Streams. Files. Exceptions

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

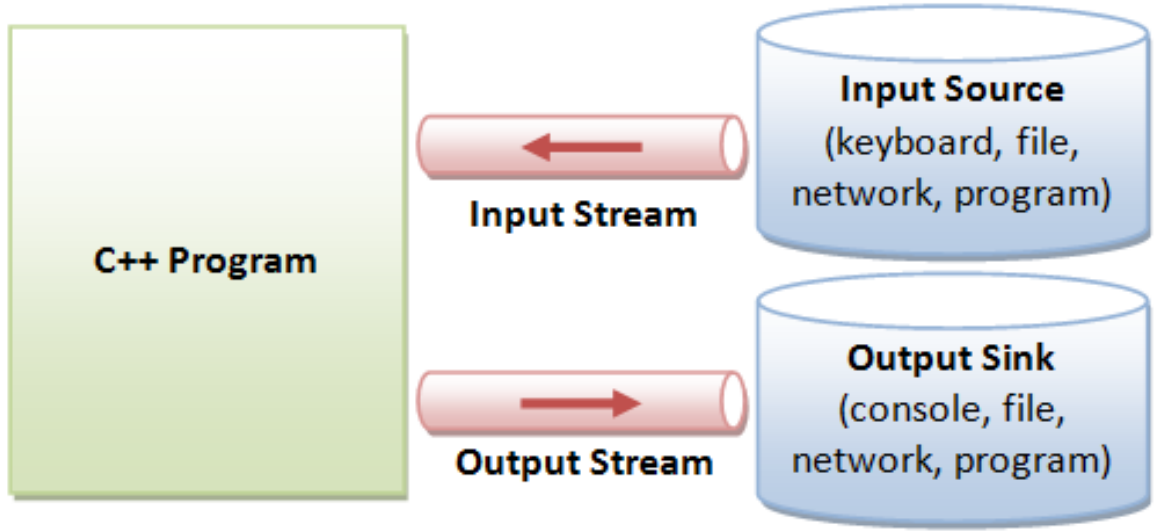
Overview

1. Input/Output streams
2. Insertion and extraction operators
3. Formatting, manipulators, flags
4. Files
5. Exception handling
6. Advantages of using exceptions
7. Exception-safe code

Streams I

- A **stream** is an abstraction for receiving/sending data in an input/output situation.
- Concrete implementations receive data from a specific source (input - keyboard, memory zone, file) and send it to a specific destination (output - display, memory zone, file).
- Streams are generally associated to a physical source or destination of characters, like a disk file, the keyboard, or the console, so the characters read or written to/from the abstraction called stream are physically input/output to the physical device.

Streams II



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)

Figure source: https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp10_IO.html

Streams III

- For example, file streams are C++ objects to manipulate and interact with files; once a file stream is used to open a file, any input or output operation performed on that stream is physically reflected in the file.

- 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.

Buffers I

- 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 some kind of stream-buf.
- When the function **put** is called to write a single character, the character is not written directly to the physical destination (ex. file) with which the stream is associated. Instead, the character is inserted in that stream's intermediate buffer.

Buffers II

- When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream).

- This process is called **synchronization** and takes place under any of the following circumstances:
  - When the file is closed: before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written to the physical medium.
  - When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.
  - Explicitly, with manipulators (flush and endl) or with the stream's member function sync().

iostream library I

The **iostream** library is an object-oriented library that provides input and output functionality using *streams*. It was created to be simply extensible and allows using new data types relatively easy.

Elements of the standard iostream library:

- **Basic class templates**:
  - The base of the iostream library is the hierarchy of class templates.
  - The class templates provide most of the functionality of the library in a way that they can be used to operate with any type of elements.

iostream library II

- **Class template instantiations**: There are two standard sets of instantiations of the entire iostream class template hierarchy (https://en.cppreference.com/w/cpp/io):
  - one is narrow-oriented (manipulates elements of type char: e.g. classes like ios, istream, ofstream);
  - the other is wide-oriented (manipulates elements of type wchar_t: e.g. classes like wios, wistream, wofstream).

- **Standard objects**:
  - certain objects that are used to perform input and output operations on the standard input and output are declared in the iostream library.
  - There are 2 types of objects: narrow-oriented (cin, cout, cerr, clog) and wide-oriented (wcin, wcout, wcerr, wclog).

