CS-202

Exceptions

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	LAST Project	PASS Session		PASS Session

Your 10th Project Deadline is this Wednesday 5/1.

- PASS Sessions held Friday-Sunday-&-Monday-Tuesday, get all the help you need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

Today's Topics

Separating Error–Detection & Error–Handling

C++ Exceptions

- The Zero-Cost Model

The `try`–`catch` Block – The `throw` Expression

- Exception Objects
- Stack Unwinding

Exception(s) from Functions

- Prior to C++11 – The `throw` Lists
- Post C++11 – The `noexcept` specifier

Exception Classes

Exception(s) in `ctor`(s) – Exception(s) in `dtor`(s)

Common Errors

There can be a wide variety of Error "types" in a program

- Could not allocate enough memory:

```
HugeDataStruct * hDS_array = new HugeDataStruct [HUGE_NUM];
```

- Going out-of-bounds on a Vector:

```
std::vector<int> intVec(100, -1);
int intVal = intVec.at(100);
```

- File Input failure:

```
std::ifstream inFile;
inFile.exceptions ( std::ifstream::failbit | std:: ifstream::badbit );
inFile.open ("a_non_existent_file.txt");
```

- Passing unreasonable values:

```
Car myCar;
myCar.setLicensePlates("#$@&%*!");
```

Error Handling

Common practices for error handling (so far)

- Print out a message:

```
cerr << "Not enough memory, allocation requires freeing resources";
```

- Pass over control:

```
cerr << "File not found or other error occurred, try again:";
cin >> inputFileName;

cerr << "License Plates cannot carry special characters, try again:";
cin >> licensePlatesString;
```

- Do nothing:

…

- Crash and burn (hopefully) / Get weird results:

```
std::vector<int> intVec(100, -1);
cout << intVec[100];
```

Undefined Behavior:
- Might output 0, 1, -3, …
- Might produce Segmentation Fault.

Error Handling

Errors are handled where they occur

Advantages:

- Code is right there – Easy to find.

Disadvantages:

- Error handling is scattered throughout source code.
- Code duplication.
- (or worse) Code inconsistency.
- Error handling is performed strictly as per the original programmer's specifications.

Error Handling

Using Functions/Classes & Handling Errors

Programmer #1 – The function / class Implementer:

- Devises and writes the definition(s) – Knows what constitutes an Error.
- Decides appropriate Error-handling behavior.

Programmer #2 – The function / class User:

- Uses the implementation(s) - Knows how they want to handle Errors.
- If Error-handling is internally hard-coded into the function / class source code, the User has no say.
  (Might not even be exposed to the fact that an error occurred.)

Error Handling

Error–Detection & Error–Handling

Error–Detection:

- Up to Programmer #1 – The function / class Implementer.
- Devises and writes the definition(s) – Knows what constitutes an Error.

Error–Handling:

- Up to Programmer #2 – The function / class User.
- Uses the implementation(s) - Knows how they want to handle Errors.

Exceptions is a mechanism to separate Error–Detection & Error–Handling.

Error Handling

Error–Handling via Exceptional Cases

Exceptions are used to handle Errors as "Exceptional Cases":

- Program structure / execution treats them as cases that shouldn't "normally" occur.

Exceptions allow separation of Error–Detection from Error–Handling:

- Indicate an Error has occurred (without being required to explicitly handle it at occurrence).

C++ (unhandled) Exception example case – Vector out-of-bounds indexing:

```
std::vector<int> intVec(100, -1);
cout << intVec.at(100);
```

```
terminate called after throwing an instance of
'std::out_of_range'  what(): vector::_M_range_check
```

Exception(s) Implementation

The keywords `try` & `catch`

The `try` – `catch` Block:

- Associates one or more Exception-Handler(s) with a specific Compound Statement.
- `try` opens a Block for the compound statement to be evaluated. Essentially it means the code in the Block Scope is going to try something, which might not perform correctly.
- `catch` is placed after that Block, and stands ready to catch at most one object type (an object of type `std::exception`, or `int`, or `Car`, etc…)
  To catch different object types, multiple `catch` statements are needed.

