Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 9

Date: October 26, 2015

What we're learning in this Slide Set:

- Exception Handling

Textbook Chapters:

Relevant to this slide set:

- Chapter 16

Coming soon:

- Chapter 17 & 18
- Sets (Set theory)

“To err is human, to really foul things up requires a computer.”

‘Senator Soaper’ (aka Bill Vaughan)

"Well the program works for most cases. I didn't know it had to work for that case."

Student Appealing a Grade

Exception Handling and Error Cases

When you write code, remember what I said at the beginning:

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."

- Author **Rick Cook**, *The Wizardry Compiled*

As such, your code design needs to be robust, assuming that the "evil" user will not use it correctly.

Exception Handling and Error Cases

Error Cases:

In this case, you are checking the user code for anticipated errors that will not work. Depending on the situation, you may choose for data reentry or termination (think of at least one example of each situation).

Exception Handling:

In this case, you are generally trying handle “exceptional” situations- things that might damage the system (e.g. memory leaks …)

General overview of Exception Handling

Either your user code or some software library provides a mechanism that indicates when something unusual happens.

- This is called *throwing an exception.*

A separate portion of code then deals with this “exceptional case”

- This is called *handling an exception*.

By separating out the unusual behaviour from the typical behaviour, your code ends up being cleaner

General overview of Exception Handling

Exceptions can also be viewed as “software interrupts” that occur at the system-level

- You’ll learn more about interrupts in ENSC 254.

Let’s look at a simple example

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts for each glass of milk.\n";
```

Suppose that milk is important enough to people that they almost never run out.

When they do, however, (`milk = 0`), we would like our code to be robust enough to handle this situation (else `dpg` $= \infty$)

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts for each glass of milk.\n";
```

Up until now, I would have expected you to solve this situation by adding a test case to your code to ensure that you don’t try to divide by zero.

The next slide has that version of the code.

```
include <iostream>
using namespace std;

int main()
{
    int donuts, milk;
    double dpg;
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;

    if (milk <= 0)
    {
        cout << donuts << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    Else
    {
        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Sample Output for program on previous slide:

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

Alternatively, we could write this same code using exception handling to “catch” the error.

- In the code on the next slide, pay particular attention to the “try”, “throw”, and “catch”

The main program is simpler (located in the “try” section of the code).

- If there is a problem it “throw”s and exception.

The “catch” section handles the exception if one occurs.

```
#include <iostream>
using namespace std;

int main()
{
    int donuts, milk;
    double dpg;

    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
            throw donuts;

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(int e)
    {
        cout << e << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Sample output for program with exception handling:

Ver 1:

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
```

Ver 2:

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

Looking between the “try” and “catch statements:

- As you can see, the exception handling code is largely the same as the original version

However, the if-else statement is much smaller:

```
if (milk <= 0)
{
    cout << donuts << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
Else
{
    dpg = donuts/static_cast<double>(milk);
    cout << donuts << " donuts.\n"
         << milk << " glasses of milk.\n"
         << "You have " << dpg
         << " donuts for each glass of milk.\n";
}
```

versus

```
if (milk <= 0)
    throw donuts;
```

The new if statement says that if there is no milk, “throw” an exception:

```
if (milk <= 0)
    throw donuts;
```

When the exception is thrown, the code execution jumps to the handler.

This handler is found in the “catch” section;

- It is designed to “handle” the “exceptional” situation.

The basic structure for handling exceptions in C++ consists of the three components:

1. try
2. throw
3. catch

Try:

A try block has the syntax:

```
try
{
    Some_Code
}
```

The try block contains the code for the basic algorithm (when everything is normal).

It's called a "try" block because it may not work, so you want to "give it a try"

If something does go wrong, we want to throw an exception.

Throw:

A try block with a throw statement has the syntax:

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

Let's go back to our example

Throw:

```
try
{
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;
    if (milk <= 0)
        throw donuts;
    dpg = donuts/static_cast<double>(milk);
    cout << donuts << " donuts.\n"
         << milk << " glasses of milk.\n"
         << "You have " << dpg
         << " donuts for each glass of milk.\n";
}
```

In this case, we "throw" the int value donuts: `throw donuts;`

Other notes:

- The value thrown (in this case donuts) is sometimes called an exception.
- The execution of a throw statement is called "throwing an exception.
- The value thrown can be of any type

Summary of the Throw statement:

throw Statement

SYNTAX

```
throw Expression_for_Value_to_Be_Thrown;
```

When the *throw* statement is executed, the execution of the enclosing *try* block is stopped. If the *try* block is followed by a suitable *catch* block, then flow of control is transferred to the *catch* block. A *throw* statement is almost always embedded in a branching statement, such as an *if* statement. The value thrown can be of any type.

EXAMPLE

```
if (milk <= 0)
    throw donuts;
```

Catch:

In the case of C++, what we are throwing to the catch statement is:

- Execution control, and
- The exception value (e.g. donuts)

When we throw an exception, the try block stops executing and the catch block begins execution.

- Execution of this block is called catching the exception or handling the exception.

Remember, if we are going to throw an exception, it should be caught by some catch block.

Catch:

In our example, the catch block immediately follow the try block:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

The catch block resembles a function definition with a parameter of type int.

- It is not a function definition, but it is similar:
  - It is a separate piece of code that is only executed when the `throw some_int` code is executed.
- This means that the throw call is similar to a function call, but instead of calling a function it “calls” the catch block.

A Catch block is often referred to as an *exception handler*.

Catch:

What is the identifier ‘e’ in the first line of the catch block?

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

It looks like a parameter (and behaves like one); the text calls it the catch-block parameter (but the catch block is not a function).

This parameter does two things:

