Week 8 - Exceptions Handling

Sahbi Ben Ismail
s.ben-ismail@imperial.ac.uk

EE2-12 – Software Engineering 2
Object Oriented Software Engineering

Imperial College
London

Module Syllabus

Week 1	Classes and Objects I: Introduction
Week 2	Classes and Objects II: Constructors, and Operator Overloading
Week 3	More on Classes, Objects, and Operator Overloading
Week 4	Objects and Dynamic Memory
Week 5	Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)
Week 6	Polymorphism and Virtual Functions
Week 7	Generic Programming: Templates, and the Standard Template Library (STL)
Week 8	**Exceptions Handling**
Week 9	C++ to Java
Week 10	Revision

Week 8 Intended Learning Outcomes (ILOs)

By the end of this week you should be better able to:

Lecture

1. Design and write **exception-safe** code
2. Understand how to **throw exceptions** and how to **catch** them
3. Understand **exceptions hierarchy** and the **order** of catching them
4. Understand the **RAII** idiom in C++ to acquire and release resources

Lab

1. Apply exceptions **throwing** and **catching** in an exception-safe code
2. [Catch up, Revision of previous labs]

Problematic: ‘normal’ vs ‘exceptional’ cases

- Dealing with exceptional (unexpected) troublesome occurrences.
- Interrupting without disrupting.

Outline

Outline

operator[] and range check

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      vector<int> v(3);
8      cout << v.size() << " " << v.capacity() << endl;
9      cout << v[33] << endl;
10     return 0;
11 }
```

```
3 3
0
```

- Or another number.
- Or segmentation fault...

Range check in `vector`

- The `vector::operator[]`, like the array subscript *does not* perform range check.
- Although a ‘coherent’ choice, we would expect more (at least to be able to choose).
- Are there any alternatives?

operator[] vs `vector::at()`

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(){
7      vector<int> v(3);
8      cout << v.size() << " " << v.capacity() << endl;
9      cout << v.at(33) << endl;
10     return 0;
11 }
```

```
3 3
terminate called after throwing an instance of↩
    'std::out_of_range'
  what():  vector::_M_range_check
Aborted
```

`vector::at()`

- Function `vector::at()` is *almost* equivalent to `operator[]`.
- It works also with assignments:
  `v.at(2) = 10;`
- It does perform range check.
- If check fails it *throws*. . . an *exception*.

Programs in a dynamic environment

- Programming is not just about algorithm correctness and performance.
- Most programs work *interactively*.
- Interaction with users, hardware, communication layers. . .
- Many things can go unexpectedly.
- Programs need to be robust and deal with it.

Usb drives

- Right after we open a file we perform a check:

```
if(fin.is_open()){
    // operations on the file
}
else{
    // print error messages, do something else
}
```

- Imagine the program is moving files from an usb drive to a computer.
- Imagine that *while* the files are being moved the drive is unplugged (so long for the initial check).
- Are we ok with the program just crashing (or terminating in an uncontrolled way)?

Usb drives

- The sudden invalidation of a file stream should never occurr and is not the programmer's fault.
- However it might still happen (and considering the possibility is part of the design of such a program).
- We want to handle the exception limiting the damage:
- Keeping files in a consistent state.
- Flushing buffers.
- We want a language feature which encourages exception handling:
- Without the need to write an `if` after every other instruction.
- Keeping the program flow as regular as possible.