Exception(s) Implementation

The keywords **try** & **catch**

The **try** – **catch** Block Syntax:

Try to run the code in this Block Scope, but an error condition might occur.

```
try {
 /*a sequence of code statements that might work erroneously*/
}
catch (const int & err_code) {
 /*code handling int object type*/
}
catch (const std::exception & ex) {
 /*code handling std::exception object type*/
}
catch ( … ) {
 /*code that is triggered by any object type*/
}
```

Exception(s) Implementation

The keywords **try** & **catch**

The **try** – **catch** Block Syntax:

```
try {
 /*a sequence of code statements that might work erroneously*/
}
catch (const int & err_code) {
 /*code handling int object type*/
}
catch (const std::exception & ex) {
 /*code handling std::exception object type*/
}
catch ( … ) {
 /*code that is triggered by any object type*/
}
```

Just one, or multiple **catch** blocks to handle multiple Exception object types.

Exception(s) Implementation

The keywords **try** & **catch**

The **try** – **catch** Block Syntax:

```
try {
 /*a sequence of code statements that might work erroneously*/
}
catch (const int & err_code) {
 /*code handling int object type*/
}
catch (const std::exception & ex) {
 /*code handling std::exception object type*/
}
catch ( … ) {
 /*code that is triggered by any object type*/
}
```

If within the code of the **try** Block an error is detected, and an **int** type Exception is produced, Error-handling based on this **int** will occur here.

The same process but for Exception Objects of **std::exception** type.

Generic Handler triggered by any Exception Object type.

Exception(s) Implementation

The `throw` Expression

Exception Throwing is associated with Error–Detection. Flow Control passes over to Exception-Handling, alongside some type of **Exception Object**.

- `throw` performs copy-initialization to derive the Exception object.
  (The Exception object is a temporary with top-level `cv`-qualifiers removed.)

Exception(s) Implementation

The `throw` Expression

Exception Throwing is associated with Error–Detection. Flow Control passes over to Exception-Handling, alongside some type of **Exception Object**.

- `throw` performs copy-initialization to derive the Exception object.
  (The Exception object is a temporary with top-level `cv`-qualifiers removed.)
- After that Program "*Stack Unwinding*" takes place:
  Control Flow moves backwards and calls the Destructors of every object of `auto` Storage-Duration that has been constructed, in reverse order of their completion.
  Note : Not for objects Constructed by a `new` Expression (*Dynamic* Storage-Duration) !

Exception(s) Implementation

The `throw` Expression

Exception Throwing is associated with Error–Detection. Flow Control passes over to Exception-Handling, alongside some type of **Exception Object**.

- `throw` performs copy-initialization to derive the Exception object.
  (The Exception object is a temporary with top-level `cv`-qualifiers removed.)
- After that Program "*Stack Unwinding*" takes place:
  Control Flow moves backwards and calls the Destructors of every object of `auto` Storage-Duration that has been constructed, in reverse order of their completion.

Sub–Note :
If the Exception is **thrown** from **within a Constructor** that was invoked by a `new` Expression, the matching Deallocation Function is called, if available.
- i.e. `operator delete` (`[]`)

Exception(s) Implementation

The `throw` Expression

Exception Throwing is associated with Error–Detection. Flow Control passes over to Exception-Handling, alongside some type of **Exception Object**.

- `throw` performs copy-initialization to derive the Exception object.
  (The Exception object is a temporary with top-level `cv`-qualifiers removed.)
- After that Program "*Stack Unwinding*" takes place:
  Control Flow moves backwards and calls the Destructors of every object of `auto` Storage-Duration that has been constructed, in reverse order of their completion.
  Note : Not for objects Constructed by a `new` Expression (*Dynamic* Storage-Duration) !
- When "*Stack Unwinding*" reaches a `try` Block, control is transferred to the `catch` corresponding to the thrown Exception Object type.
  If none appropriate is found, the control flow continues to "*Unwind*"

Exception(s) Implementation

The `throw` Expression

The `throw` Syntax:

```
/*a block scope somewhere*/
{
  throw _expression_ ;
}
```

Or

```
/*a block scope somewhere*/
{
  throw ;
}
```

Throwing can happen from inside a Function Block, a Class Method Block, an Unnamed Block, a `try-catch` Block, a Constructor, a Destructor (rarely), …

Exception(s) Implementation

The `throw` Expression

The `throw` Syntax:

```
/*a block scope somewhere*/
{
   throw _expression_ ;
}
```

Evaluates the value of **`_expression_`** and uses it to copy-initialize an Exception Object of the same type (Copy-`ctor` of the type must be available).