- It is preceded by a type name that specifies what kind of thrown value the catch block can catch/handle.
- It gives you a name for the thrown value that can be used in the code written for the catch block that wants to manipulate the thrown value.

Catch:

You can use any legal identifier for the catch block parameter, but people commonly use ‘e’

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

Our example catch statement can only catch int values.

It only gets executed when the value of milk is zero (or less than zero).

- When this happens, the value of donuts is thrown and plugged into the catch-block parameter e.

If the value of milk is positive, the catch block is not executed. The complete try block is executed and the statement(s) after the catch block are then executed.

The try-throw-catch setup is very similar to an if-else statement, except that the if-else statement cannot throw a value.

The ability to throw a value lets the try-throw-catch mechanism act like an if-else statement with the added ability to send a message to one of the branches.

- This can be very useful.

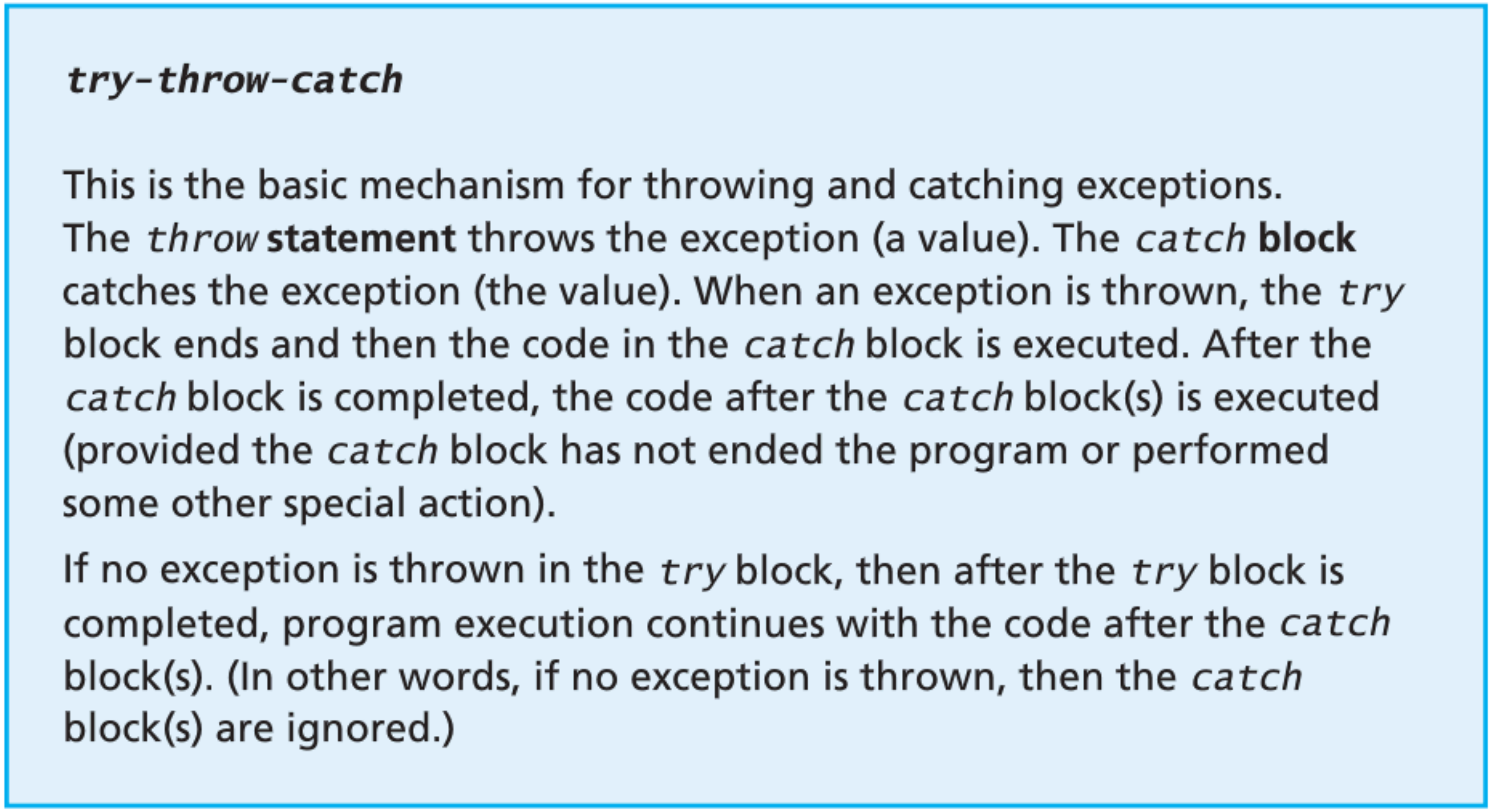
A couple other notes:

- When an exception is thrown, the rest of the code in the try block is ignored.
- A catch block applies only to an immediately preceding try block.
- If an exception is thrown, the value of the exception object is assigned to the catch-block parameter.

Note: If no exception is thrown in the try block, then once the try block code is completed, program execution continues with the code ***after*** the catch block (it is ignored).

- This should reflect normal execution.

Summary:



try-throw-catch

This is the basic mechanism for throwing and catching exceptions. The *throw* **statement** throws the exception (a value). The *catch* **block** catches the exception (the value). When an exception is thrown, the *try* block ends and then the code in the *catch* block is executed. After the *catch* block is completed, the code after the *catch* block(s) is executed (provided the *catch* block has not ended the program or performed some other special action).

If no exception is thrown in the *try* block, then after the *try* block is completed, program execution continues with the code after the *catch* block(s). (In other words, if no exception is thrown, then the *catch* block(s) are ignored.)

Syntax Review:

```
try
{
    Some_Statements
        < Either some code with a throw statement or a
               function invocation that might throw an
               exception>
    Some_More_Statements
}
catch(Type_Name e)
{
        < Code to be performed if a value of the
          catch-block parameter type is thrown in the
          try block>
}
```

Notes: This is a high-level language perspective of exception handling.

This done considerably differently in Assembly language and in C (there are more details to worry about) particularly when you use an OS.

Defining your own Exception Classes:

Recall that a throw statement can throw a value of any type.

Often people define a class whose objects can carry the precise kind of information you want thrown to the catch block.

- This lets you throw more than one value to an exception handler.
- More importantly, it lets you specialize the exception class so that you can use different types to identify each possible kind of exception.
  - This is particularly valuable if all of your exceptions throw ints.

Defining your own Exception Classes:

Exception classes are classes like any other.

What makes it an exception class is how it's used.

You should also pay attention to the Name (so it makes sense) and other details.

In the example on the next slide, the exception handling class is NoMilk and the throw statement is:

```
throw NoMilk(donuts);
```

Note that the part NoMilk(donuts) is an invocation of a constructor for the class NoMilk with one int argument (to store the value of donuts).

After the object is created, the object is thrown to the handler.

Our Exception Class:

```
#include <iostream>
using namespace std;
class NoMilk
{
public:
    NoMilk();
    NoMilk(int how_many);
    int get_donuts();
private:
    int count;
};
NoMilk::NoMilk()
{}
NoMilk::NoMilk(int how_many) : count(how_many)
{}

int NoMilk::get_donuts()
{
    return count;
}
```

This is just a toy example to learn C++ syntax. Do not take it as an example of good typical use of exception handling.

The sample dialogues are the same as in Display 16.2.

The main program:

```
int main()
{
    int donuts, milk;
    double dpg;
    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;
        if (milk <= 0)
                throw NoMilk(donuts);
        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(NoMilk e)
    {
        cout << e.get_donuts() << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    cout << "End of program.";
    return 0;
}
```

Multiple Throws and Catches:

A try block can throw multiple exception values of the same or different types

- However, in any given execution of a try block, only one exception can be thrown (remember it ends the execution of the try block).

Since each catch block can only catch values of one type, you simply add additional catch blocks to catch the exception values of differing types.

Our next example has two catch blocks after the try block.

- Note that the catch block DivideByZero has no parameter:
  - This is fine, you do not need a parameter and can simply list the type with no parameter.

Our Exception Classes and their implementation:

Although not done here, exception classes can have their own interface and implementation files and can be put in a namespace.
This is another toy example.

```
#include <iostream>
#include <string>
using namespace std;

class NegativeNumber
{
public:
    NegativeNumber();
    NegativeNumber(string take_me_to_your_catch_block);
    string get_message();
private:
    string message;
};

class DivideByZero
{};


NegativeNumber::NegativeNumber()
{}

NegativeNumber::NegativeNumber(string take_me_to_your_catch_block)
     : message(take_me_to_your_catch_block)
{}

string NegativeNumber::get_message()
{
    return message;
}
```

The main program and try block:

```
int main()
{
    int jem_hadar, klingons;
    double portion;

try
{
    cout << "Enter number of JemHadar warriors:\n";
    cin >> jem_hadar;
    if (jem_hadar< 0)
        throw NegativeNumber("JemHadar");

    cout << "How many Klingon warriors do you have?\n";
    cin >> klingons;
    if (klingons< 0)
        throw NegativeNumber("Klingons");
    if (klingons != 0)
        portion = jem_hadar/static_cast<double>(klingons);
    else
        throw DivideByZero();
    cout << "Each Klingon must fight "
         << portion << " JemHadar.\n";
}
```

The main program catch blocks:

```
    catch(NegativeNumber e)
    {
        cout << "Cannot have a negative number of "
             << e.get_message() << endl;
    }

    catch (DivideByZero)
    {
        cout << "Send for help.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Sample Outputs from the program:

Version 1:

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 JemHadar.
End of program
```

Version 2:

```
Enter number of JemHadar warriors:
-10
Cannot have a negative number of JemHadar
End of program.
```

Version 3:

```
Enter number of JemHadar warriors:
1000
How many Klingon warriors do you have?
0
Send for help.
End of program.
```

Multiple Throws and Catches:

When catching multiple exceptions, the order of the catch blocks can be important.

When an exception value is thrown in a try block, the catch blocks are tried in order and the first one that matches the type of the exception thrown is executed.

For example, the following is a special kind of catch block that will catch a thrown value of any type:

```
catch(...)
{
    <Place whatever you want in here>
}
```

Note: You actually type the three dots into your program. Since this can catch any type, it makes a good default catch block, but it should be placed after all other catch blocks.

Multiple Throws and Catches with a default catch block:

For example this is good:

```
catch(NegativeNumber e)
{
    cout << "Cannot have a negative number of "
         << e.get_message() <<endl;
}
catch(DivideByZero)
{
    cout<< "Send for help.\n";
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
```

Multiple Throws and Catches with a default catch block:

For example this is ***bad***:

```
catch(NegativeNumber e)
{
    cout << "Cannot have a negative number of "
         << e.get_message() <<endl;
}
catch(...)
{
    cout << "Unexplained exception.\n";
}
catch(DivideByZero)
{
    cout << "Send for help.\n";
}
```

When a DivideByZero value is thrown, it will be caught by the “default” catch block as catch(…) can catch any value.

- This means that the DivideByZero catch block can never be reached. (Luckily many compilers will warn you of this mistake).

Exception Classes:

Exception classes can be trivial (with no member variables and no member functions (other than the default constructor).

- They have nothing but a name, but this is sufficient to uniquely identify them for cases when you want to throw multiple exceptions.

When using a trivial exception class, you won't be able to do anything with the value thrown (the exception) once it is caught by the catch block.

- In fact all it can do is uniquely identify the correct catch block (so you can omit the catch block parameter).
  - Even if the exception type is not trivial, you can still omit the catch-block parameter anytime you do not need it.

Throwing an Exception in a Function:

If you throw an exception in a function, you may wish to catch it in the main program or any function that calls this function.

- You might do this because different programs should behave differently if the exception is thrown (e.g. termination versus initialization to default values…)

To make this work, you place the function call in the try block and catch the exception following the try block.

- However, you throw the exception inside your function.

Note, even though there is no throw statement visible in the try block, from the perspective of program execution, the function call happens inside of the try block.

Our Exception Class, the function prototype and the beginning of main:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class DivideByZero
{};

double safe_divide(int top, int bottom) throw (DivideByZero);

int main()
{
    int numerator;
    int denominator;
    double quotient;
    cout << "Enter numerator:\n";
    cin >> numerator;
    cout << "Enter denominator:\n";
    cin >> denominator;
```

The try-throw-catch mechanism:

```
    try
    {
        quotient = safe_divide(numerator, denominator);
    }
    catch(DivideByZero)
    {
        cout << "Error: Division by zero!\n"
             << "Program aborting.\n";
        exit(0);
    }

    cout << numerator << "/" << denominator
         << " = " << quotient <<endl;

    cout << "End of program.\n";
    return 0;
}
```

```
double safe_divide(int top, int bottom) throw (DivideByZero)
{
    if (bottom == 0)
        throw DivideByZero();

    return top/static_cast<double>(bottom);
}
```

Throwing an Exception in a Function:

If a function might throw an exception in its definition, but does not catch its exception, it should warn programmers that any invocation of the function might possibly throw an exception.

Those exception types should be listed in an ***exception specification***:

```
double safe_divide(int top, int bottom) throw (DivideByZero);
```

This exception specification should appear in both the function declaration and the function definition (as shown in our example).

Throwing an Exception in a Function:

If a function has more than one function declaration (i.e. overloaded), then all function declarations must have identical exception specifications.

The exception specification is sometimes called the *throw list*.

If there is more than one possible exception that can be thrown in the function definition, then the exception types are separated by commas:

```
    void some_function( ) throw (DivideByZero, OtherException);
```

If there is no exception specification (no throw list) at all (not even an empty one), then it is the same as if all possible exception types were listed in the exception specification, enabling any exception that is thrown to be treated normally.

- Akin to an empty sensitivity list in VHDL.

Throwing an Exception in a Function:

When an exception is thrown in a function, but is not listed in the exception specification (i.e. it is excluded from the throw list and is not caught inside the function), the program ends.

- In this case, the function has indicated a throw list (even if it is an empty one) and this exception wasn't on it.
- That means that it will not be caught by any catch block so your program will end

Throwing an Exception in a Function:

Remember, the exception specification is for exceptions that "get outside" the function.