Example: `sqrt`

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main(){
6
7      double n, result;
8      cin >> n;
9      result = sqrt(n);
10     cout << result << endl;
11
12     return 0;
13 }
```

```
-1
-nan
```

- What if we don't just print `result` but it is an argument, e.g. for `very_delicate_function_controlling_aircraft()`?

Example: sqrt

```
1  #include <iostream>
2  #include <cmath>
3  #include <cstdlib>
4  using namespace std;
5
6  int main(){
7      double n, result;
8      cin >> n;
9      if(n<0){
10         cout << "cannot compute square root of negative number" <<↩
               endl;
11         exit(EXIT_FAILURE);
12     }
13     result = sqrt(n);
14     if(isnan(result)){
15         cout << "cannot compute square root" << endl;
16         exit(EXIT_FAILURE);
17     }
18     cout << result << endl;
19     return 0;
20 }
```

Example: my_sqrt

```
1  double my_sqrt(double n){
2      double result;
3      if(n<0){
4          cout << "cannot compute square root of negative number" <<↩
                endl;
5          exit(EXIT_FAILURE);
6      }
7      result = sqrt(n);
8      if(isnan(result)){
9          cout << "cannot compute square root" << endl;
10         exit(EXIT_FAILURE);
11     }
12     return result;
13 }
```

- Factorizing checks in a new function.

Example: `my_sqrt`

```
1  int main(){
2      double n;
3      cin >> n;
4      cout << my_sqrt(n) << endl;
5      cout << "now I'd like to do other things unrelated with sqrt" ↩
           << endl;
6
7      return 0;
8  }
```

```
-1
cannot compute square root of negative number
```

- Calling `exit` terminates the program, not just the function.

Outline

Throwing exceptions

```
1  double my_sqrt(double n){
2      double result;
3      if(n<0){
4          throw string("cannot compute square root of negative ↩
               number");
5      }
6      result = sqrt(n);
7      if(isnan(result)){
8          throw string("cannot compute square root");
9      }
10     return result;
11 }
```

Exception thrown

```
1  int main(){
2      double n;
3      cin >> n;
4      cout << my_sqrt(n) << endl;
5      cout << "now I'd like to do other things unrelated with sqrt"
           << endl;
6
7      return 0;
8  }
```

```
terminate called after throwing an instance of
    'std::string'
Aborted
```

- Exception reached 'top level' without being handled.

Catching exceptions

```
1  int main(){
2      double n;
3      cin >> n;
4      try{
5          cout << my_sqrt(n) << endl;
6          cout << "hope you enjoyed your square root" << endl;
7      }
8      catch(const string& msg){
9          cout << msg << endl;
10     }
11     cout << "now I'd like to do other things unrelated with sqrt" ↩
           << endl;
12
13     return 0;
14 }
```

```
-1
cannot compute square root of negative number
now I'd like to do other things unrelated with↩
    sqrt
```

Exceptions and program flow

- When a function `throw`s an exception:
  - No other subsequent instructions in the function are executed.
  - Control goes immediately back to the caller.
- If in the caller the function call is in a `try` block:
  - No other subsequent instructions in the `try` block are executed.
  - Control goes to the `catch` block.
- Otherwise control goes up one more level.
  - Until a `try` block is encountered, or the top level is reached (which means the exception was not handled!).

Catching the right exceptions

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main(){
6      vector<int> v(3);
7      try{
8          cout << v.at(33) << endl;
9          cout << "the index was ok!" << endl;
10     }
11     catch(const string& a){
12         cout << "be careful with your bounds!" << endl;
13     }
14     return 0;
15 }
```

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

Catching the right exceptions

```
1  #include <iostream>
2  #include <vector>
3  #include <stdexcept>
4  using namespace std;
5
6  int main(){
7      vector<int> v(3);
8      try{
9          cout << v.at(33) << endl;
10     }
11     catch(const out_of_range& a){
12         cout << "be careful with your bounds!" << endl;
13     }
14     return 0;
15 }
```

```
be careful with your bounds!
```

What can we `throw` and `catch` exactly?

- After `throw` there can be any variable or value of any type (primitive or object).
- The `catch` block needs to intercept the right type, otherwise the exception is forwarded to the upper level.
- Throwing different types of exception for different kinds of occurrences (in order to perform the right handling in a specific `catch`).