Or

```
/*a block scope somewhere*/
{
   throw ;
}
```

Abandons current `catch` Block and re-`throw`s the currently handled Exception object (the exact same – not a copy).

Exception(s) Implementation

The `throw` Expression

The `throw` Syntax:

```
/*a block scope somewhere*/
{
   throw _expression_ ;
}
```

Evaluates the value of **`_expression_`** and uses it to copy-initialize an Exception Object of the same type (Copy-`ctor` of the type must be available).

Note:
"Stack Unwinding" after Exception Object Construction wipes away everything until it reaches a `try` Block. The only object that persists until reaching is the (temporary) Exception Object.

Or

```
/*a block scope somewhere*/
{
   throw ;
}
```

Abandons current `catch` Block and re-`throw`s the currently handled Exception object (the exact same – not a copy).

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL)
      cerr << "Requested ID is invalid, nothing changed…";
    else
      m_carID = id;
}
```

Or (overriding flow):

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL){
      cerr << "Requested ID is invalid, nothing changed…";
      return;
    }
    m_carID = id;
}
```

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL) {
      throw (id) ;
    }
    m_carID = id;
}
…
  Car myCar;
  try{
    myCar.setCarID(-1);
  }
  catch(const int & ex_id){
    cerr << "Requested ID is invalid, nothing changed…";
  }
```

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
     if (id < MIN_ID_VAL || id > MAX_ID_VAL) {
        throw (id) ;
     }
     m_carID = id;
}
…
   Car myCar;
   try{
     myCar.setCarID(-1);
   }
   catch(const int & ex_id){
     cerr << "Requested ID is invalid, nothing changed…";
   }
```

a) Erroneous case is detected, `throw` evaluates expression: `(id)` and derives a copy-initialized Exception Object of type `int`.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL) {
        throw (id) ;
    }
    m_carID = id;
}
…
  Car myCar;
  try{
    myCar.setCarID(-1);
  }
  catch(const int & ex_id){
    cerr << "Requested ID is invalid, nothing changed…";
  }
```

b) Stack is Unwound until first `try` Block is encountered.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL) {
        throw (id) ;
    }
    m_carID = id;
}
…
  Car myCar;
  try{
    myCar.setCarID(-1);
  }
  catch(const int & ex_id){
    cerr << "Requested ID is invalid, nothing changed…";
  }
```

c) Control is passed to matching-type `catch` Block.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Error – Handling By-Example:

```
void Car::setCarID(int id){
    if (id < MIN_ID_VAL || id > MAX_ID_VAL) {
        throw (id) ;
    }
    m_carID = id;
}
…
  Car myCar;
  try{
    myCar.setCarID(-1);
  }
  catch(const int & ex_id){
    cerr << "Requested ID is invalid, nothing changed…";
  }
```

Note: Recommended practice is to throw by-Value, catch by-Reference.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Stack Unwinding & Exception Object(s) By-Example:

```
void Car::setLicensePlates(const char * lPlates){
  try{
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("abcdef…ABCDEF…0123456789") {
      throw (lPlates_str) ;
    }
    m_licensePlates = lPlates_str;
  }
  catch(const std::string & ex_lp){
    cerr << "Plates " << ex_lp << "  contain invalid characters…";
  }
}
```

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Stack Unwinding & Exception Object(s) By-Example:

```
void Car::setLicensePlates(const char * lPlates){
  try{
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("abcdef…ABCDEF…0123456789") {
      throw (lPlates_str) ;
    }
    m_licensePlates = lPlates_str;
  }
  catch(const std::string & ex_lp){
    cerr << "Plates " << ex_lp << "  contain invalid characters…";
  }
}
```

Stack Unwinding will first destroy ***lPlates_str*** before reaching the first `try` Block.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Stack Unwinding & Exception Object(s) By-Example:

```
void Car::setLicensePlates(const char * lPlates){
  try{
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("abcdef…ABCDEF…0123456789") {
      throw (lPlates_str) ;
    }
    m_licensePlates = lPlates_str;
  }
  catch(const std::string & ex_lp){
    cerr << "Plates " << ex_lp << "  contain invalid characters…";
  }
}
```

Stack Unwinding will first destroy ***lPlates_str*** before reaching the first `try` Block.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Stack Unwinding & Exception Object(s) By-Example:

```
void Car::setLicensePlates(const char * lPlates){
  try{
    std::string lPlates_str( lPlates );
    if (lPlates_str.find_first_not_of("abcdef…ABCDEF…0123456789") {
      throw (lPlates_str) ;
    }
    m_licensePlates = lPlates_str;
  }
  catch(const std::string & ex_lp){
    cerr << "Plates " << ex_lp << " contain invalid characters…";
  }
}
```

But the temporary Exception Object copy-initialized from expression ***lPlates_str*** is propagated in catch Block(s), and can be manipulated within them.