- If you catch the exception inside the function, then it does not belong in the exception specification.
- If they get outside the function, then they belong in the exception specification no matter where they originate.
  - For example if a function definition (fooA) includes an invocation of another function (fooB) and that other function (fooB) can throw an exception that is not caught, then the type of the exception should be placed in the exception specification of this function (fooA) as well.
- To say that a function should not throw any exceptions that are not caught inside the function, you use an empty exception specification.

```
void some_function( ) throw ( );
```

Throwing an Exception in a Function:

In summary:

```
void some_function( ) throw (DivideByZero, OtherException);
//Exceptions of type DivideByZero or OtherException are
//treated normally. All other exceptions end the program
//if not caught in the function body.

void some_function( ) throw ( );
//Empty exception list; all exceptions end the
//program if thrown but not caught in the function body.

void some_function( );
//All exceptions of all types treated normally.
```

Notes about type matching for exceptions:

Keep in mind that an object of a derived class is also an object of its base class.

- So if derived class D from base class B is in the exception specification, then a thrown object of class D will be treated normally since it is an object of class B type and B is in the exception specification

No type automatic type conversions are done for exceptions:

- If a double is in the exception specification, that does not account for throwing an int value. You need to include both int and doubles in the exception specification.

Notes about the exception specification:

Not all compilers treat the exception specification the same (some treat it as a comment),

Always including it makes your code more readable and consistent (as long as you do not fail to anticipate an exception that your function might throw in the exceptions specification which will cause program termination).

Obviously, this is a runtime behaviour, but it does depend on your compiler.

Notes about the exception specification for derived classes:

If you redefine or override a function definition in a derived class, it should have the same exception specification a it had in the base class.

- Or it should have an exception specification whose exceptions are a subset of those in the base class exception specification.

In other words, when you override/redefine a function, you cannot add any exceptions to the exception specification but you can delete some if you want.

- Remember this is because an object of a derived class can be used anywhere than an object of the base class can be used, so redefining/overwriting functions must fit any code written for an object of the base class.

When to throw an exception in C++:

You should reserve use of the try-catch-throw mechanism for situations where the way the exceptional condition is handled depends on ***how*** and ***where*** the situation occurs.

- Generally, it is best to let the programmer who invokes the function handle the exception.
- Otherwise if you can easily handle the problem in some other way, it is best to avoid throwing exceptions.

Generally, you will want to separate throwing the exception and catching the exception into separate functions.

In most cases, you should include any throw statement within a function definition, list the exception in the exception specification for that function and then place the catch clause in a different function.

When to throw an exception in C++:

This means that you have your exception thrown and its specification included in one function:

```
void  functionA() throw (MyException)
{
            .
            .
            .
      throw MyException(<Maybe an argument>);
            .
            .
            .
}
```

When to throw an exception in C++:

Then in another function (that may even be in another file, you have:

```
void functionB()
{
        .
        .
        .
    try
    {
        .
        .
        .
      functionA();
        .
        .
        .
    }
    catch(MyException e)
    {
        <Handle exception.>
    }
        .
        .
        .
}
```

<u>Other notes on exceptions:</u>

Every exception that is thrown by your code should be caught someplace in your code.

- Otherwise the program ends (and it is not good style).

Exceptions allow you to write programs whose flow of control is so involved that it is difficult to impossible to understand the program.

- It is not hard to do and allows you to transfer flow of control of your program from anyplace in your program to almost anyplace else in your program.
  - This sort of unrestricted flow was supported using goto statements (back when dinosaurs roamed the earth).
  - However, unrestricted flow is very difficult to read/follow and very poor programming style.
  - Since catch-throw statements allow you to revert to this<u>*, you should only use them when you have no other choice that produces reasonable code.*</u>

Other notes on exceptions:

You also can place a try block and following catch block(s) inside a larger try block or inside a larger catch block.

- In rare cases this may be useful, but if you are thinking of doing this, there is likely a better way to organize your program.
- It is almost always better to place the inner try-catch blocks inside a function definition and place an invocation of the function in the outer try or catch block.
- Ideally, it would be better to just eliminate one or more of the nested try/catch blocks completely
- If you have a try-catch block nested inside a larger try block, when an exception is thrown inside the inner try block, but not caught in the inner catch block(s), then the exception is throw to the outer try block for processing an might be caught there.