Catching various kinds of exceptions

```
1   int main(){
2       vector<int> v(3);
3       int i;
4       double num, result;
5       cin >> i;
6       cin >> num;
7       try{
8           result = my_sqrt(num);
9           v.at(i) = result;
10          cout << v.at(i) << endl;
11      }
12      catch(const out_of_range& e){
13          cout << "index not suitable: " << e.what() << endl;
14      }
15      catch(const string& e){
16          cout << "unsuccessful sqrt computation: " << e << endl;
17      }
18      cout << "something else" << endl;
19      return 0;
20  }
```

Catching various kinds of exceptions

```
3
0
index not suitable: vector::_M_range_check
something else
```

```
2
-1
unsuccessful sqrt computation: cannot compute ↩
    square root of negative number
something else
```

```
3
-1
unsuccessful sqrt computation: cannot compute ↩
    square root of negative number
something else
```

Catching exceptions and upcasting

```
1   int main(){
2       vector<int> v(3);
3       int i;
4       double num, result;
5       cin >> i;
6       cin >> num;
7       try{
8           result = my_sqrt(num);
9           v.at(i) = result;
10          cout << v.at(i) << endl;
11      }
12      catch(const logic_error& e){
13          cout << "index not suitable: " << e.what() << endl;
14      }
15      catch(const string& e){
16          cout << "unsuccessful sqrt computation: " << e << endl;
17      }
18      cout << "something else" << endl;
19      return 0;
20  }
```

- `class out_of_range : public logic_error`

Defining our own exceptions

```
1  class invalid_sqrt_argument : public invalid_argument {
2      public:
3          invalid_sqrt_argument(const string& what) :
4              invalid_argument(what) {}
5  };
```

```
1  double my_sqrt(double n){
2      double result;
3      if(n<0){
4          throw invalid_sqrt_argument("cannot compute square root of↩
               negative number");
5      }
6      result = sqrt(n);
7      if(isnan(result)){
8          throw logic_error("cannot compute square root");
9      }
10     return result;
11 }
```

- `class invalid_argument : public logic_error`

Does order matter?

