Object Oriented Programming - Lecture 7 - 8

Diana Borza - diana.borza@ubbcluj.ro

April 5, 2024

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off." (B. Stroustrup)

Content

- C++ standard library - STL
- Containers
- Iterators
- Lambdas
- Streams

Container classes

- **A container** - is a class designed to hold and organize multiple instances of another type (either another class, or a fundamental type).
- Common operations on containers:
  - Create an empty container (via a constructor);
  - Insert a new object into the container;
  - Remove an object from the container;
  - Return the number of objects from the container;
  - Empty the container (remove all the objects from it);
  - Access to the stored objects;
  - Sort the elements (optional).

Growth rates

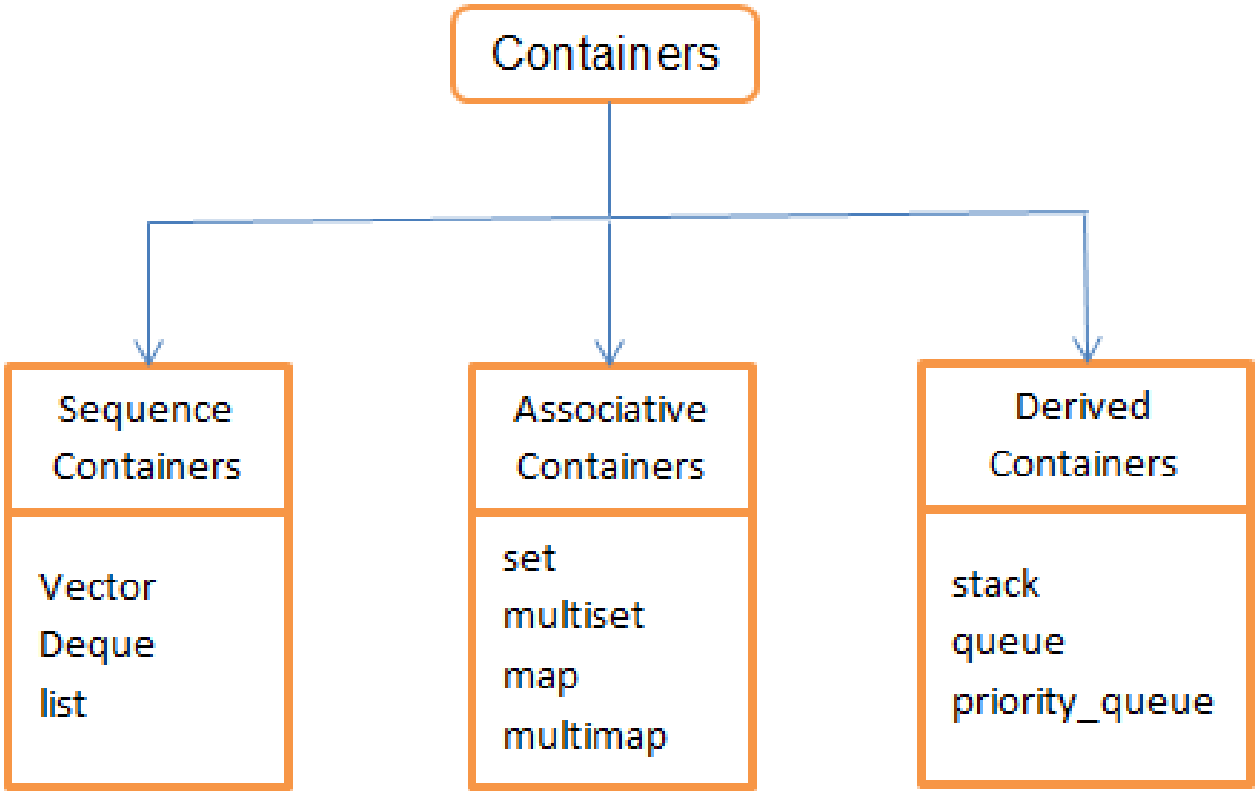
n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 2.4: Growth rates of common functions measured in nanoseconds

C++ STANDARD LIBRARY - STL

- Advantages of using STL:
  - **simplicity**: use well written existing code, instead of writing everything from scratch;
  - **correctness**: well tested, known to be correct;
  - **efficiency**: generally, structures and algorithms from STL have a better performance than the code we write;
  - **maintainability**: code is easier to understand and more straightforward.
  - Using STL your code becomes easier to read, write, maintain and enhance!