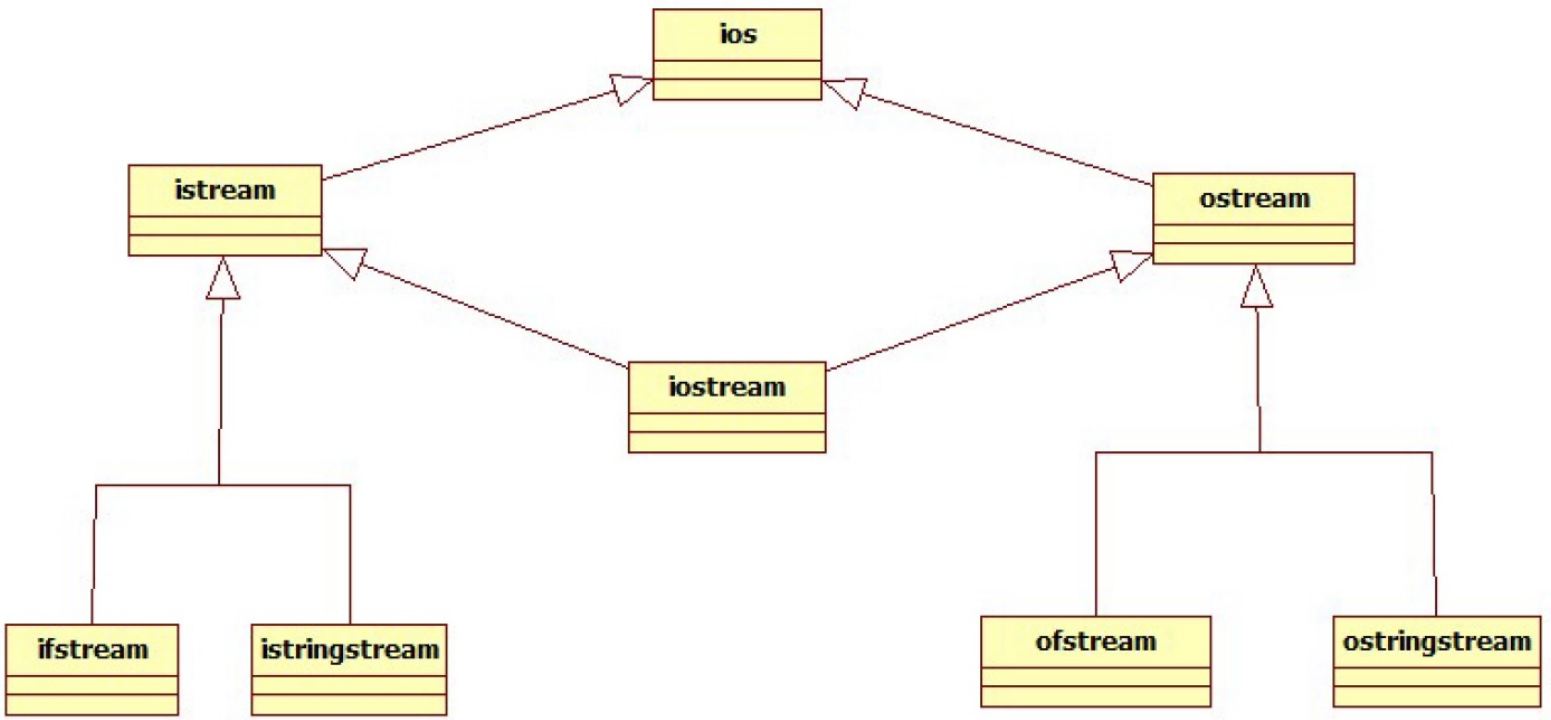
iostream library III

- **Types**: the iostream classes use specially defined types: streampos, streamoff, streamsize - to define positions, offsets and sizes.

- **Manipulators**:
  - global functions that modify properties and formatting settings of the streams.
  - are designed to be used with the insertion ($<<$) and extraction ($>>$) operators performed on *iostream* stream objects.
  - E.g.: endl, hex, scientific.

IO Class hierarchy I

IO Class hierarchy



IO Class hierarchy II

- **ios**: base class for all stream classes (using narrow characters); defines the components of streams that do not depend on whether the stream is an input or an output stream.

- **istream**:
  - used for input operations - read and interpret input from sequences of characters;
  - the standard object cin is an object of this type; it represents the standard input stream.

IO Class hierarchy III

- **ostream**:
  - used for output operations - write sequences of characters and represent other kinds of data;
  - the standard objects cout, cerr, clog are objects of this type:
    - cout - the standard output stream;
    - cerr - the standard error stream.
    - clog - the standard logging stream.
- **iostream**: this class inherits all members from its two parent classes istream and ostream, thus being able to perform both input and output operations.