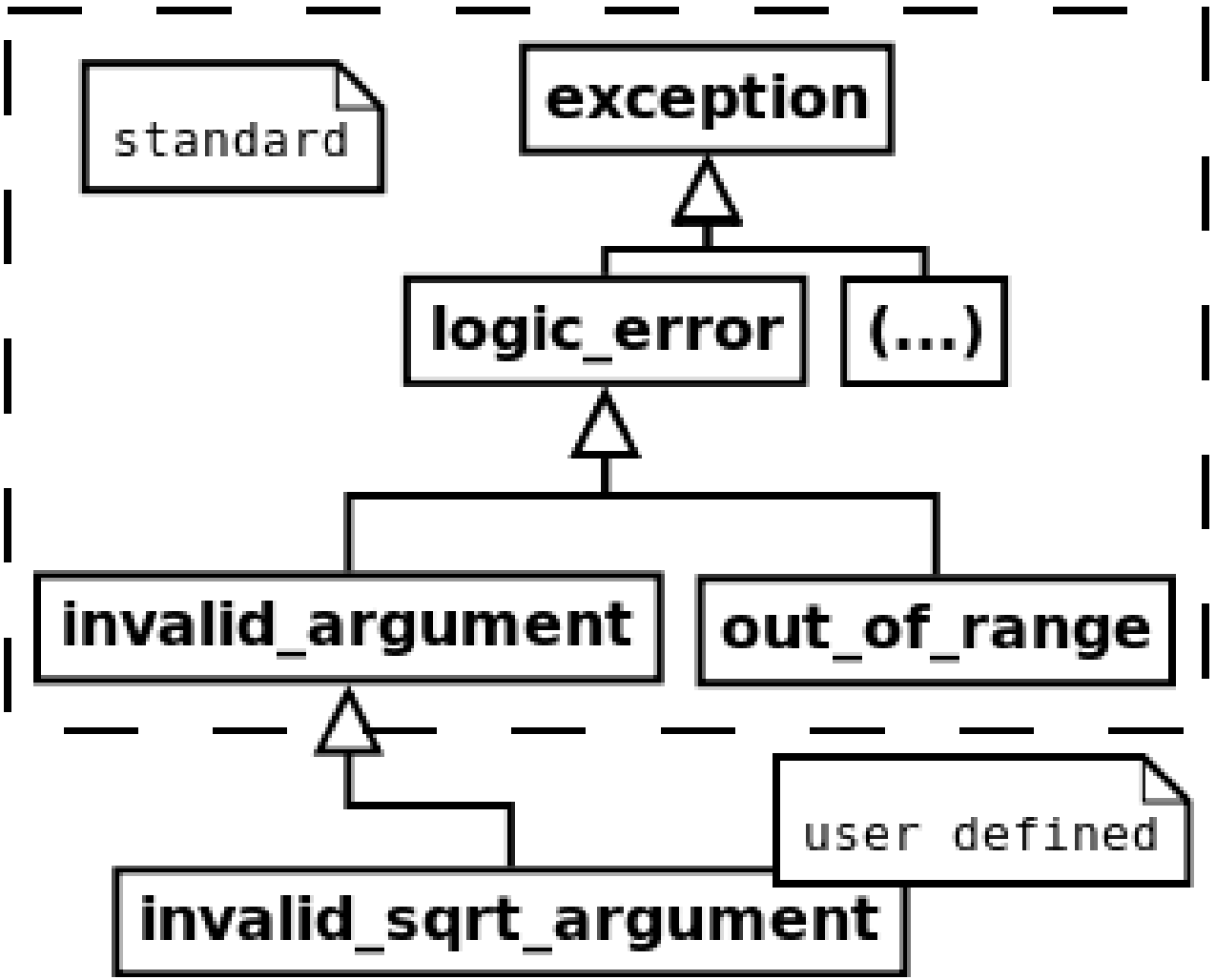
```
1   int main(){
2       vector<int> v(3);
3       int i;
4       double num, result;
5       cin >> i;
6       cin >> num;
7       try{
8           result = my_sqrt(num);
9           v.at(i) = result;
10          cout << v.at(i) << endl;
11      }
12      catch(const logic_error& e){
13          cout << "some other logic error: " << e.what() << endl;
14      }
15      catch(const invalid_sqrt_argument& e){
16          cout << "unsuccessful sqrt computation: " << e.what() << ↩
                endl;
17      }
18      catch(const out_of_range& e){
19          cout << "index not suitable: " << e.what() << endl;
20      }
21      cout << "something else" << endl;
22      return 0;
23  }
```

Order matters

```
In function 'int main()':
41: warning: exception of type '↩
    invalid_sqrt_argument' will be caught
38: warning:    by earlier handler for 'std::↩
    logic_error'
```

```
3
0
some other logic error: vector::_M_range_check
something else
```

Exception hierarchy



Catching exceptions in the right order

```
catch(const invalid_sqrt_argument& e){
    cout << "unsuccessful sqrt computation: " << e.what() << ↩
        endl;
}
catch(const out_of_range& e){
    cout << "index not suitable: " << e.what() << endl;
}
catch(const logic_error& e){
    cout << "some other logic error: " << e.what() << endl;
}
catch(const exception& e){
    cout << "some other exception" << endl;
}
```

- From most specific to most generic (of those we can actually handle).

Exceptions example: database lock

- We want to write a database updating script.
- Multiple scripts might run at the same time on the same database.
- We need a way to prevent scripts overwriting each other.
- Before updating the database each script has to put a ‘lock’ on the resource, when the update is complete the lock is removed (if there is a lock already, no update takes place).
- The lock can just be represented by a file with a conventional name (created to put a lock, deleted to remove it).