STL containers

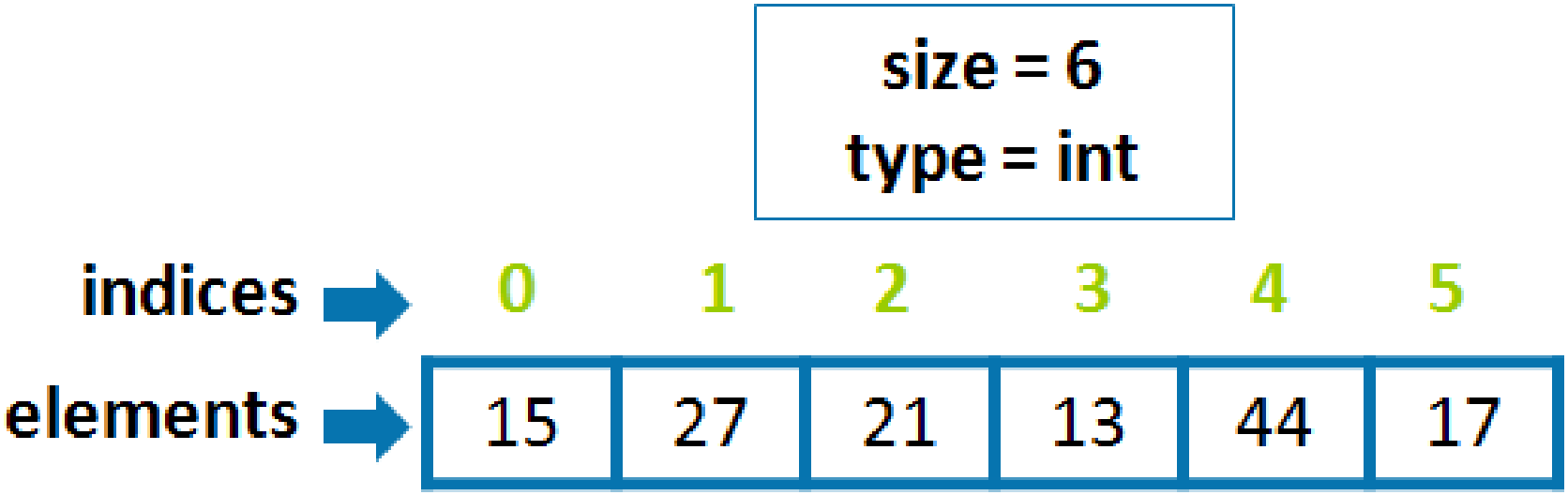


Sequence containers

- Sequence containers are container classes that **maintain the ordering** of elements in the container.
- What is the difference between ordered and sorted?
- You can choose where to insert your element by position.

std::array

- Is a container that encapsulates fixed size arrays.
- Has the same semantics as a struct holding a C-style array T[N] as its only data member.
- Combines the performance and accessibility of a C-style array with the benefits of a standard container (knowing its own size, supporting assignment, random access iterators).



