Github: LoreDN
Author: Lorenzo Di Napoli
Repository: https://github.com/LoreDN/Cpp

Exception_v1.1.0 — Documentation

Contents

Library Description

This library is a special one, because it is the foundation that allows all the other **LDN** libraries to work the intended way.
Here are collected all the custom **Exceptions** that can be triggered when working with the libraries, togheter with some test functions, used in order to catch the wanted **Exception**.

CONTENTS OF THE LIBRARY

This library is *header-only*, since the methods and functions implemented are really short.
In this way, the library can be used easily without the need to link external shared-library files *.so*.
The only precaution to take when compiling the library headers, is to set the flag **-std=c++20**, since it is needed in order to work properly.

The library is divided in multiple headers:

- **"IF_Exception.hpp":** here is defined the Interface `LDN::Exception`, which is the Polimorphic Abstract Class all the Exceptions implements.
- **"Exceptions.hpp":** the generic header, which includes all the Exceptions (it has been designed in order to be included alone).
- **"*Specification*.hpp":** the headers of the Specific Classes, each specification has its own one (such as the `LDN::exception::Index` Class which is defined in the header *"Index.hpp"*).

INHERITARY STRUCTURE

The library starts with the definition of an Interface, represented by a simple abstract class: *"Exception"*.
Starting from this class, all the derivated ones create a Tree Structure, which allows to manage in a simple way all the **Exceptions**, and makes easy including new ones or derivating more specific **Exceptions** from already existing ones.

Classes Documentation

In the *"Exception"* library have been implemented a total of five classes:

IF. **LDN::Exception:** the Root Interface, from which all the other Exceptions derive from.

1. **LDN::exception::Index:** handles out-of-memory limit accesses for a Data-Structure (as accessing to `array[i > size]`).
2. **LDN::exception::Size:** handles Data-Structure size limits (as pushing an element to an Heap with `heap.usage == heap.size`).
3. **LDN::exception::File:** handles filestreams and checks if they are valid (works also with `FILE*`).
4. **LDN::exception::Dimension:** handles dimension compatibility for two Data Structures (as checking dimensions for a `Matrix * Vector` multiplication or for copying an array into another one).

IF. Exception

```
class Exception {

     protected:

          // attributes
          std::string message;

     public:

          // constructor / destructor
          explicit Exception(const std::string& msg) :  message(msg) {}
          virtual ~Exception() noexcept = default;

          // getter
          inline const std::string& getMessage() const noexcept { return message; }

          // Default Method -- Print Exception message
          virtual void print(std::ostream& exc_stream = std::cerr) const {
              exc_stream « "[Exception]:  " « message « std::endl;
          }

};
```

`std::string message;`

The Exception message, it is a *protected* member, since the user is not able to modify it.
This allows each **Exception** to manage its error message indipendently from the others.

`explicit Exception(const std::string& msg);`

This is the *constructor*, which assignes a string `msg` (usually set when catching the **Exceptions**, in order to proper describe it) to the *protected* member `Exception.message`.

`virtual ~Exception() noexcept = default;`

This is the *destructor*, which is automatically invocated when deleting the object.
It has been left as a *virtual method*, in order to ensure the Class to be abstract.

`virtual void print(std::ostream& exc_stream = std::cerr) const;`

Core of the *"Exception"* Interface, this *virtual method* imposes that each **Exceptions** prints something in order to let the user know it was catched, (by default it prints the message).
It is already define as a default method, however since it has been left as a *virtual method*, each **Exception** is able to modify it with an Override; this is not the case in the already implemented Classes, given in the library.

1. Index

```
class Index :  public LDN::Exception {

     public:

          // constructor / destructor
          explicit Index(const std::string& msg) :  LDN::Exception("[Index Exception]
              -> " + msg) {}
          ~Index() noexcept override = default;

};

// Utility function to check index validity and throw Index exception if invalid
inline void throwIfIndexException(int index, size_t size) {
     if (index < 0 || static_cast<size_t>(index) >= size) {
          throw LDN::exceptions::Index("Index " + std::to_string(index) + " is out of
              bounds for size " + std::to_string(size) + "!!!");
     }
}
```

`explicit Index(const std::string& msg);`

This is the *constructor*, which assignes a string `msg` (set when catching the **Exception**, in order to proper describe it) to the *protected* member `Exception.message`, specifing that it is an `[Index Exception]`.

`~Index() noexcept override = default;`

This is the *destructor*, which is automatically invocated when deleting the object.

`inline void throwIfIndexException(int index, size_t size);`

Test function independent from the **Index** class, which allows to catch the **Exception**.
The test throws the **Exception** only if there is an attempt to access out-of-bounds memory; it is checked by a simple `if (index < 0 || index >= size)`.