Exception(s) Implementation

The `try` – `throw` – `catch` Paradigm

Combining with *Flow Control* By-Example:

```
int main(){
   Car myCar;
   while ( !myCar.IDisSet() ){
      int id_input;
      cin << id_input;
      try{  myCar.setCarID(id_input);  }
      catch(const int & ex_id){
        cerr << "Requested ID" << ex_id << "is invalid, retry…";
      }
   }
}
```

```
void Car::setCarID(int id){
  if (id<MIN_ID_VAL || id>MAX_ID_VAL)
    throw (id) ;
  m_carID = id;
}
```

Exception(s) Implementation

Exception–Handling *Zero-Cost* Model

"Simple" way:

```
bool Car::setCarID(int id){
 if (id<MIN_ID_VAL || id>MAX_ID_VAL){
  cerr << "Invalid ID…";
  return false;
 }
 else
  m_carID = id;
 return true;
}

Car myCar;
if ( !myCar.setCarID(-1) ){
 /*handle error in main program flow*/
}
```

Exception(s) Implementation

Exception–Handling *Zero-Cost* Model

With Exceptions:

```
bool Car::setCarID(int id){
 if (id<MIN_ID_VAL || id>MAX_ID_VAL){
  cerr << "Invalid ID…";
  return false;
 }
 else
  m_carID = id;
 return true;
}

Car myCar;
if ( !myCar.setCarID(-1) ){
 /*handle error in main program flow*/
}
```

```
void Car::setCarID(int id){
  if (id<MIN_ID_VAL || id>MAX_ID_VAL)
   throw (id) ;
  m_carID = id;
}
…
 Car myCar;
 try {  myCar.setCarID(-1); }
 catch (const int & ex_id) {
  cerr << "Invalid ID…";
 }
```

Exception-Handling code lies at a **vtable** *outside* of main program flow !

Exception(s) Implementation

Catching (keyword **catch**) Semantics

The keyword **catch** requires:

- One parameter, given as usual

  _type_name_ (**int**, **std::exception**, **std::out_of_range**, …) and
  _param_name_ (**ex_id**, **ex**, **ex_oor**, …, name is optional)

Or:

```
catch (const int & ex){ /*handling & manipulating ex*/ }
catch (const int &)    { /*handling*/ }
```

- Special parameter (…) – (three periods)

  Allows catching any type of Exception Object.

```
catch ( … ){ /*handling for any possible thrown exception*/ }
```

Exception(s) Implementation

Catching (keyword **catch**) Semantics

To catch different Exception types, multiple **catch** Blocks are required:

```
try{
  …
 if ( !validIDInput(id) ){  throw (id);  }
  …
 if ( !validLicencePlatesInput(lPlates_str) ){ throw (lPlates_str) }
  …
}
catch( const int & ex_id){
 cerr << "Input id" << ex_id << "is over/under the allowed limits…";
}
catch( const std::string & ex_plates){
 cerr << "Input Plates " << ex_plates << "contain invalid characters…";
}
catch( … ){ cerr << "Unknown exception caught …"; }
```

Exception(s) Implementation

Throwing (**throw** expression) Semantics

(Re-)**throw**ing from inside a **catch** Block is allowed

- However, nested **try**–**catch** Blocks often require reconsideration.

```
try{
 /* try top-level statements here … */
 if ( !goodObjectCondition(myClassObject) ){ throw myClassObject; }
 …
}
catch ( const myClassObject & ex_mc ){
 try{
  /* nested try (in catch) statements here … */
  if ( logErrorCondition() ){ throw std::ios_base::failure("(°5°)"); }
 …
 }
}
```

Example #1: Catches this, but *might* also (re-)throw that.

