

CMSC202

Computer Science II for Majors

Lecture 16 – Exceptions

Dr. Katherine Gibson

- Inheritance
- Polymorphism
- Virtual functions
 - Abstract Classes

- Exam 2

Any Questions from Last Time?

- Error handling
- Exceptions
- Defining exception classes
- Using exceptions
 - Try
 - Throw
 - Catch
- When to throw exceptions

Error Handling

- We have seen a number of error types:
 - Could not allocate memory
 - Out-of-bounds on vector
 - File not found/could not be opened

 - Attempting to add a train car that's not allowed
 - A poker hand with invalid cards

- How are these errors handled?
 - Print a message
 - “You cannot add a second Snack Car”
 - Do nothing
 - Exit the program
- The errors are handled right where they occur

- Advantages:
 - Easy to find because code is right there
- Disadvantages:
 - Error handling scattered throughout code
 - Code duplication
 - Code inconsistency (even worse!)
 - Errors are handled however the original coder decided would be best

- Class *implementer*
 - Creates the class definition
 - Knows what constitutes an error
 - Decides how to handle errors
- Class *user*
 - Uses the class implementation
 - Knows how they want to handle errors
 - (But if handled internally, the class user may not even know an error occurred)

- Want to separate errors into two pieces:
 - ***Error detection***
 - Implementer knows how to detect
 - ***Error handling***
 - User can decide how to handle
- Use ***exceptions*** to do this

Exceptions

- ***Exceptions*** are used to handle exceptional cases
 - Cases that shouldn't occur normally
- Allow us to indicate an error has occurred without explicitly handling it
 - C++ uses these too, like when we try to use `.at()` to examine an out-of-bounds element

- Exceptions are implemented using the keywords try, throw, and catch

- Exceptions are implemented using the keywords **try**, **throw**, and **catch**
- The **try** keyword means we are going to try something, even though we are not sure it is going to perform correctly

- Exceptions are implemented using the keywords `try`, **`throw`**, and `catch`
- The **`throw`** keyword is used when we encounter an error
- Means we are going to “throw” two things
 - A value (explicit)
 - Control flow (implicit)

- Exceptions are implemented using the keywords `try`, `throw`, and **catch**
- The **catch** keyword means we are going to try to catch at most **one** type of value
 - To catch different types of values, we need multiple catch statements


```
// inside SetCarID() function
```

```
if (newID < MIN_ID_VAL ||  
    newID > MAX_ID_VAL) {  
    cerr << "ID invalid, no change";  
}
```

```
// inside SetCarID() function
try {
    if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
        cerr << "ID invalid, no change";
    }
}
catch () {

}
```

```
// inside SetCarID() function
try {
    if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
        throw(newID) ;
    }
}
catch () {

}
```

```
// inside SetCarID() function
try {
    if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
        throw(newID);
    }
}
catch (int ID) {

}
```

```
// inside SetCarID() function
try {
    if (newID < MIN_ID_VAL ||
        newID > MAX_ID_VAL) {
        throw(newID);
    }
}
catch (int ID) {
    cerr << "ID invalid, no change";
}
}
```

Catching and Throwing

- The **catch** keyword requires:
 - One parameter
 - Typename (int, exception, out_of_range, etc)
 - Name (newID, e, oor, etc.) [optional]
- To catch multiple types of exceptions, you need to use multiple ***catch blocks***

- You can throw from inside a catch block
- But this should be done sparingly and only after careful consideration
 - Most of the time, a nested try-catch means you should re-evaluate your program design
- Uncaught exceptions will cause the **terminate ()** function to be called

- Catch blocks are run in order, so exceptions should be caught in order from
 - Most specific to least specific
- To catch all possible exceptions, use:
`catch (. . .)`
- (Literally use three periods as a parameter)

- We can throw exceptions without try/catch
 - Most commonly done within functions
- Requires that we list possible exception types in the function prototype and definition
 - Called a *throw list*

- Warn programmers that functions throw exceptions without catching them
- Throw lists should match up with what is thrown and not caught inside the function
 - Otherwise, it can lead to a variety of errors, including the function **unexpected()**
- Can also have empty throw lists for clarity:

```
int GetCarID() throw ();
```

- Functions can specify their throw lists

```
// Throws only 1 type of exception
```

```
retType funcName( params ) throw (excep);
```

```
// Throws 2 types of exceptions (comma separated list)
```

```
retType funcName( params ) throw (excep1, excep2);
```

```
// Promises not to throw any exceptions
```