Exception Class Hierarchies:

Exception Class hierarchies assume that it is useful to have a base exception class (e.g. Arithmetic Error) from which you might wish to define derived classes. (e.g. DivideByZeroError).

This means that every catch block for an ArithmeticError will also catch a DivideByZeroError.

Also, if you list ArithmeticError in an exception specification, then you have in effect also added DivideByZeroError to the exception specification (along with any other derived classes from ArithmeticError), whether or not you list DivideByZeroError by name.

Testing for Available Memory:

Remember when we created new dynamic variables with code such as:

```
struct Node
{
    int data;
    Node *link;
};
typedef Node* NodePtr;
    . . .
NodePtr pointer = new Node;
```

As long as there is sufficient memory available to create the node, it is fine.

However, when there is insufficient memory, then `new` will throw a `bad_alloc` exception.

- This is part of the C++ language so you do not need to define it.

Testing for Available Memory:

Since `new` will throw a `bad_alloc` exception, you can use it to check if you are running out of memory::

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

Obviously, you can do other things besides giving a warning message, but the details of how you would handle this situation depend on the programming situation.

Rethrowing an Exception:

It is legal to throw an exception within a catch block.

You (rarely) may want to catch an exception and then (depending on the details) decide to throw the same or a different exception for handling it farther up the chain of exception-handling blocks.

Final summary:

You won't have much cause to use this mechanism in our class because the code is too simple.

However, it is an important abstraction for larger programs (e.g. operating systems, databases), so you need to know about it.

Review Questions for Slide Set 9

- When an error occurs in a program, sometimes you may want to terminate and sometimes you may wish to choose for data re-entry. Give an example for each situation.
- What is the difference between error handling and exception handling? Give an example of each situation?
- Why do you need to have exception handling? Give examples of when you might need exception handling?
- Explain when and why you might want to throw an exception.
- Explain why and how you would handle an exception.
- What are the functions of “try”, “catch”, and “throw” in exception handling?

Review Questions for Slide Set 9

- What is the name for the thrown value?
- What is the term for executing a throw statement?
- Do thrown values have to be of a specific type?
- What do you throw to a catch statement in C++?
- What is execution of the catch block called?
- What happens if you throw an exception and there is no catch block?
- Is the catch block a function? Be able to compare and contrast a catch block and function definition
- Compare and contrast a throw call and a function call.
- What is another name for an exception handler?

Review Questions for Slide Set 9

- What does the catch-block parameter do?
- When an exception is thrown, is the rest of the code in the try block executed?
- Does a catch block apply to any try block?
- If no exception is thrown, does the catch block execute?
- Since an exception can only throw one value, how do you work around this?
- Can you call a constructor when you throw an exception?
- Can a try throw block throw multiple different exceptions with the same or different types of exception values?
- During the execution of a try block, how many different exceptions can be thrown? Explain why?

Review Questions for Slide Set 9

- How many different types of values can a single catch block catch?
- Is it legal to have a catch block with a type but no parameter?
- Does the order of catch blocks matter?
- What is the structure of a catch statement that can catch a thrown value of any type?
- Can you throw an exception in a function? If yes, do you have to catch it in the same function? If not, how do you warn programmers that if the function is invoked, it might throw an exception?
- What is an exception specification?
- What needs to happen with the exception specification if a function is overloaded?

Review Questions for Slide Set 9

- What is another name for the exception specification?
- Can a function throw more than one type of exception?
- What if a function has no exception specification?
- What happens when an exception is thrown in a function, but is not listed in the exception specification?
- If you catch an exception inside of a function, does it need to be in the exception specification list?
- How do you indicate that a function should not throw any exceptions that are not caught inside the function?
- Do exceptions support automatic type conversions?
- When redefining/overriding a function from the base class, can you add or remove exceptions from the exception specification? Why?

Review Questions for Slide Set 9

- When should you throw an exception in C++?
- What happens if you throw an exception and it isn’t caught?
- Can you nest try/catch blocks?
- Can you rethrow exceptions?