std::vector

- It is a sequence container that encapsulates dynamic size arrays.
- The elements are stored *contiguously* $\rightarrow$ elements can be accessed not only through iterators, but also using offsets to regular pointers to elements.
- The storage of the vector is handled automatically, being expanded and contracted as needed.
- capacity() actual allocated memory; capacity() - size of the vector;
- Complexity of common operations:
  - Random access (*at*, *[] operator* - constant O(1);
  - Insertion or removal of elements at the end *push_back*, *pop_back* - amortized constant O(1);
  - Insertion or removal of elements *insert*, *erase* - linear in the distance to the end of the vector O(n)

std::deque

- Double-ended queue class, implemented as a dynamic array that can grow from both ends.
- Allows fast insertion and deletion at both its beginning and its end.
- As opposed to std::vector, the elements of a deque are not stored contiguously.
- The storage of a deque is automatically expanded and contracted as needed. Expansion of a deque is cheaper than the expansion of a std::vector.
- Complexity of common operations:
  - Random access - constant O(1);
  - Insertion or removal of elements at the end or beginning - constant O(1);
  - Insertion or removal of elements - linear O(n).

std::list

- List containers are implemented as doubly-linked lists;
- Adding, removing and moving the elements does not invalidate the iterators or references. An iterator is invalidated only when the corresponding element is deleted.
- Compared to other sequence containers (array, vector and deque), lists perform generally better in *inserting*, *extracting* and *moving* elements in any position within the container;
- Main drawback: that they lack direct access to the elements by their position.

Associative containers

- Associative containers are containers that automatically sort their inputs when those inputs are inserted into the container. By default, associative containers compare elements using operator $<$.

std::set

- A set is a container that stores unique elements, with **duplicate elements disallowed**.
- The elements are sorted according to their values. Sorting is done using a comparison function that you can redefine.
- **A multiset** is a set where duplicate elements are allowed.

std::map

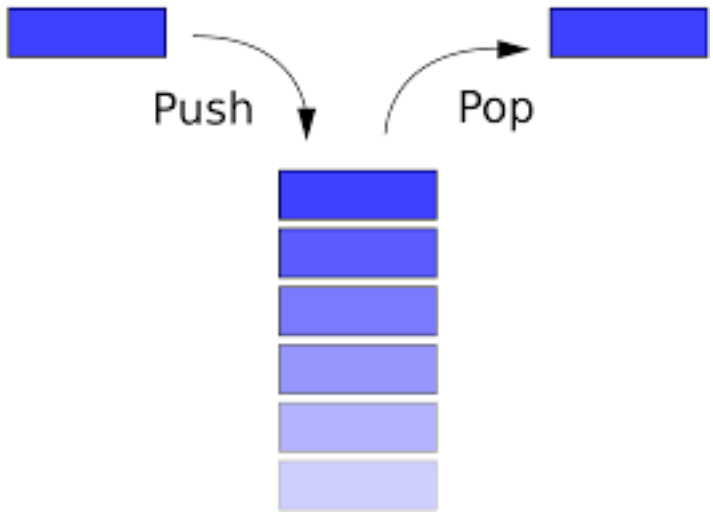
- A **map**: is a set where each element is a *key/value pair*.
- The *key* is used for *sorting* and *indexing* the data, and must be unique.
- The value is the actual data.
- A **multimap** (a dictionary) is a map that allows duplicate keys.

Derived containers

- Derived containers are special predefined containers that are adapted to specific uses.
- They provide a different interface for sequential containers.
  - stack
  - queue
  - priority_queue

