Object Oriented Programming - Lecture 7

Diana Borza - diana.borza@ubbcluj.ro

April 2021

Content

- Lambda expressions
- Exceptions
- Streams

Lambda expressions I

- A **lambda expression** (**lambda** or **closure**) allows the definition of an anonymous function inside another function.
- The anonymous function is defined in the function where it is called.
- Very useful for some algorithms defined in the STL std::find_if, std::count_if, std::transform.
- Syntax:

```
[ captureClause ] ( parameters ) -> returnType
{ function body;}
[captureClause] (parameter list) {function body;}
```

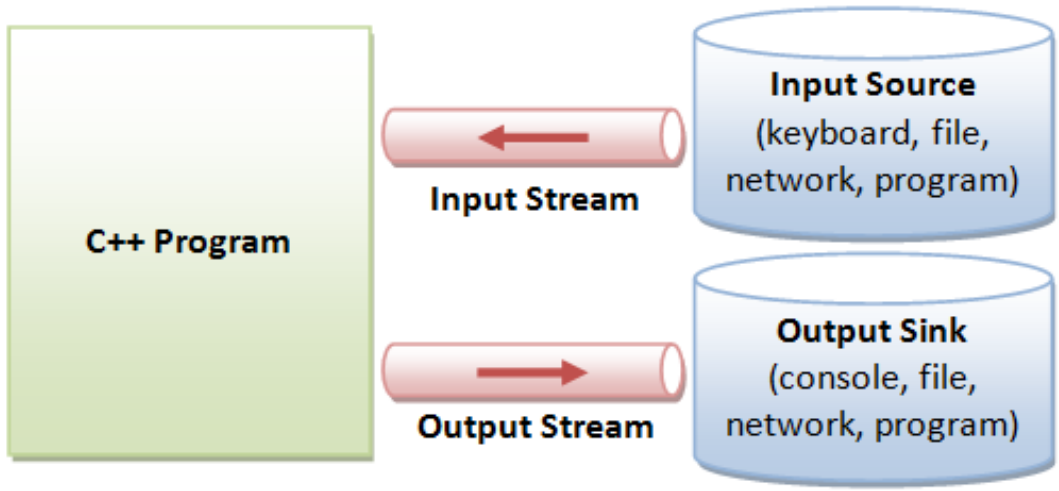
  - *captureClause* and *parameters* can be empty if not needed;
  - *returnType* is optional, and if omitted, auto will be assumed.

Lambda expressions II

- The **captureClause** clause is used to (indirectly) give a lambda access to variables available in the surrounding scope that it normally would not have access to.
- We need to list all the variables we want to access from within the lambda as part of the capture clause.
- For each variable in **captureClause**, **a clone!** of that variable is made (with an identical name) inside the lambda.
- A default capture (also called a capture-default) captures all variables that are mentioned in the lambda.
  - To capture all used variables by value, use a capture value of $=$.
  - To capture all used variables by reference, use a capture value of &.

Streams I

- **Stream**: a sequence of bytes flowing in or out of a program. The bytes are accessed *sequentially*.
- A stream is an abstraction for receiving/sending data in an input/output situation.



Internal Data Formats:

- Text: `char`, `wchar_t`
- `int`, `float`, `double`, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