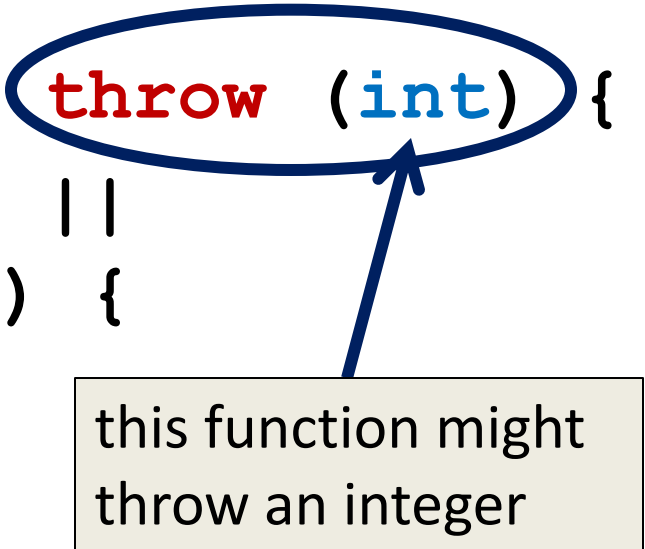
```
retType funcName( params ) throw ( );
```

```
// Can throw any exceptions [backwards compatibility]
```

```
retType funcName( params );
```

Throw List Example: Inside

```
void SetCarID (int newID) throw (int) {  
    if (newID < MIN_ID_VAL ||  
        newID > MAX_ID_VAL) {  
        throw (newID) ;  
    }  
    else {  
        m_carID = newID ;  
    }  
}
```



this function might
throw an integer

```
// inside main()
```

```
train.at(0).SetCarID(-1);
```

- What will happen if we run this code?
 - The exception won't be caught
 - The `terminate()` function will be called

```
// inside main()
```

```
try {  
    train.at(0).SetCarID(-1);  
  
} catch (int ID) {  
    cerr << "ID invalid, no change";  
}
```

this user has based their code
on getting input from a file

```
// inside main()
while(set == false) {
    try {
        train.at(0).SetCarID(userID);
        set = true;
    } catch (int ID) {
        cerr << "ID" << ID
            << "invalid, give another";
        cin >> userID;
    }
}
```

this user has based their code on getting input from a user, and being able to repeat requests

Exception Classes

- We can create, throw, and catch exception classes that we have created
- We can even create hierarchies of exception classes using inheritance
 - Catching the parent class will also catch all child class exceptions

```
class MathError { /*...*/ };
```

```
class DivideByZeroError:  
    public MathError { /*...*/ };
```

```
class InvalidNegativeError:  
    public MathError { /*...*/ };
```

- Name of class reflects the error
 - Not the code that throws error
- Contains basic information or a message
 - Parameter value
 - Name of function that detected error
 - Description of error
- Methods required
 - Constructor (one or more)
 - Accessor (one or more)

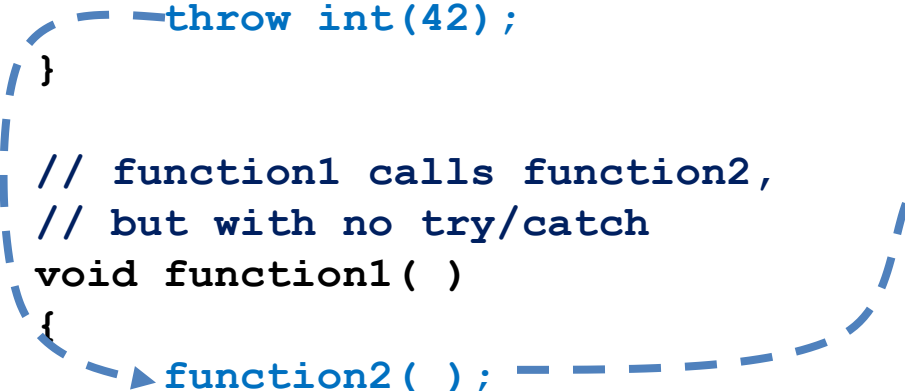
Nested Functions?

```
// function2 throws an exception
void function2( )
{
    cout << "function2" << endl;
    throw int(42);
}

// function1 calls function2,
// but with no try/catch
void function1( )
{
    function2( );
    cout << "function1" << endl;
}
```

```
// main calls function1,
// with try/catch
int main( )
{
    try {
        function1( );
    }
    catch (int)
    {
        cout << "Exception"
            << "occurred"
            << endl;
    }

    return 0;
}
```



What happens here?

Stack is unwound until something catches the exception OR until unwinding passes main

What happens then?

- Best way to handle Constructor failure
 - Replaces Zombie objects!
 - Any sub-objects that were successfully created are destroyed (destructor is *not* called!)
- Example:

```
// MyClass constructor
MyClass::MyClass ( int value )
{
    m_pValue = new int(value);

    // pretend something bad happened
    throw NotConstructed( );
}
```

- Bad, bad idea...
 - What if your object is being destroyed in response to another exception?
 - Should runtime start handling your exception or the previous one?
- General Rule...
 - Do not throw exceptions in destructor

- Project 4 is out!
- We'll go over Exam 2 next time