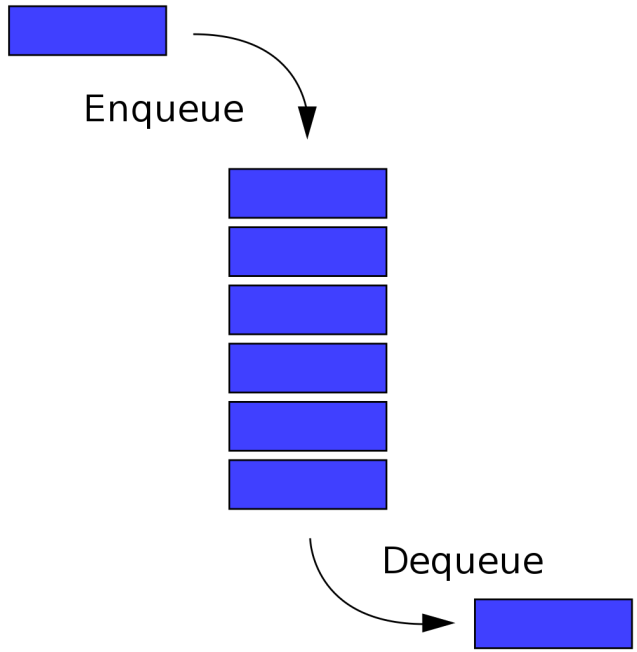
std::stack

- a container where elements operate in a **LIFO** (Last In, First Out) context;
- elements are inserted (pushed) to the front of the container;
- elements are removed (popped) from the front of the container;
- use deque as their default sequence container.