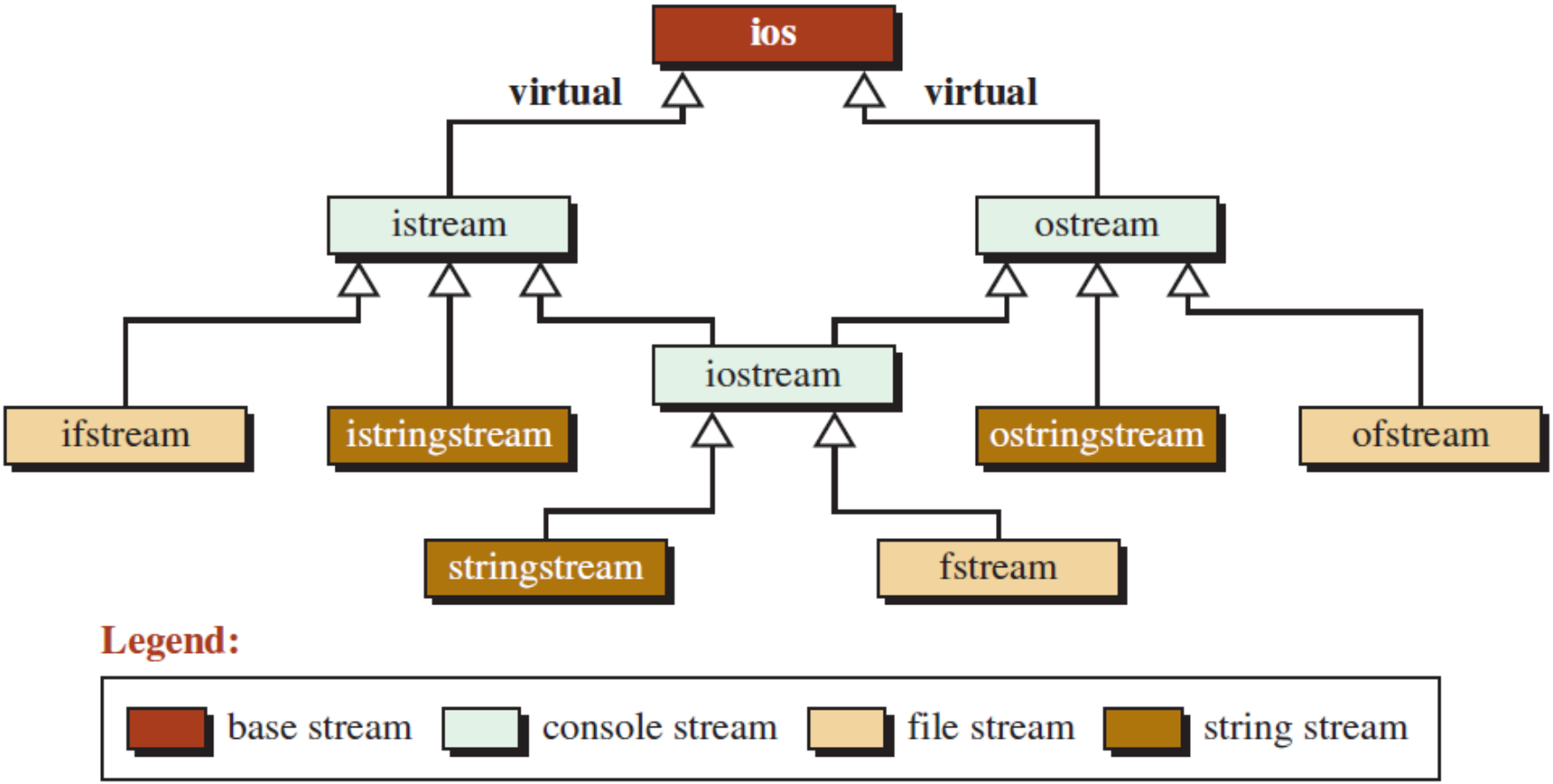
Streams II

- Streams act as an intermediate between programs and the actual IO devices (the programmers don't need to write code for handling the actual devices). We only need to know how to interact with the stream.
- *Input streams* are used to hold input from a data producer (or a source), such as a keyboard, a file, or a network.
- *Output streams* are used to hold output for a particular data consumer (or a sink), such as a monitor, a file, or a printer.
- Some devices, such as files and networks, are capable of being both input and output sources.