IO Class hierarchy IV

- **ifstream/ofstream**: input/output stream classes to operate on files.
- **istringstream/ostringstream**:
  - input/output stream classes to operate on strings;
  - objects of this classes use a string buffer that contains a sequence of characters;
  - this sequence of characters can be accessed directly as a string object, using the member function str.

IO Stream header files

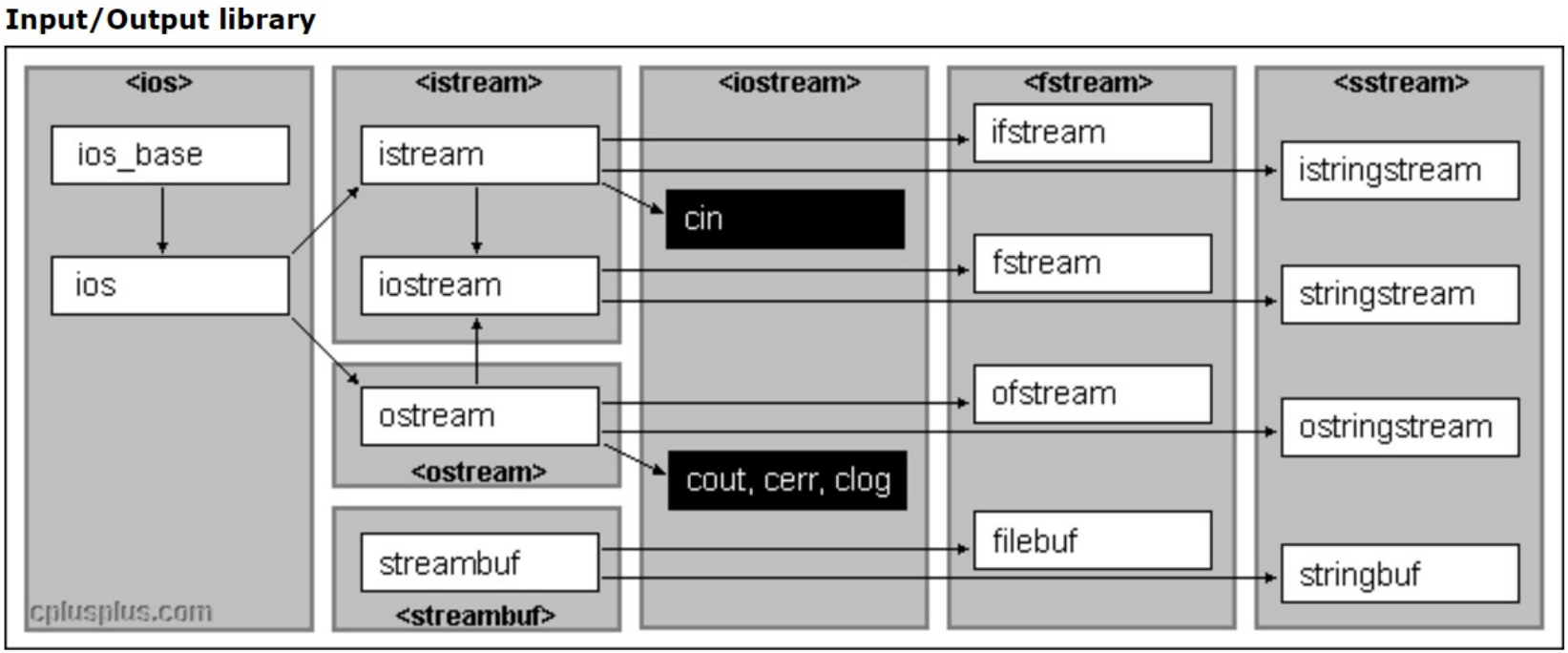


Figure source: http://www.cplusplus.com/reference/iolibrary/

Output (insertion operator) I

- writing operations on a stream (the standard output, in a file or in a memory zone) are performed through the operator $<<$, called **insertion operator**.
- 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.

Output (insertion operator) II

- the << operator is overloaded for standard types, and for custom types the programmer must overload it.

Declaration

```
std::ostream& operator<<(std::ostream&
    stream, const Animal& a);
```

- 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) I

- reading operations are performed through the operator >>, called **extraction operator**.
- 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 can be an expression.

Input (extraction operator) II

- the >> operator is overloaded for standard types, and for custom types the programmer must overload it.

Declaration

```
std::istream& operator>>(std::istream&
    stream, Animal& a);
```

- since the >> operator returns a reference to the current class, it may be chained and the function calls are made from left to right.