Update function

```
1  bool update_db(int data) {
2      bool exit_code;
3      ifstream infile(".lockdb");
4      if(infile.is_open()){
5          cout << "resource is busy" << endl;
6          infile.close();
7          exit_code = false;
8      }
9      else{
10         ofstream ofile(".lockdb");
11         if(!ofile.is_open()){
12             throw runtime_error("couldn't lock resource");
13         }
14         cout << "resource is locked" << endl;
15         ofile.close();
16         cout << "doing some operations" << endl;
17         write_db(data);
18         remove(".lockdb");
19         cout << "resource unlocked" << endl;
20         exit_code = true;
21     }
22     return exit_code;
23 }
```

Testing

```
1  void write_db(int){
2      // some operations
3  }
```

```
1  int main(){
2      try{
3          update_db(10);
4      }
5      catch(exception e){
6         cout << "exception!" << endl;
7      }
8      return 0;
9  }
```

```
resource is locked
doing some operations
resource unlocked
$ ls .lockdb
No such file or directory
```

What happens with exceptions?

```
1  void write_db(int){
2      throw exception();
3  }
```

```
resource is locked
doing some operations
exception!
$ ls .lockdb
.lockdb
```

Locked in

- An exception was thrown while working on the file.
- The instructions which would normally release the lock weren't executed!
- No other process now can access the resource or release the lock.

Exception safe code

- Concept of *exception safe* code: keeping state consistent and handling resources suitably even if an exception is thrown.
- Our code is not exception safe because a resource is not released.
- It's necessary to write code having in mind that exception could interrupt the flow.
- But how can the resource be released if the flow is interrupted and no other instructions can be executed?

RAII

- Before the handling of an exception (in the `catch` block), the destructors of objects allocated on the stack are called.
- Resource Acquisition is Initialization (RAII) idiom (invented by Stroustrup):
- The resource is acquired by the constructor of an associated object.
- The resource release is in the object destructor.

Bjarne Stroustrup - GoingNative 2013 - The Essence of C++. t=22'08" https://youtu.be/D5MEsboj9Fc?t=1328

class Lockdb

```
1  class Lockdb{
2      public:
3          Lockdb();
4          ~Lockdb();
5          bool get_status();
6      private:
7          bool status;
8  };
```

Lockdb constructor

```
1  Lockdb::Lockdb(){
2      ifstream infile(".lockdb");
3      if(infile.is_open()){
4          status = false;
5          cout << "resource is busy" << endl;
6      }
7      else{
8          ofstream ofile(".lockdb");
9          if(!ofile.is_open()){
10             status = false;
11             throw runtime_error("couldn't lock resource");
12         }
13         status = true;
14         cout << "resource is locked" << endl;
15     }
16 }
```

Lockdb destructor and get_status()

```
1   Lockdb::~Lockdb(){
2       cout << "destructor" << endl;
3       if(status){
4           remove(".lockdb");
5           cout << "resource unlocked" << endl;
6       }
7   }
8
9   bool Lockdb::get_status(){
10      return status;
11  }
```

Update function

```
1  bool update_db(int data){
2      bool exit_code;
3      Lockdb ldb;
4      if(ldb.get_status()){
5          cout << "doing some operations" << endl;
6          write_db(data);
7          exit_code = true;
8      }
9      else{
10         exit_code = false;
11     }
12     return exit_code;
13 }
```

Testing

```
1  void write_db(int){
2      throw exception();
3  }
```

```
1  int main(){
2      try{
3          update_db(10);
4      }
5      catch(exception e){
6         cout << "exception!" << endl;
7      }
8      return 0;
9  }
```

```
doing some operations
destructor
resource unlocked
exception!
$ ls .lockdb
No such file or directory
```

Outline

Wrap-up

- “The practical difficulty in following these principles is that innocent-looking operations (such as <, =, and sort()) might throw exceptions. Knowing what to look for in an application takes experience.”
  [B. Stroustrup, Exception Safety: Concepts and Techniques]
- In some languages (e.g. Java) exception handling is checked by the compiler.
- (Exceptions weren’t in C++ from the beginning.)
- Try to write exception safe code.

What to do next?

Next Lab

- Catch up, revision

Next Lecture

Week 9 - C++ to Java