Exception(s) Implementation

Throwing (**throw** expression) Semantics

(Re-)**throw**ing from inside a **catch** Block is allowed

- However, nested **try**–**catch** Blocks often require reconsideration.

```
try{
 /* try top-level statements here … */
 if ( !goodObjectCondition(myClassObject) ){ throw myClassObject; }
 …
}
catch ( myClassObject & ex_mc ){
 try{
  /* nested try (in catch) statements here … */
  ex_mc.setMemberValue(1.0);
  if ( logErrorCondition() ){ throw ; }
 }
}
```

Example #2: Catches this,
but *might* also (re-)throw it (the same Object).

Exception(s) Implementation

Throwing (**throw** expression) Semantics

(Re-)**throw**ing from inside a **catch** Block is allowed

- However, nested **try**–**catch** Blocks often require reconsideration.

```
try{
 /* try top-level statements here … */
 if ( !goodObjectCondition(myClassObject) ){ throw myClassObject; }
 …
}
catch ( myClassObject & ex_mc ){
 try{
  /* nested try (in catch) statements here … */
  ex_mc.setMemberValue(1.0);
  if ( logErrorCondition() ){ throw ; }
 }
}
```

Example #2: Catches this, but *might* also (re-)throw it (the same Object).

Note: Exception Object in this Handler is non-**const** → can modify it.

Exception(s) Implementation

Throwing (**throw** expression) Semantics

throwing from inside a **catch** Block is allowed

- However, nested **try**–**catch** Blocks often require reconsideration.

```
try{
 /* try top-level statements here … */
 if ( !goodObjectCondition(myClassObject) ){ throw myClassObject; }
 …
}
catch ( myClassObject& ex_mc ){
 try{
  /* nested try (in catch) statements here … */
  ex_mc.setMemberValue(1.0);
  if ( logErrorCondition() ){ throw ; }
 }
}
```

Example #2: Catches this, but *might* also (re-)throw it (the same Object).

Note: Do not use **throw ex_mc;** → Will copy-initialize Exception Object again and delete –call Destructor of– original (+*Slicing* Possibility).

Exception(s) Implementation

Throwing (`throw` expression) Semantics

`throw`ing and not `catch`ing results in Uncaught Exception(s)

- Exception Throwing without guaranteeing that during Stack Unwinding a `try`–`catch` Block will be encountered.
- Leaving Exceptions Unhandled – Not guaranteeing that a matching Exception type `catch` Block will be found.

Uncaught / Unhandled Exceptions will cause the `std::terminate()` function to be called.

Exception(s) Implementation

Throwing (`throw` expression) Semantics

`throw`ing from Functions

- Exception Throwing can be performed anywhere in the code.
- Compiler allows (does not check) for a matching `try`–`catch` Block.
- Typically used to `throw` from inside a Function.

Requirements – prior to C++11 Standard:

- List all possible Exception Types thrown in the Function Prototype and Definition – The Dynamic Exception Specification (also called Function `throw` List).

Exception(s) Implementation

Throwing (`throw` expression) Semantics

Function `throw` Lists – prior to C++11 Standard:

- To warn programmer that a Function might `throw` an Exception (it will not `catch` and handle it internally, and it will propagate).

`throw` Lists should match up with what is thrown and not caught inside the function. Otherwise, the function `std::unexpected()` is called:

- Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an exception-specification that does not allow the exception, then, if the exception-specification is a dynamic-exception-specification, the function `std::unexpected()` is called (15.5.2), otherwise, the function `std::terminate()` is called (15.5.1).

Exception(s) Implementation

Throwing (`throw` expression) Semantics

Function `throw` Lists – prior to C++11 Standard:

Can be of type
`int`, `std::string`,
`std::exception`,
`std::bad_alloc`,
`std::ios_base_failure`, …

```
// Throws only 1 type of exception
_ret_type_ myFunc1( _params_list_ ) throw (_ex_type_);

// Throws 2 types of exceptions (comma separated list)
_ret_type_ myFunc2( _params_list_ ) throw (_ex_type_1_, _ex_type_2_);

// Promises not to throw any exceptions
_ret_type_ myFunc0(_params_list_ ) throw ( );

// Can throw any exceptions
_ret_type_ myFunc(_params_list_ );
```

Exception(s) Implementation

Function **throw** Lists – prior to C++11 Standard:

By-Example:

```
void Car::setCarID(int id) throw(int) {
  if (id < MIN_ID_VAL || id > MAX_ID_VAL) {  throw (id) ;  }
  m_carID = id;
}

int main(){
  Car myCar;
  try{  myCar.setCarID(-1); }
  catch(const int & ex_id){
   cerr << "Requested ID" << ex_id << "is invalid, nothing changed…";
  }
}
```

Dynamic Exception Specification notifies that the Function might **throw** an **int** at run time.