Streams III

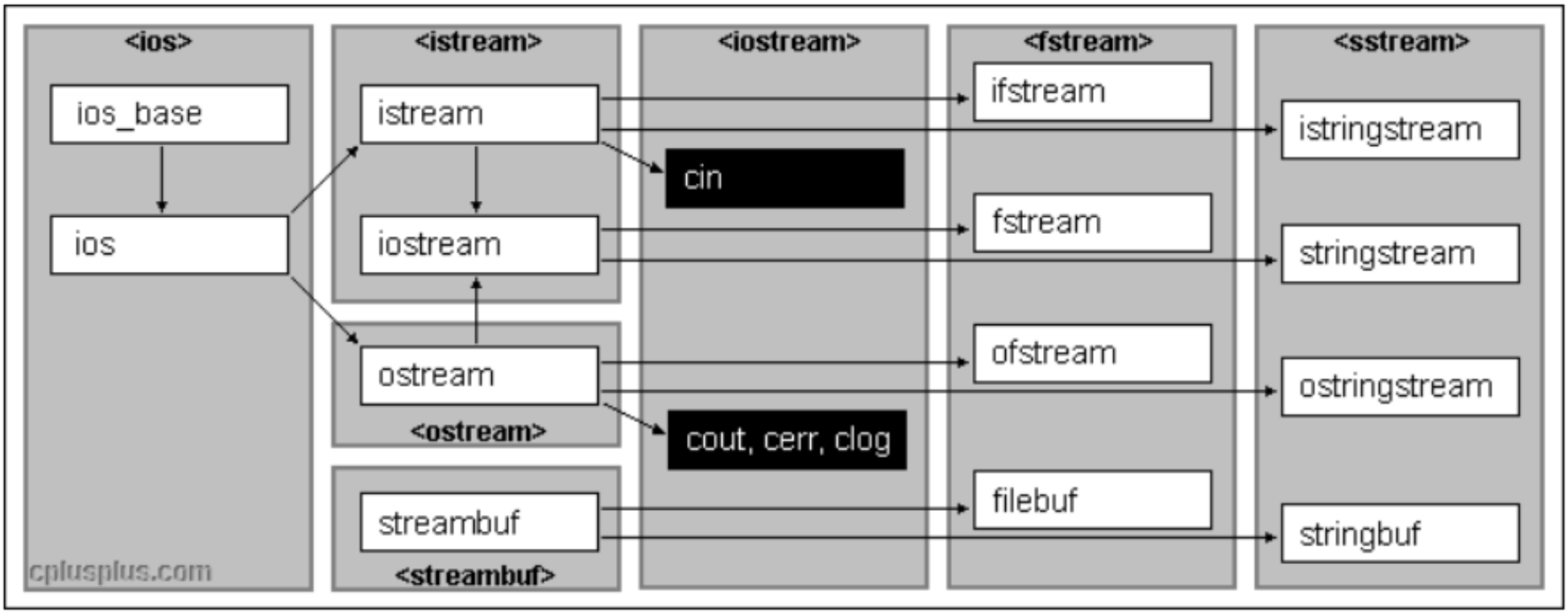
- **Streams are serial**:
  - data elements must be sent to or received from a stream one at a time or in a serial fashion.
  - random access (random reads/writes) are not possible; it's possible to seek a position in a stream and perform a read or write operation at that point.

Streams in C++


ios
virtual
virtual
istream
ostream
iostream
ifstream
istringstream
ostringstream
ofstream
stringstream
fstream
Legend:
base stream
console stream
file stream
string stream

Streams in C++

Input/Output library



Streams in C++

- The **ios** class defines the functionalities and variables that are common to both input and output streams.
- The **istream** class is the primary class used when dealing with input streams.
  - the **extraction operator (>>)** is used to remove values from the stream.
- The **ostream** class is the primary class used when dealing with output streams.
  - the **insertion operator (<<)** is used to put values into the stream
- The **iostream** class can handle both input and output, allowing bidirectional I/O.

Standard streams

- cin – object of an istream class tied to the standard input (typically the keyboard);
- cout – object of an ostream class tied to the standard output (typically the monitor);
- cerr – object of an ostream class tied to the standard error (typically the monitor), providing unbuffered output;
- clog – object of an ostream class tied to the standard error (typically the monitor), providing buffered output.

Unbuffered output is typically handled immediately, whereas buffered output is typically stored and written out as a block.