Overloading << and >> for user defined types

- Is similar to any operator overloading.
- Both << and >> are a binary operators.
- On the left hand side there needs to be a stream object (like cin or cout) and on the right hand side - the user defined object.
- The overloaded operator functions are usually declared as friends of the class in order to be able to access the private data.

DEMO

Overloading << and >> (*Lecture_7_streams*).

Output formatting

- *ios_base::width(std::streamsize wide)* (header iostream) - minimum number of characters for next output.
- *ios::fill(char fillch)* (iostream) - character used to fill with in the case that the width needs to be elongated to fill the minimum.
- *ios_base::precision(streamsize prec)* - sets the number of significant digits for floating-point numbers.

DEMO

Output formatting (*Lecture_7_streams*).

Example - printing formatted penguin data

DEMO

Printing formatted penguin data (*Lecture_7_streams* - function *formattedPenguinData*).

Manipulators I

- Functions specifically designed to be used in conjunction with the insertion and extraction operators on stream objects.
- They are functions that take a reference to a stream as their only argument.
- Are used to change formatting parameters on streams and to insert or extract certain special characters.
- Are defined in the header iomanip.
- **One can create user defined stream manipulators.**

Manipulators II

- setw, setfill, setprecision - have the same effect as the functions presented for "Output formatting".
- dec, hex, oct - change the numerical base.
- endl - insert new line.
- flush - flush the stream buffer.

DEMO

Manipulators (*Lecture7_demo* - function *formattedPenginData-Manipulators*).

Error state flags

- Indicate the internal state of a stream.
- They are automatically set by some input/output functions of the stream, to signal certain errors.
- https://en.cppreference.com/w/cpp/io/ios_base/iostate

Flag	Description	Verify flag
failbit	I/O operation failed	fail()
badbit	Irrecoverable stream error	bad()
goodbit	OK	good()
eofbit	End of file detected	eof()

Files I

- 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.
- fstream (class) provides an interface to read and write data from files as input/output streams.
- header <fstream>
  - ifstream (class derived from istream) - input file stream
  - ofstream (class derived from ostream) - output file stream

Files II

- The file to be associated with the stream can be specified either as a parameter in the constructor or by calling member function open.
- After all necessary operations on a file have been performed, it can be closed (or disassociated) by calling member close.
- Once closed, the same file stream object may be used to open another file.
- The member function is_open can be used to determine whether the stream object is currently associated with a file.

Opening a file I

- Opening a file = associating a file stream with the file on the disk.
- The constructor of the classes ifstream and ofstream will open the file, if the name of the file is passed as an argument.

```
ofstream myFile{ "example.txt" };
```

- Alternatively, a file can be opened with the fstream member function open.

```
open (filename, mode);
```

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 on opening. 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 it already existed, its previous content is deleted and replaced by the new one.

Opening a file III

- These flags can be combined using the bitwise operator OR (|).
- The default mode for opening a file with ofstream constructor is to create it if it does not exist, or delete everything in it if something does exist in it.
- The default mode for opening a file with ifstream constructor is to open it for input operations (ios::in).
- The default mode for opening a file with fstream constructor is to open it for input and output operations (ios::in | ios::out).

Opening a file IV

- To check whether a file stream was successfully opened for a file - use the method is_open() (returns a boolean value).

```
if ( !myFile.is_open() )
{
    // The file could not be opened
}
else
{
    // Safely use the file stream
}
```

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.

Reading from a file I

- The ifstream's flag EOF (end of file) gets set only **after** a failed attempt to read past the end of the file.

DEMO

Reading from a file (*Lecture_7_files* - function *testRead*).

- In case of corrupted data, the stream might fail to read them and EOF will never be reached (infinite loop).

Reading from a file II

- 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;
  - read while the boolean value of the stream is true.

DEMO

Reading from a file (*Lecture_7_files* - function *testReadCorrected*).

  - This can also be used to validate user input.

Read/write user defined objects

DEMO

Reading/writing spaceships' data (*Lecture_7_files* - functions *readSpaceships*, *writeSpaceships*).

Exception handling I

- **Exception** - exceptional/abnormal circumstance that arises while a program is running, causing the interruption of the normal flow of execution.
- **?** What are some examples of exceptions?
- How can one do "exception handling" in a multi-layer application?

Exception handling II

Adressing the problem without exceptions (e.g. in C)

- Error flags (global variables).
- Return codes (error codes).

- Problems with these approaches:
  - By default, the error is ignored (unless the error flag or the error code is verified).
  - Ambiguity about which call failed.
  - Difficult to chain: if one function (in the call chain) ignores the error ("breaks the chain") $\Rightarrow$ this is no longer propagated (it is lost).
  - Code is tedious to read and write.