Function is called with bad argument and **throw**s.
Exception handled outside of **throwing** Function.

Exception(s) Implementation

Throwing (`throw` expression) Semantics

Post C++11 Standard – Function `throw` Lists are deprecated

Why?

- Run-Time Checking: C++ Exception Specifications are checked at runtime rather than at compile time (*Why?*), so they offer no programmer guarantees that all exceptions have been handled. The run-time failure mode (calling `std::unexpected()`) does not lend itself to recovery.
- Run-Time Overhead: Run-time checking requires the compiler to produce additional code, impacts optimizations.
- Unusable in Generic Code: In generic code, not generally possible to know what types of Exceptions may be thrown from within operations on one template argument or another: Precise exception specification cannot be written.

Exception(s) Implementation

Throwing (`throw` expression) Semantics

Post C++11 Standard – Function `throw` Lists are deprecated

In practice two forms of Exception-Throwing guarantees are useful:

- An operation might throw an (any) Exception

```
_ret_type_ myFunc(_params_list_ );
```

Simply omit Exception Specification.

- An operation promises to never throw an (any) Exception

```
_ret_type_ myFunc(_params_list_ ) noexcept ;
```

`noexcept` Specifier introduced in C++11: Unlike empty `throw()`, it does not require the compiler to introduce code to check whether an exception is thrown (If a `noexcept` Function is exited via Exception, `std::terminate()` is called).

Exception(s) Implementation

Exception–Handling and Nested Functions:

By-Example:

```
int main(){
  try{
    func_level_1();
  }
  catch(const int & ex){
      cerr << ex;
  }
}
```

```
void func_level_1() {
  AClass aC;
  func_level_2();
}
```

```
void func_level_2() {
  BClass bc;
  throw (-1);
}
```

Output:

```
A ctor → B ctor
```

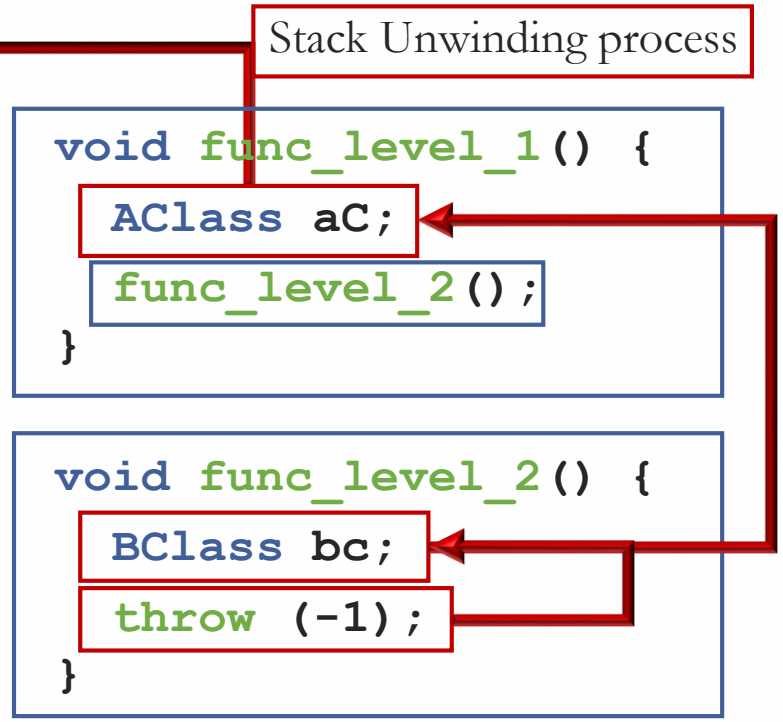
```
class AClass{
 public:
 AClass()
 { cout << "A ctor"; }
 ~AClass()
 { cout << "A dtor"; }
};

class BClass{
 public:
 BClass()
 { cout << "B ctor"; }
 ~BClass()
 { cout << "B dtor"; }
};
```

Exception(s) Implementation

Exception–Handling and Nested Functions:

By-Example:

```
int main(){
  try{
    func_level_1();
  }
  catch(const int & ex){
      cerr << ex;
  }
}
```




```
class AClass{
 public:
 AClass()
 { cout << "A ctor"; }
 ~AClass()
 { cout << "A dtor"; }
};

class BClass{
 public:
 BClass()
 { cout << "B ctor"; }
 ~BClass()
 { cout << "B dtor"; }
};
```