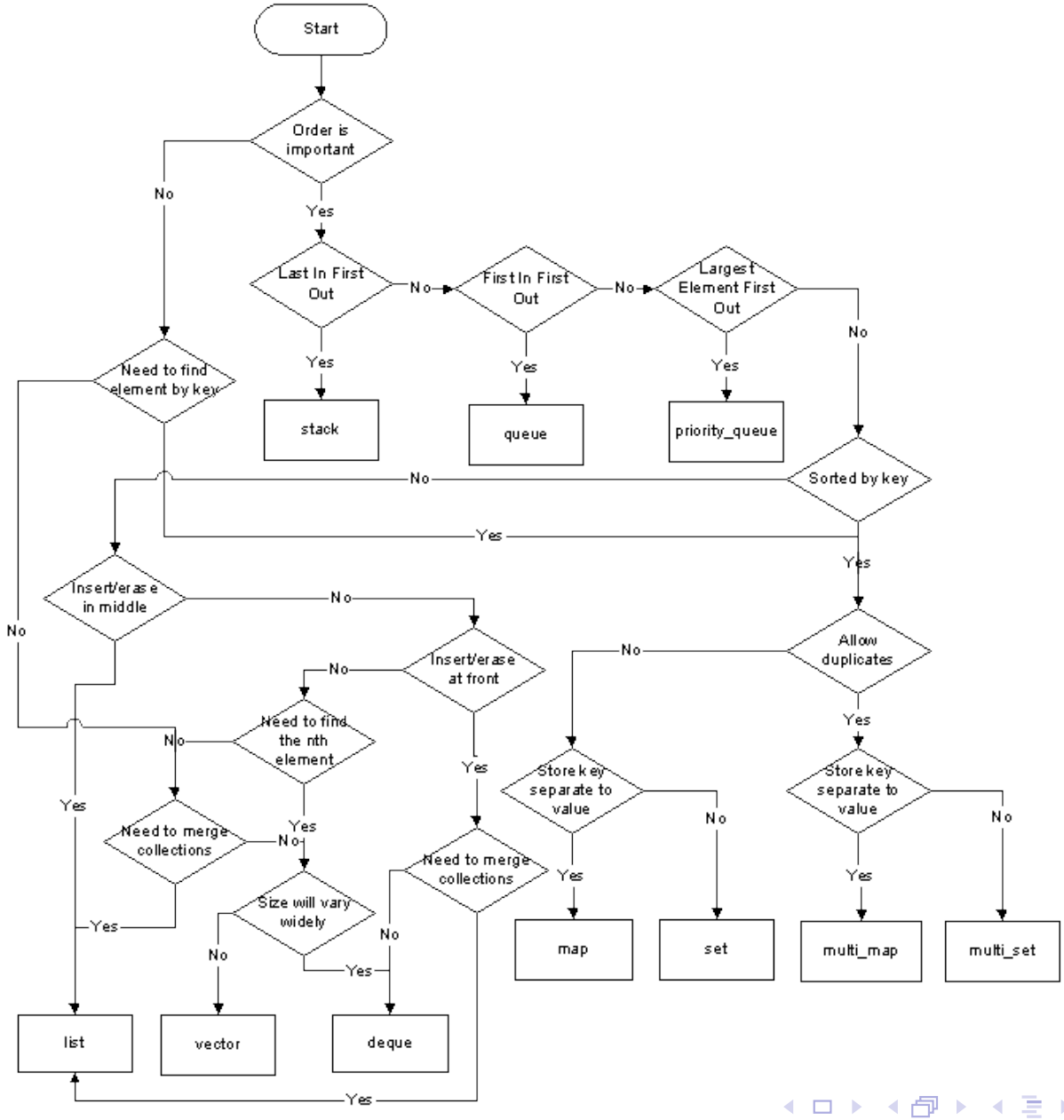


std::queue, std::priority_queue

- a container where elements operate in a **FIFO** (First In, First Out) context;
- elements are inserted (pushed) to the back of the container;
- elements are removed (popped) from the front of the container;
- use deque as their default sequence container.
- A **priority queue** is a queue where the elements are sorted (via operator<). At push the element is sorted in the queue. At pop (deque) the element with the highest priority is removed.



Decisions, decisions...



Iterators I

- An **Iterator** is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented.
- ? What OOP feature is this ?
- Can be visualized as a pointer to a given element in the container.

Iterators II

- Operators defined in iterators:
  - **Operator*** : dereferences the iterator $\rightarrow$ return the element that the iterator is pointing at.
  - **Operator++**: moves the iterator to the next element in the container.
  - **Operator-** - to move to the previous element.
  - **Operator==** and **Operator!=**: basic comparison operators;
  - **Operator=**: assigns the iterator to a new position.

Iterators III

- All C++ containers provide (at least) two types of iterators:
  - container::iterator - read/write iterator
  - container::const_iterator - read-only iterator
- Each container includes four basic member functions to work with iterators:
  - **begin()**: returns an iterator representing the beginning of the elements in the container.
  - **end()**: returns an iterator representing the element just past the end of the elements.
  - **cbegin()**: returns a const (read-only) iterator representing the beginning of the elements in the container.
  - **cend()**: returns a const (read-only) iterator representing the element just past the end of the elements.

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

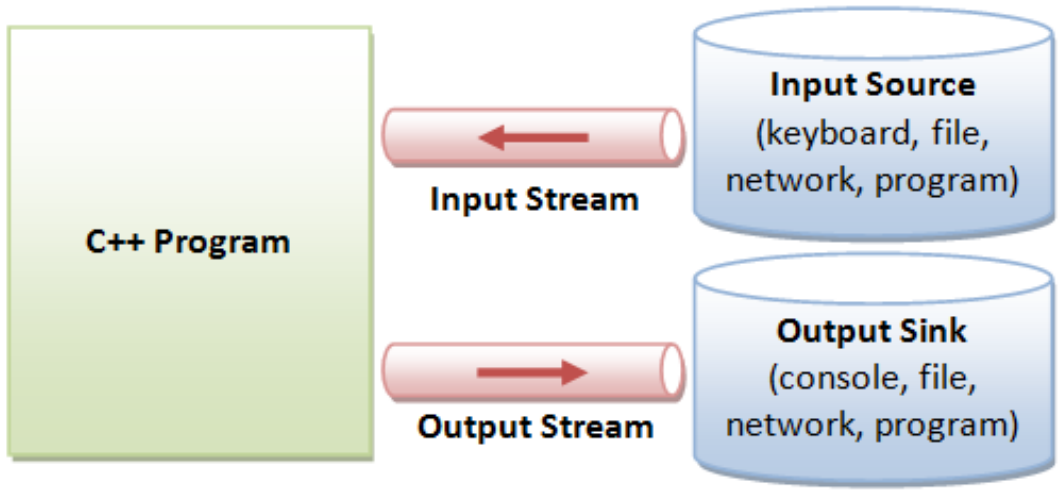
- The **captureClause** clause is used to (indirectly) give a lambda access to variables available in the surrounding scope that it normally would not have access to.
- We need to list all the variables we want to access from within the lambda as part of the capture clause.
- For each variable in **captureClause**, **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 &.