2. Size

```
class Size :  public LDN::Exception {

     public:

          // constructor / destructor
          explicit Size(const std::string& msg) :  LDN::Exception("[Size Exception]
              -> " + msg) {}
          ~Size() noexcept override = default;

};

// Utility function to check Data-Structure usage and and throw Size Exception if full
inline void throwIfSizeException(size_t size, size_t limit) {
     if (size >= limit) {
          throw LDN::exceptions::Size("Size " + std::to_string(size) + " is out of
              bounds for the maximum limit " + std::to_string(limit) + "!!!");
     }
}
```

`explicit Size(const std::string& msg);`

This is the *constructor*, which assignes a string `msg` (set when catching the **Exception**, in order to proper describe it) to the *protected* member `Exception.message`, specifing that it is a `[Size Exception]`.

`~Size() noexcept override = default;`

This is the *destructor*, which is automatically invocated when deleting the object.

`inline void throwIfSizeException(size_t size, size_t limit);`

Test function independent from the **Size** class, which allows to catch the **Exception**.
The test throws the **Exception** only if there is an attempt to expand an already full Data-Structure; it is checked by a simple `if (size >= limit)`.

3. File

```
class File :  public LDN::Exception {

    public:

        // constructor / destructor
        explicit File(const std::string& msg) :  LDN::Exception("[File Exception]
            -> " + msg) {}
        ~File() noexcept override = default;

};

// File modes
enum class FileMode { Read, Write, IO };

// Concept to check if a type is a File stream
template <typename Stream>
concept FileStream = requires(Stream s) {{s.is_open()} -> std::convertible_to<bool>;};

// Utility function to check FileStream validity and throw File exception if invalid
inline void throwIfFileException(Stream& stream, const std::string& path, FileMode
mode) {
    if (!stream.is_open()) {
        std::string prefix;
        switch (mode) {
            case FileMode::Read:    prefix = "Input";   break;
            case FileMode::Write:    prefix = "Output";  break;
            case FileMode::IO:   prefix = "IO";   break;
        }
        throw LDN::exceptions::File(prefix + " file error:  unable to open '" + path
        + "'!!!");
    }
}

// Overload from FileStream to C style FILE*
inline void throwIfFileException(File* file, const std::string& path, FileMode mode) {
    if (!file) {
        std::string prefix;
        switch (mode) {
            case FileMode::Read:    prefix = "Input";   break;
            case FileMode::Write:    prefix = "Output";  break;
            case FileMode::IO:   prefix = "IO";   break;
        }
        throw LDN::exceptions::File(prefix + " file error:  unable to open '" + path
        + "'!!!");
    }
}
```

```
explicit File(const std::string& msg);
```

This is the *constructor*, which assignes a string `msg` (set when catching the **Exception**, in order to proper describe it) to the *protected* member `Exception.message`, specifing that it is a `[File Exception]`.

```
~File() noexcept override = default;
```

This is the *destructor*, which is automatically invocated when deleting the object.

```
enum class FileMode { Read, Write, IO };
```

Enumeration Class for the File Opening Mode (used for the **Exception message**).

```
template <typename Stream>
concept FileStream = requires(Stream s) { { s.is_open() } -> std::convertible_to<bool>;
};
```

Concept which ensure that a `Stream` has a method `is_open()`.

```
inline void throwIfFileException(Stream& stream, const std::string& path, FileMode mode);
```

Test function independent from the **File** class, which allows to catch the **Exception**.
The test throws the **Exception** only if there is an attempt to use a filestream which has not been open correctly; it is checked by a simple `if (!stream.is_open())`.

```
inline void throwIfFileException(File* file, const std::string& path, FileMode mode);
```

Overload of the `throwIfFileException()` function which allows to use `File*` as arguments instead of `Stream`.

4. Dimension

```
class Dimension :  public LDN::Exception {

     public:

         // constructor / destructor
         explicit Dimension(const std::string& msg) :  LDN::Exception("[Dimension
             Exception] -> " + msg) {}
         ~Dimension() noexcept override = default;

};

// Utility function to check index validity and throw Index exception if invalid
inline void throwIfDimensionException(const size_t& DIM1, const size_t& DIM2) {
     if (DIM1 != DIM2) {
         throw LDN::exceptions::Dimension("The two objects have not compatible
             dimensions " + std::to_string(DIM1) + " vs " + std::to_string(DIM2)
             + " !!!");
     }
}
```

`explicit Dimension(const std::string& msg);`

This is the *constructor*, which assignes a string `msg` (set when catching the **Exception**, in order to proper describe it) to the *protected* member `Exception.message`, specifing that it is a `[Dimension Exception]`.

`~Dimension() noexcept override = default;`

This is the *destructor*, which is automatically invocated when deleting the object.

`inline void throwIfDimensionException(const size_t& DIM1, const size_t& DIM2);`

Test function independent from the **Dimension** class, which allows to catch the **Exception**.
The test throws the **Exception** only if there is an attempt to compare two Data Structures with non compatible sizes; it is checked by a simple `if (DIM1 != DIM2)`.