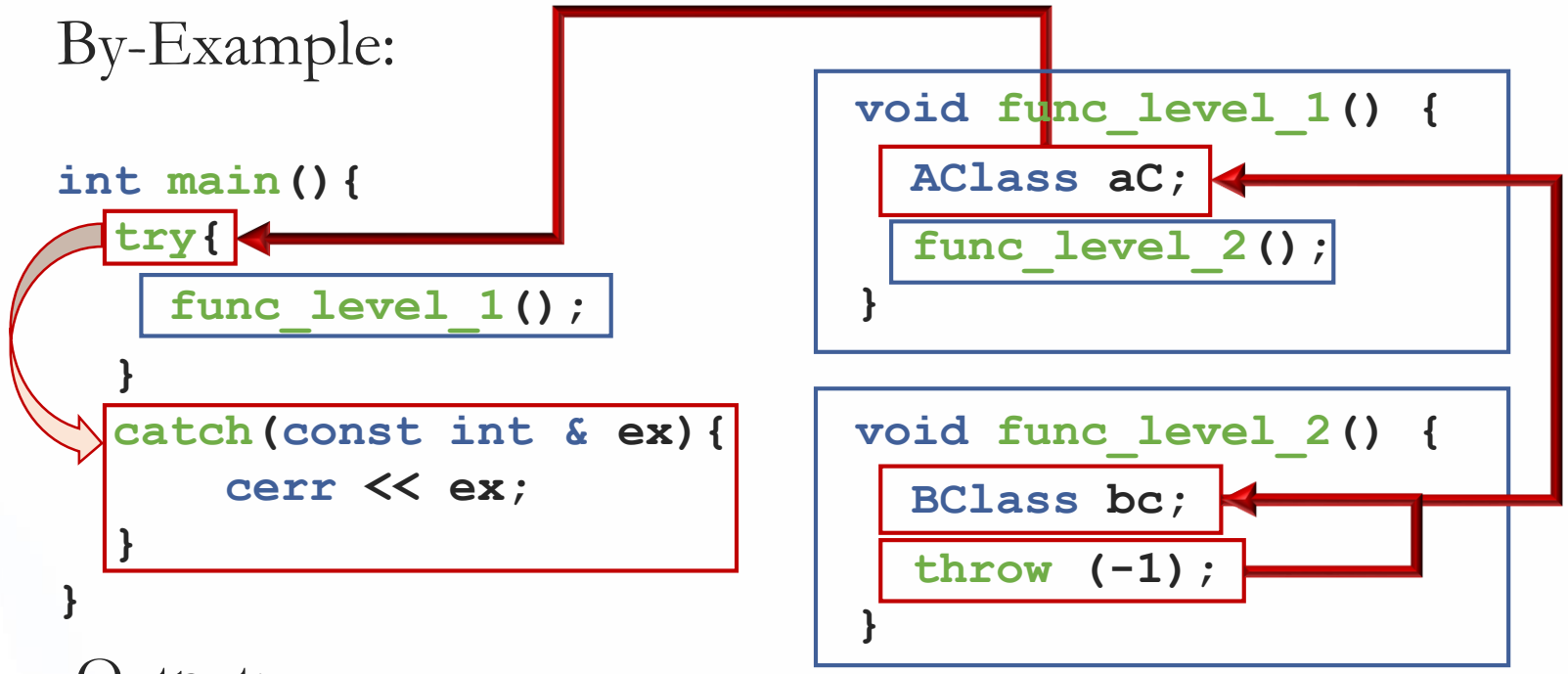
Output:

```
A ctor → B ctor → B dtor → A dtor
```

Exception(s) Implementation

Exception–Handling and Nested Functions:

By-Example:




```
class AClass{
 public:
 AClass()
 { cout << "A ctor"; }
 ~AClass()
 { cout << "A dtor"; }
};

class BClass{
 public:
 BClass()
 { cout << "B ctor"; }
 ~BClass()
 { cout << "B dtor"; }
};
```

Output:

A ctor → B ctor → B dtor → A dtor → -1

Exception(s) Implementation

Exception Class(es)

The `try` – `throw` – `catch` Paradigm works with Exception Objects

- A custom Exception Class Object can be created, thrown, and caught.
  (`std::exception`, `std::runtime_error`, `std::ios_base::failure` are Classes)

Exception Class Hierarchies using Inheritance can also be created.