STL algorithms

- algorithms are implemented as functions that operate using iterators'
- std::min_element() and std::max_element() algorithms find the min and max element in a container class;
- std::find() algorithm to finds a value in a container; returns the end of the iterator if the element is not present in the container;
- std::sort() - sorts a container; doesn't work on list container classes!
- std::reverse() - reverses a container;

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)

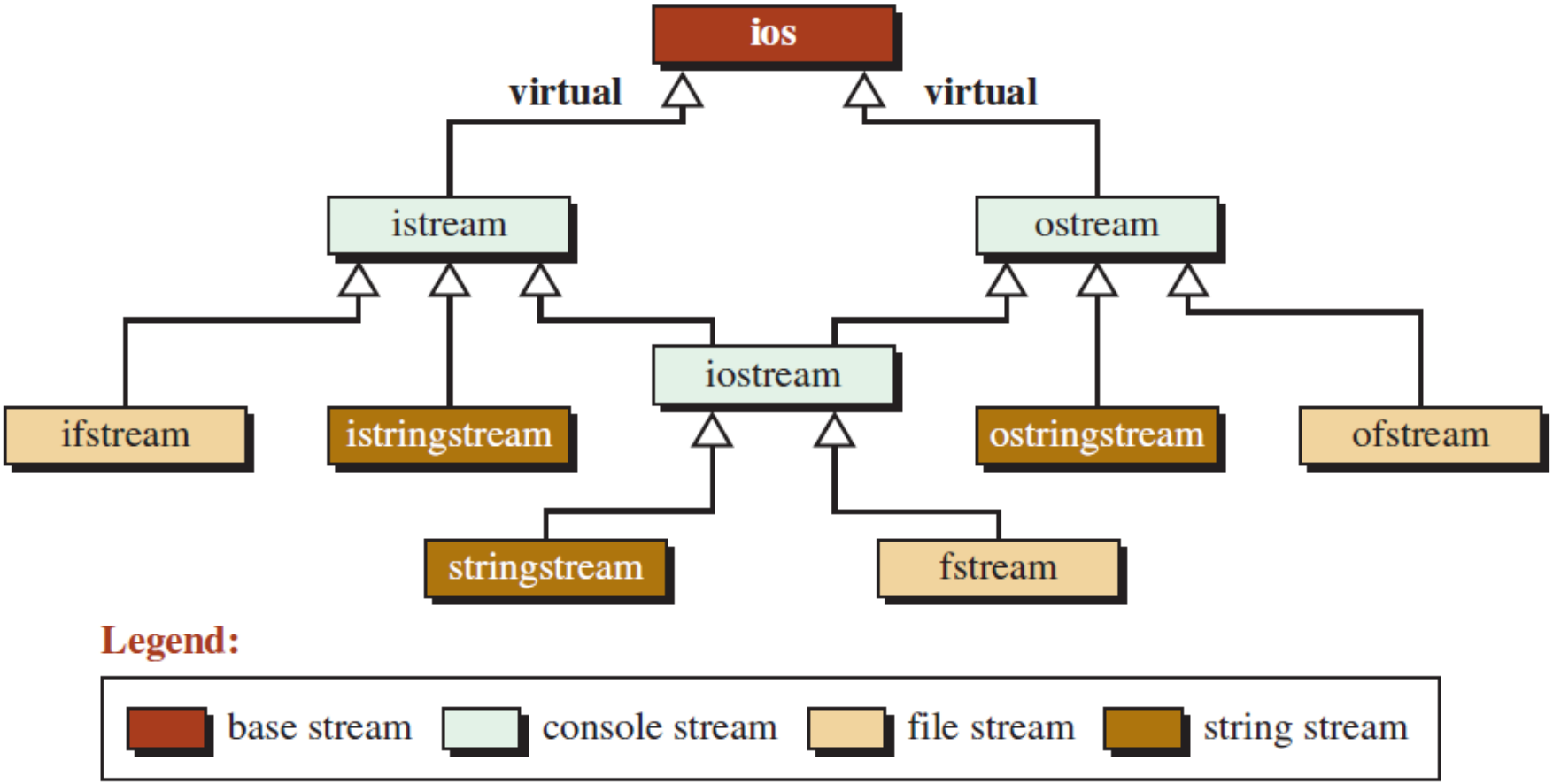
Streams II

- 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

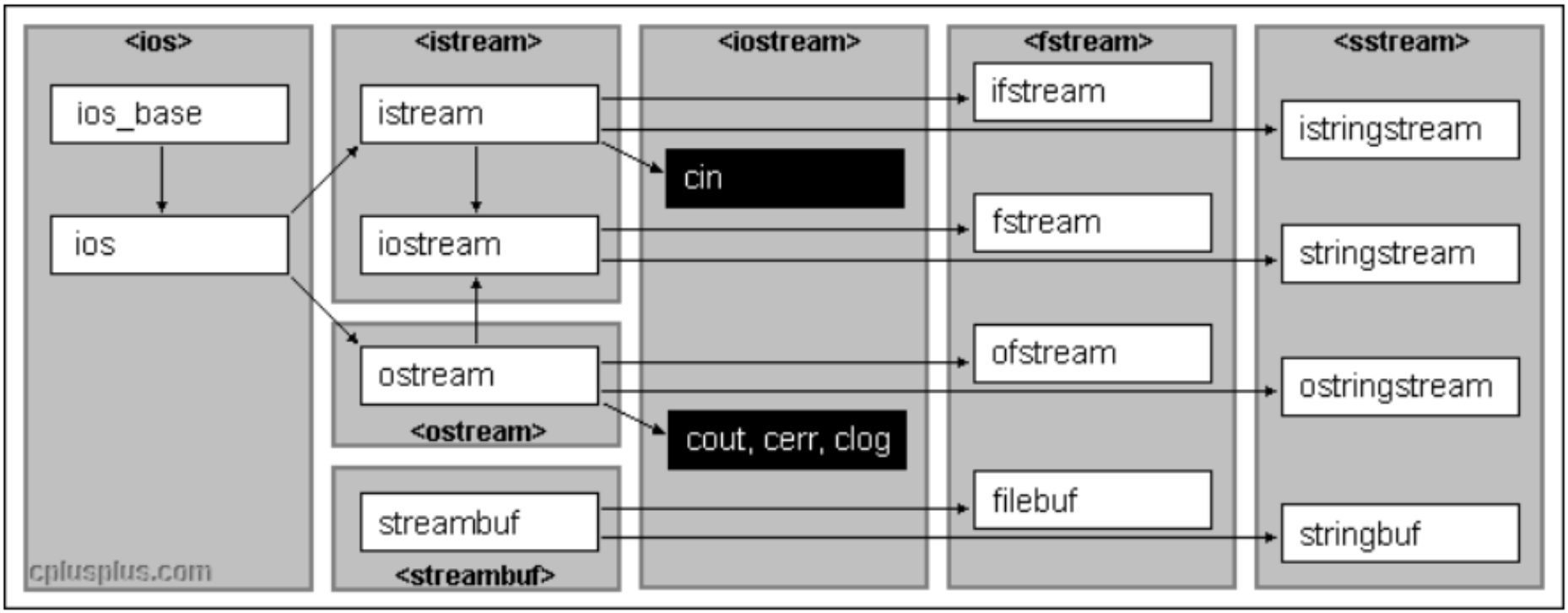
- **Streams are serial**:
  - data elements must be sent to or received from a stream one at a time or in a serial fashion.
  - random access (random reads/writes) are not possible; it's possible to seek a position in a stream and perform a read or write operation at that point.

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.