Output: Insertion operator <<

- **Insertion operator** << is used for writing operations on a stream (the standard output, in a file or in a memory zone).
- the operand from the left hand side of the operator<< must be an object of class ostream (or of a derived class).
- the operand from the right hand side can be an expression.
- the << operator is overloaded for standard types, and for custom types the programmer must overload it.
- since the << operator returns a reference to the current class, it may be chained and the function calls are made from left to right.

Input: Extraction operator >>

- **The extraction operator** >> is used for reading operations (from the keyboard, from a file, from the network).
- the operand from the left hand side of the operator >> must be an object of class istream (or of a derived class).
- the operand from the right hand side should be a variable.
- the >> operator is overloaded for standard types, and for custom types the programmer must overload it.
- since the >> operator returns a reference to the current class, it may be chained and the function calls are made from left to right.

Error flags

- Error flags indicate the internal state of a stream.
- They are automatically set by some input/output functions of the stream, to signal certain errors.

Flag	Meaning
goodbit	Everything is okay
badbit	Some kind of fatal error occurred (e.g. the program tried to read past the end of a file)
eofbit	The stream has reached the end of a file
failbit	A non-fatal error occurred (eg. the user entered letters when the program was expecting an integer)

Stream states

- To check an error flag, the following methods can be used:

Member function	Meaning
good()	Returns true if the goodbit is set (the stream is ok)
bad()	Returns true if the badbit is set (a fatal error occurred)
eof()	Returns true if the eofbit is set (the stream is at the end of a file)
fail()	Returns true if the failbit is set (a non-fatal error occurred)
clear()	Clears all flags and restores the stream to the goodbit state
clear(state)	Clears all flags and sets the state flag passed in
rdstate()	Returns the currently set flags
setstate(state)	Sets the state flag passed in

Manipulators I

- Functions specifically designed to be used in conjunction with the insertion and extraction operators on stream objects.
- Manipulators are placed in a stream and affect the way elements are displayed or read.
- Are used to change formatting parameters on streams and
- Are defined in the header iomanip.

Manipulators II

- http://www.cplusplus.com/reference/library/manipulators/
- boolalpha - Alphanumerical bool values
- showbase - Show numerical base prefixes
- showpoint - Show decimal point
- showpos - Show positive signs
- dec - Use decimal base
- hex Use hexadecimal base
- oct - Use octal base
- internal - Adjust field by inserting characters at an internal position
- left - Adjust output to the left
- right - Adjust output to the right
- endl - Insert newline and flush
- ends - Insert null character
- flush - Flush stream buffer

Input validation

- from header <cctype>

Function	Meaning
isalnum(int)	Returns non-zero if the parameter is a letter or a digit
isalpha(int)	Returns non-zero if the parameter is a letter
iscntrl(int)	Returns non-zero if the parameter is a control character
isdigit(int)	Returns non-zero if the parameter is a digit
isgraph(int)	Returns non-zero if the parameter is printable character that is not whitespace
isprint(int)	Returns non-zero if the parameter is printable character (including whitespace)
ispunct(int)	Returns non-zero if the parameter is neither alphanumeric nor whitespace
isspace(int)	Returns non-zero if the parameter is whitespace
isxdigit(int)	Returns non-zero if the parameter is a hexadecimal digit (0-9, a-f, A-F)

File I/O

- Files are data structures that are stored on a disk device.
- To work with files one must connect a stream to the file on disk.
- Any input or output operation performed on the stream will be applied to the physical file associated with it.
- There are 3 basic file I/O classes in C++:
  - fstream (derived from iostream);
  - ifstream (derived from istream) $\rightarrow$ input operations;
  - ofstream (derived from ostream) $\rightarrow$ output operations;
- To use a file in C++ you have to follow the steps:
  - open a file for reading and/or writing (simply instantiate an object of the appropriate file I/O class);
  - use the insertion (<<) or extraction (>>) operator to write to or read data from the file; when finished, close the file.