- A `catch` block expecting a Base/Parent Class type will also catch a Derived/Child Class Object.

Exception(s) Implementation

Exception Class(es)

General Approach:

- Exception Class name should be descriptive, reflecting the error.
- Exception Class should contain error-relevant (important) information
  a) a parameter value, b) the name of Function that detected the error,
  c) an error description

Exception Class is required to have:

a) Constructor(s) – and Copy-Constructor available (can be the default).
b) Accessor(s) – for member access.

Exception(s) Implementation

Exception Class(es) – By-Example:

```
class DivByZeroExcept : public MathExcept {
public:
 DivByZeroExcept(double numerator,
      int code, string description)
    : MathExcept(code, description){
   m_numerator = numerator;
 }
 DivByZeroExcept(const DivByZeroExcept & o)
    : MathExcept(o){
   m_numerator=o.m_numerator;
 }
 double GetNumerator() const
 { return m_numerator; }
private:
 double m_numerator;
};
```

```
class MathExcept {
public:
 MathExcept(int code,
            string description){
  m_code=code;
  m_description=description;
 }
 MathExcept(const MathExcept & o){
  m_code=o.m_code;
  m_description=o.m_description;
 }
 int GetCode() const
 { return m_code; }
 string GetDescription() const
 { return m_description; }
private:
 int m_code;
 string m_description;
};
```

Exception(s) Implementation

Exception Class(es) – By-Example:

```
class DivByZeroExcept : public MathExcept {
public:
 DivByZeroExcept(double numerator,
       int code, string description)
    : MathExcept(code, description){
   m_numerator = numerator;
 }
 DivByZeroExcept(const DivByZeroExcept & o)
    : MathExcept(o){
   m_numerator=o.m_numerator;
 }
 double GetNumerator() const
 { return m_numerator; }
private:
 double m_numerator;
};
```

```
class MathExcept {
public:
 MathExcept(int code,
          string description){ … }
 MathExcept(const MathExcept & o){ … }
…
};
```

Note:
The only way to call Base Class `ctor` from Derived Class `ctor` is via Initialization List(s).

Exception(s) Implementation

Exception Class(es) – By-Example:

```
try{
   …
}
catch(const DivByZeroExcept & ed){
  …
}

try{
   …
}
catch(const MathExcept & em){
  …
}
```

```
class MathExcept {
public:
 MathExcept(int code,
            string description){ … }
 MathExcept(const MathExcept & o){ … }
…
};
```

```
class DivByZeroExcept : public MathExcept {
public:
 DivByZeroExcept(double numerator,
       int code, string description)
    : MathExcept(code, description){ … }
 DivByZeroExcept(const DivByZeroExcept & o)
    : MathExcept(o){ … }
 …
};
```

Exception(s) Implementation

Exception(s) in Class `ctor`(s)

Good idea to perform Handling of Failed Constructor(s):

- If something in the Constructor fails, the Object is initialized in a "Zombie State" (without Exception `throw`ing, Class has to have a flag-member to indicate validity).
- Exception `throw`ing *Unwinds the Stack* and destroys any internally created sub-Objects.

Note:
The Destructor of the Class that throws the Exception is not automatically called (only Stack Unwinding is performed).

Exception(s) Implementation

Exception(s) in Class `ctor`(s)

Good idea to perform Handling of Failed Constructor(s):

- Removes "Zombie State" Objects.

```
MyClass::MyClass ( int value ) {
 m_intVal_Pt = new int(value);
 …
 /* detection of failure*/
 if ( bad_state_detected() )
  throw ConstructorFailure( );
 …
}
```

```
int main(){
  MyClass * mc_Pt = NULL;
  try{
    mc_Pt = new MyClass(1000);
  }
  catch(const ConstructorFailure &){
    delete mc_Pt; mc_Pt = NULL;
  }
}
```

Class `dtor`-calling is handled in catch Block, while internally created sub-Objects are deleted via *Stack Unwinding*.

Exception(s) Implementation

Exception(s) in Class `dtor`(s)

Bad idea to perform Handling of Failed Destructor(s):

- If your Object is being destroyed as part of Stack Unwinding triggered by another `thrown` Exception ?

- Generic Rule is to not throw Exceptions from Destructors.

CS-202

Time for Questions !