Exception handling III

Exception handling

- an organised way of managing the exceptional situation occurring during execution.

- Elements:
  - **try block** - marks the instruction block that might cause problems (throw exceptions).
  - **catch block** - immediately follows the try block and holds the code that deals with the problem.
  - **throw statement** - mechanism by which the problematic code (which could generate exceptions) notifies the calling code.

Exception handling IV


```
void testTryCatch()
{
    try
    {
        // code that may throw an exception
    }
    catch (ErrorClass& 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
    }
}
```

Exception handling V

DEMO

Exceptions handling (*Lecture_7_exceptions* - function *exceptionHandling*).

Exceptions - execution flow I

- Place the source code that you want guarded against errors inside a try block.
- Then construct one or more catch program blocks that act as the error handler.
- If the code in the try block (or any code called from the try block) throws an exception, the try block immediately transfers the control to the exception handler (catch block).
- The catch program block doesn't have to be in the same function as the one in which the exception is thrown.

Exceptions - execution flow II

- The thrown exception propagates.
- When an exception is thrown, the system starts "unwinding the stack", looking for the nearest catch block.
- If the catch block is not found in the function that threw the exception, the system looks in the function that called the throwing function.
- All local (static) variables of functions along the way are correctly deallocated. This ensures no resource leaks.

Exceptions - execution flow III

- This search continues "down" the function-call stack. If the 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. Re-thrown exceptions cannot be handled by the same try-catch block - they are thrown up to the calling scope.

DEMO

Execution flow (*Lecture_7_exceptions* - functions *exceptionExecutionFlow1, exceptionExecutionFlow2*).

Exception objects I

- The throw expression can accept one parameter - the thrown object, which is passed as an argument to the exception handler.
- The exception object thrown can be just about any data type: predefined types (int, char, string) or user defined types.
- It is possible to use pointers, but it is **not** recommended.

? Why not?

Exception objects II

- 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.

Exception objects III

- The type of the argument in the catch block is checked against the type that was thrown.
- The exception is caught by the handler only if the types match (directly or by inheritance).
- Multiple handlers can be chained, with different parameter types.
- The handler whose argument type matches the type of the exception specified in the throw statement is executed.

Exception objects IV

- 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.
- A catch block with an ellipsis (...) as parameter will catch any type of exception.

DEMO

Multiple handlers (*Lecture_7_exceptions* - function *multipleHandlers*).

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++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions: class std::exception in the header <exception>.

User defined exceptions II

- If one wants to create a class that inherits from std::exception, the what() method can be overriden (it returns a const char*).

- However, std::exception does not accept a string in the constructor. Therefore, one could also inherit from std::runtime_error, which receives a string as a parameter in the constructor; std::runtime_error is a sub-class of std::exception.

DEMO

User defined exceptions (*Lecture_7_exceptions - SpaceshipValidator.h/.cpp* and function *testSpaceshipException*).

Exception class hierarchy

- The number of try blocks should not have to grow exponentially with the size of the program.
- Exception classes should be organised in hierarchies to minimise the number of exception handlers.
- Group related exception types by using inheritance.
- Exception hierarchies allow for object-oriented error handling (dynamic binding, the same handler used for different type of exceptions).

Exception specification

- The noexcept specifier indicates whether a function will throw exceptions or not.
- The noexcept specification on a function is **not** a compile-time check; it is merely a method for a programmer to inform the compiler whether or not a function should throw exceptions. The compiler can use this information to enable certain optimizations.

```
void f() noexcept; // the function f() does
    not throw any exceptions
void f() noexcept(false); // the function f
    () might throw exceptions
```

- If a function marked noexcept throws exceptions, std::terminate is called (causing abnormal program termination).

Advantages of exceptions over adhoc error handling (control flags, error codes,etc) I

- 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.

Advantages of exceptions over adhoc error handling (control flags, error codes,etc) II

- Bjarne Stroustrup: *"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")."*

Exception-safe code I

"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, 1994)

- It is quite difficult to write code that always behaves predictable, even in cases when exceptions arise.

Exception-safe code II


```
void g()
{
    //...
}
void f()
{
    char* s = new char[10];
    //...
    g(); //if g throws an exception => memory
        leak (the delete instruction will not be
         run)
    //...
    delete[] s;
}
```

Exception-safe code III

What is exception-safe code?

- If an exception occurs:
  - there are no resource leaks;
  - there are no visible effects;
  - the operation is either completely executed, or not executed at all (transaction).

Exception-safe code IV

Writing exception-safe code

- 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.