thrown
  - The `catch`-block then uses the statement

    ```
    e.get_kolaches( )
    ```

    to retrieve the number of kolaches

Multiple Throws and Catches

- A `try`-block can throw any number of exceptions of different types
  - In any one execution, only one exception can be thrown
  - Each `catch`-block can catch only one exception
  - Multiple `catch`-blocks may be used
    - A parameter is not required in a `catch`-block
- A sample program with two catch-blocks is found in `DeepSpaceNine.cpp`

A Default catch-block

- When catching multiple exceptions, write the catch-blocks for the most specific exceptions first
    - Catch-blocks are tried in order and the first one matching the type of exception is executed
- A default (and last) catch-block to catch any exception can be made using "..." as the catch-block parameter

```
catch(…)
{
    <the catch block code>
}
```

Exception Class DivideByZero

- In DeepSpaceNine.cpp, exception class DivideByZero was defined as
```
    class DivideByZero
    { } ;
```
  - This class has no member variables or member functions
  - This is a trivial exception class
  - DivideByZero is used simply to activate the appropriate catch-block
  - There is nothing to do with the catch-block parameter so it can be omitted as shown in DeepSpaceNine.cpp

Exceptions In Functions

- In many cases, an exception generated in a function is not handled in the function
  - It might be that some programs should end, while others might do something else, so within the function you might not know how to handle the exception
- In this case, the program places the function invocation in a try block and catches the exception in a following catch-block

Function `safeDivide`

- SafeDivision.cpp includes a function that throws, but does not catch an exception
  - In function safeDivide, the denominator is checked to be sure it is not zero. If it is zero, an exception is thrown:

```
        if (bottom == 0)
            throw DivideByZero( );
```

  - The call to function safeDivide is found in the try-block of the program

Programming Techniques for Exception-Handling

Programming Techniques for Exception Handling

- A guideline for exception handling is to separate throwing an exception and catching an exception into separate functions
  - Place the throw-statement in one function
  - Place the function invocation and catch-clause in a try-block of a different function

try and throw…Again

- Here is a general example the approach to use in using throw:

```
void functionA( )
{
        …
        throw MyException(<an argument?>);
}
```

catch…again

- Using FunctionA from the previous slide, here is how to catch MyException:

```
void functionB( )
{
        …
     try
     {
            …
          functionA( );
            …
     }
     catch(MyException e)
      {
          < handle the exception>
      }
}
```

When to Throw An Exception

- Throwing exceptions is generally reserved for those cases when handling the exceptional case depends on how and where the function was invoked
  - In these cases it is usually best to let the programmer calling the function handle the exception
  - An uncaught exception ends your program
- If you can easily write code to handle the problem do not throw an exception

Nested try-catch Blocks

- Although a try-block followed by its catch-block can be nested inside another try-block
    - It is almost always better to place the nested try-block and its catch-block inside a function definition, then invoke the function in the outer try-block
- An error thrown but not caught in the inner try-catch-blocks is thrown to the outer try-block where it might be caught

Overuse of Exceptions

- Throwing an exception allows you to transfer flow of control to almost any place in your program
- Such un-restricted flow of control is generally considered poor programming style as it makes programs difficult to understand
- Exceptions should be used sparingly and only when you cannot come up with an alternative that produces reasonable code

Exception Class Hierarchies

- It can be useful to define a hierarchy of exception classes.
  - You might have an ArithmeticError exception class with DivideByZeroError as a derived class
  - Since a DivideByZeroError object is also an ArithmeticError object, every catch-block for an ArithmeticError will also catch a DivideByZeroError

Checking For Available Memory

- The new operator allocates memory from the heap:
  NodePtr pointer = new Node;
  - What if there is no memory available?
  - bad_alloc is a predefined exception and can be used in this way since new throws a bad_alloc exception:

```
try
{
     NodePtr pointer = new Node;
}
catch(bad_alloc)
{
     cout << "Ran out of memory!";
}
```

```
// bad_alloc example
#include <iostream>     // std::cout
#include <new>          // std::bad_alloc

int main () {
  try
  {
    int* myarray= new int[10000];
  }
  catch (std::bad_alloc& ba)
  {
    std::cerr << "bad_alloc caught: " << ba.what() << '\n';
  }
  return 0;
}
```

Rethrowing an Exception

- The code within a catch-block can throw an exception
  - This feature can be used to pass the same or a different exception up the chain of exception handling blocks