Opening a file I

- Opening a file means that you are associating a file stream with a file on the disk;
- The constructor of the classes ifstream and ofstream will open the file, if the filename is passed as an argument.
- Alternatively, a file can be opened with the fstream member function open.
- To check that a file is open you can use the function method is_open() (returns a bool).

Opening a file II

- *mode* is an optional parameter with a combination of the following flags:
  - ios::in - open for input operations.
  - ios::out - open for output operations.
  - ios::binary - open in binary mode.
  - ios::ate - set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
  - ios::app - all output operations are performed at the end of the file, appending the content to the current content of the file.
  - ios::trunc - if the file is opened for output operations and if already existed, its previous content is deleted and replaced by the new one.
- These flags can be combined using the bitwise operator OR (|).

Reading from a file

- Simply use the extraction operator to read data from a file.
- The ifstream's flag EOF (end of file) gets set only after a failed attempt to read past the end of the file.
- In case of corrupted data, the stream might fail to read them and EOF will never be reached (**infinite loop**)!
  - Solution: if a read operation fails, the ifstream's operator bool() will convert the ifstream object to false, if any errors occur during the read operation; $\rightarrow$ **read while the boolean value of the stream is true.**

Writing to a file

- Use the insertion operator.
- Output in C++ may be buffered (the elements from the stream might not be written immediately to the file).
- A **buffer** is a memory block that acts as an intermediary between the stream and the physical source or destination.
  - Each iostream object contains a pointer to a streambuf.
- When a buffer is written to disk, this is called *flushing* the buffer.
- *Flushing* occurs when:
  - the close() function is called;
  - ostream::flush() function is called;
  - std::endl manipulator also flushes the buffer.

Closing a file

- The stream's member function close() flushes the buffer and closes the file.
- Once a file is closed, it is available again to be opened by other processes.
- The close command will automatically be called when the stream object associated with an open file is destroyed.

Exceptions I

- **Exceptions** provide a mechanism to decouple handling of errors or other exceptional circumstances from the typical control flow of your code.
- When writing reusable code, error handling is a necessity.
- Exceptions allow programmers the freedom to handle errors when and how ever is most useful for a given situation.

Exceptions II

- Error handling without exceptions:
  - Error flags (global variables).
  - Return codes (error codes).
- Problems:
  - By default, the error flags are ignored (unless you write the code to verify the error flag or the error code).
  - Handling code is tedious to write and sometimes hard to understand;
  - What happens if the caller is not able to equipped to handle the error? Or if the caller just ignores the error?
  - What happens if the abnormal situation occurs in a constructor?
  - Error handling code is intricately linked to the normal control flow of the code.

Exceptions in C++ I

- In C++ we use 3 keywords, which operate in conjunction to each other:
  - **throw** - signals that an exception or error case has occurred; used to *raise* an exception.
  - **try** - marks an instruction block that might cause problems (i.e. throw exceptions). Acts like an *observer*, looking for the errors.
  - **catch** - immediately follows the try block and contains the code to handle the error.
    - print an error;
    - return a value or error code back to the caller;
    - may throw another exception.

Exceptions in C++ II

```
void exceptionExample(){

    try {
        // code that may throw an exception
    } catch (ExceptionClass &e) {
        // error handling , if the thrown error is of type ErrorClass ,
        // or any other type derived from ErrorClass
    } catch(...){
        // error handling - any type of error
    }
}
```

Exceptions - execution flow

- If no exception is thrown during execution of the try guarded section, the catch clauses that follow the try block are not executed.
- When an exception occurs, control moves from the throw statement *to the first* catch statement that can handle the thrown type: **stack unwinding**.
- This search continues "down" the function-call stack. If an exception is never caught, the program halts.
- After an exception has been handled the program, execution resumes after the try-catch block, not after the throw statement.
- Exceptions may be re-thrown with throw.
  Rule: When rethrowing the same exception, use the throw keyword by itself.

