# Exceptions Continued

# Function try blocks

- When an exception is thrown from within a constructor 's initialization list, the compiler will call the destructor for any of the data members of that object that have already been constructed.

  - ```
    MyC::MyC( int n )
      : mField1(n), mField2( n+1)        // This is the initialization list
    { }
    ```

- Also, if you want to catch that exception, you can use a **function try block**. This looks a little different from the regular try catch.
    ```

# Function try block

```
class MyC
{
public:
        MyC::MyC(int n)
        try                     // function try block
          : mc2(n)
        {

        }
        catch(...)
        {
                cout << "Caught in MyC constructor " << endl;
        }

private:
        MC1 mc1;
        MC2 mc2;
};
```

See code example in IDE.

# Exception guarantees

There are three levels here:

- **Basic guarantee**:

  This says that if an exception is thrown, then the objects are in a usable state and can be destructed (you would expect this at least, right… that's why this is a basic guarantee), and no resources are leaked.

- **Strong guarantee**:

  This says that if an exception is thrown, then the state of the objects remains unchanged. This is not easy to achieve. It is similar to the commit-rollback guarantee that you would see in transactional systems, like databases.

# Exception guarantees

- Strong guarantee is provided by list insert (single element) and by push_back on a container.
- Insert methods that insert multiple elements do not provide this guarantee because a reasonable implementation would not be able to achieve this ( it would need to remove all the elements that were added up until the point the exception was thrown).
- **No throw guarantee**:

  This says that the code providing this guarantee will not throw. swap and pop_back provide this guarantee.
- Your class destructors should provide this guarantee, as it is not a good idea for your destructors to throw an exception.

# Exception guarantees

- If you want to specify that a method does not throw any exception, you provide the exception specification list and keep it empty.

- Example:

```
void Foo( ) throw();

OR
void Foo() noexcept;      // C++11

OR
void Foo() noexcept( true ); // C++11
```

# Exceptions

- In the operators lesson, we saw that you should be careful when implementing assignment operators… for that reason, if you have to, say, allocate memory for a data member, you don't first delete the existing memory and then allocate new memory (because the new allocation might fail, in which case you don't want the previous memory to have been deleted).

- The examples we looked at would allocate memory first and do the copying of data, etc., and then would delete existing memory and assign the new memory.

# Exceptions

- Or another way to implement the assignment operator would be to use swap semantics, as below:

```
MyClass & MyClass::operator=( const MyClass & other )
{
   MyClass temp( other );          // use copy constructor to create copy
   Now swap this with temp
   return *this;
}
```

- OR even better:

```
MyClass & MyClass::operator=( MyClass other ) // pass by value creates a copy
{
   Now swap this with other
   return *this;
}
```

# vector push_back

- We had talked about the vector class earlier, and also about strong exception guarantees.
- vector's push_back member function provides a strong exception guarantee.
  - It can do that because it is only appending at the end.
  - And in case it has to reallocate to grow,
    - it can allocate new memory,
    - copy the objects over,
    - If no exception so far, then delete old memory at the end and make vector internals to point to new memory.

# vector push_back

- But with move semantics, you have to keep some things in mind.

- If push_back reallocates and now "moves" the existing objects to the new memory, then it is possible that the move throws an exception.
- And if the exception happens, its original memory block has changed, since it moved the objects from original memory to new memory.
- And so the strong exception guarantee cannot be offered in this case.

- Now, if the move constructor is declared as noexcept (i.e., it does not throw), then the strong guarantee can be given.
- push_back calls move_if_noexcept.

- So, strong guarantee is given if the objects are no throw move constructible, or copy constructible.

# std:: move_if_noexcept

- std::move_if_noexcept
  - Cast to rvalue if:
    - Move constructor does not throw (is declared as noexcept)
    - Or the type is not copy constructible.
  - Else
    - return as lvalue

# noexcept

- noexcept is used to specify that a function does not throw an exception.
- Specifying a function as noexcept can enable some compiler optimizations.
- template <typename T> void Foo ( const char * ) noexcept (true);
  - Same as: template <typename T> void Foo ( const char * ) noexcept;
- template <typename T> void Foo ( const char * ) noexcept (false);
  - Same as : template <typename T> void Foo ( const char * )

# noexcept

- noexcept can take an expression as well:

```
template <typename T> void Foo( T & t1 ) noexcept ( noexcept ( ++ t1)  );

template <typename T> void Foo ( T & t1 ) noexcept ( noexcept (*t1)  );

struct pair
{
   void swap( pair & other ) noexcept ( noexcept(swap(first, other.first)
                                          && swap(second, other.second ) ) );
};
```

# Termination & resumptive model

- In the termination model, the execution is not resumed at the point (or after the point) where the exception occurred.
- This is when the stack unwinding happens and control is transferred to the caller function, and this goes up the calling stack, looking for a handler (catch).

- In the resumption model, the code is expected to handle the exception and then retry the code that caused the exception.

- This means that the code needs to be in some loop, so that in case of an exception, the code is retried (after taking some action that you expect will fix the problem that caused the exception to be throw).

# Termination & resumptive model

```
int             retryCount = 0;
bool            done = false;
const int       kMaxRetryCounts = 3;            // or whatever value makes sense in ur application

while (   ! done   &&    ( retryCount < kMaxRetryCounts)   )
{
        try
        {
                Foo();
                done = true;
        }
        catch ( Exception  &  e )
        {
                ++ retryCount;
                DoSomething();          //  do something that might fix the problem, and try again
        }
}
```