Catch clauses I

- The type of the argument in the catch block is checked against the type that was thrown.
- Multiple handlers can be chained, with different parameter types.
- The exception is caught by the handler only if the types match (directly or by inheritance). For primitives type, no casting is performed: e.g. an exception thrown with a char cannot be handled by a catch block of int.
- In case of multiple handlers, these are matched in the order of appearance.
- When the thrown objects belong to the same class hierarchy, **the most derived types should be handled first**.

Catch clauses II

- A catch block with an ellipsis (...) as parameter will catch any type of exception.
- A good coding standard is to throw by value and **catch by reference** (to avoid copying the object and to preserve polymorphism).
- The exception object is destroyed only after the exception has been handled (when the (last) catch block completes).
- The exception object is used to transmit information about the error that occurred.
- It is a good practice to create exception classes for certain kinds of exceptions that occur in your programs.

noexcept specifier

- noexcept - is an exception specifier, which is used to indicate that a function cannot throw an exception;
- destructors are generally implicitly noexcept (as they can't throw an exception);
- noexcept is not a compile-time check
  - it is merely a method for a programmer to inform the compiler whether or not a function throws exceptions;
  - the compiler can use this information to enable certain optimizations.
- If a noexcept function does try to throw an exception, then std::terminate is called to terminate the application.

```
void myFunction1() noexcept; // myFunction1 does not throw any exceptions
void myFunction2() noexcept(false); // myFunction2 MIGHT throw exceptions
```

User defined exceptions I

- The exception object thrown can be just about any kind of data structure you like.
- It is a good practice to create exception classes for certain kinds of exceptions that might occur in one's programs.
- The C++ STL provides a base class specifically designed to declare objects to be thrown as exceptions: class std::exception in the header <exception>.
- As of C++17, in STL there are 25 different exception classes that can be thrown (subclasses of std::exception).
  `https://en.cppreference.com/w/cpp/error/exception`

User defined exceptions II

- nothing throws a std::exception directly, and neither should you;
- std::runtime_error (included as part of the stdexcept header) is a popular choice, because it has a generic name, and its constructor takes a customizable message
- You can create a class that inherits from std::exception, the what() method can be overriden (it returns a const char*).

Exceptions

"What good can using exceptions do for me? The basic answer is: Using exceptions for error handling makes you code simpler, cleaner, and less likely to miss errors. But what's wrong with "good old errno and if-statements"? The basic answer is: Using those, your error handling and your normal code are closely intertwined. That way, your code gets messy and it becomes hard to ensure that you have dealt with all errors (think "spaghetti code" or a "rat's nest of tests")." – **Bjarne Stroustrup.**

Exceptions

"Counter-intuitively, the hard part of coding exceptions is not the explicit throws and catches. The really hard part of using exceptions is to write all the intervening code in such a way that an arbitrary exception can propagate from its throw site to its handler, arriving safely and without damaging other parts of the program along the way."" – **Tom Cargill.**

Advantages of using exceptions

- separation of the error handling code from the normal flow of control;
- different types of errors can be handled in one place (inheritance);
- the program cannot ignore the error, it will terminate unless there is a handler for the exception;
- functions will need fewer arguments and return values $\rightarrow$ this makes them easier to use and understand; any amount of information can be passed with the exception.

Downsides of using exceptions

- Exceptions do come with a small performance price to pay.
  - they increase the size of your executable;
  - they may also cause it to run slower due to the additional checking that has to be performed.
- Exception handling is best used when all of the following are true:
  - The error being handled is likely to occur only infrequently.
  - The error is serious and execution could not continue otherwise.
  - The error cannot be handled at the place where it occurs.
  - There isn't a good alternative way to return an error code back to the caller.

Exception-safe code

- If an exception occurs, an exception-safe code means that:
  - **there are no resource leaks;**
  - **there are no visible effects;**
  - **the operation is either completely executed, or not executed at all (transaction).**
- For every managed resource (e.g. memory, file) - create a class.
- Any pointer will be encapsulated in an object which is automatically managed by the compiler.
- Such objects will be created locally in the function (not allocated on the free store).
- This way - one ensures that the destructor is called when the execution leaves scope (even if an exception is thrown).
- Use RAII - Resource Acquisition Is Initialization.