FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 8

C++ Standard Library

Standard C++ library

standard C++ library: collection of functions, constants, classes, objects and templates that extends the C++ language

- Input/Output Stream Library
- Standard Template Library (STL)
- C Library
- Miscellaneous C++ libraries (e.g. string, new, limits, exception, …)

Object-oriented Library

- **object-oriented library**: Set of classes related by inheritance.
  - Base class of an object-oriented library defines an interface shared by the classes derived from that base class.
  - In the IO library, the istream and ostream classes serve as base classes for the types defined in the fstream and sstream headers.
  - We can use an object of a derived class as if it were an object of the base class.
  - For example, we can use the operations defined for istream on an ifstream object.

IO Library

IO Library

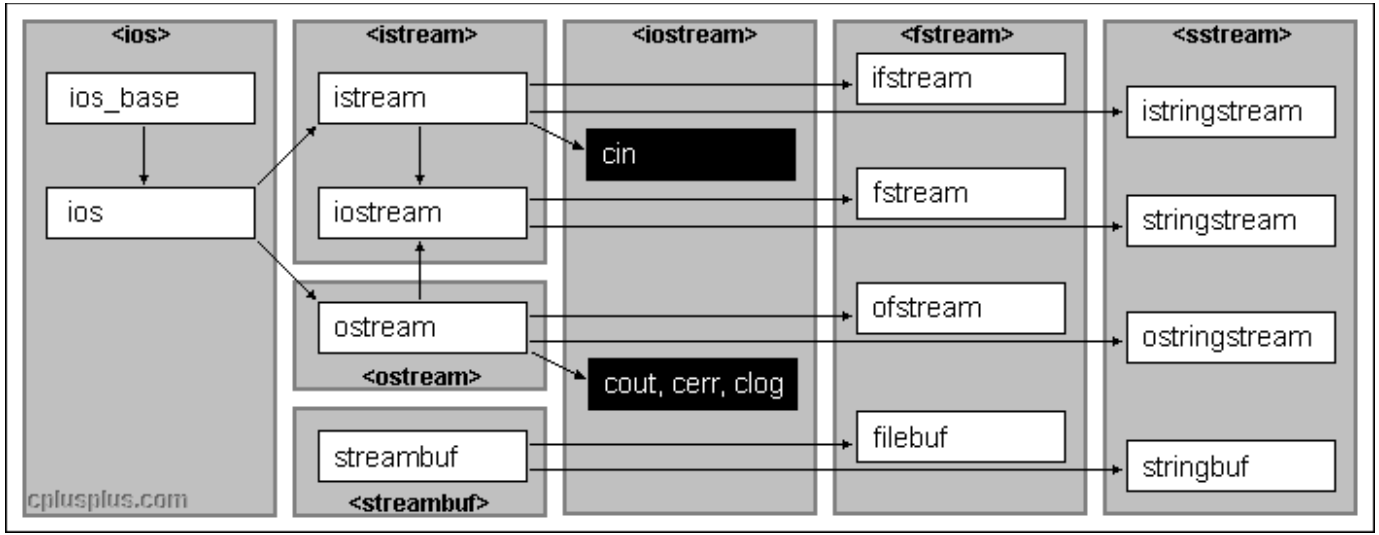


Image from http://www.cplusplus.com/reference/iostream/

Features of IO Library

- International character support
  - Data type wchar_t
  - Classes: wostream, wiostream, …
  - Objects: wcout, wcin, …
- No copy or assign for IO objects
  - Example: ofstream out1, out2; out1 = out2; //error
- Manipulators like endl or skipws

Format control for IO operations


```
#include <iomanip>

void display(ostream& os) {
 ostream::fmtflags curr_fmt = os.flags();
 // print stream and change flags here
 os.flags(curr_fmt);
}

 int i = 2;
 cout << "print " << oct << i << endl;
 // showbase, noshowbase
 // uppercase, nouppercase
 // dec, hex, oct
 // left, right, internal
 // fixed, scientific
 cout.unsetf(ostream::floatfield);
 // flush
 // endl -> after that stream is flushed!
 // unitbuf, nounitbuf -> flush buffer after every output operation

 // setfill(ch) -> fill whitespace with ch
 // setprecision -> set floating-point precision to n
 // setw(w) -> read or write value to w characters
 // setbase(b) -> output integers in base b

 cin >> noskipws;
 while (cin >> ch)
    cout << ch;
 // skipws, noskipws
```

Condition States

- **condition state**: Flags and associated functions usable by any of the stream classes that indicate whether a given stream is usable.
  - States: strm::iostate, strm::badbit, strm::failbit, strm::eofbit
  - functions to get and set these states: s.eof(), s.fail(), s.bad(), s.good()
  - helper functions: s.clear(), s.setstate(flag), s.rdstate()
- Examples
  - istream::iostate oldstate = cin.rdstate(); do_sth(); cin.setstate(oldstate);
  - ifstream is; is.setstate(ifstream::badbit | ifstream ifstream::failbit);

Buffers

- Conditions that cause a buffer to be flushed
  - return from main
  - If buffer becomes full
  - by manipulators like endl
  - By unitbuf manipulator after each output operation
  - We can tie the output stream to an input stream, in which case the output stream buffer is flushed whenever the associated input stream is read
  - Buffers are NOT flushed if the program crashes!
- Example
  - cin.tie(&cout); ostream *old_tie = cin.tie(); cin.tie(0);
  - Buf

fstream

- **fstream**: Stream object that reads or writes a named file.
  - In addition to normal iostream operations, fstream class also defines open and close members.
  - The open member function takes a C-style character string that names the file to open and an optional open mode argument.
  - By default ifstreams are opened with in mode, ofstreams with out mode, and fstreams with in and out mode set.
  - The close member closes the file to which the stream is attached. It must be called before another file can be opened.

File modes

- **file mode**: Flags defined by fstream classes that are specified when opening a file and control how a file can be used.
  - in, out, app, ate , trunc, binary
- Mode is an attribute of a file, not a stream
- Valid file mode combinations:

out	Open for output; deletes existing data in the file
out | app	Open for output; all writes at end of file
out | trunc	Same as out
in	Open for input
in | out	Open for both input and output; read at beginning of file
in | out | trunc	Open for both input and output; deletes existing data in the file
ate	Seek to end immediately after the open

Further IO operations

- Single-Byte operations

is.get()	Puts next byte from istream is in character ch; returns is
os.put(ch)	Puts character ch onto ostream os; returns os
is.get()	Returns next byte from is as an int
is.unget()	Moves is back one byte; returns is
is.putback(ch)	Puts character ch back on is; returns is (DO NOT USE!)
is.peek()	Returns the next byte as an int but does not remove it

- Multi-Byte operations: getline, read, write
- Random access to a stream

seekp, seekg	Reposition marker in an input/output stream
tellg, tellp	Return current position of marker in an input/output stream

- Offset argument in seek: beg, cur, end of stream

File IO in C++

- IO library uses C-style strings to refer to file names for historical reasons
  - use c_str() member function to obtain a C-style string from an IO library string)
- if we reuse a file stream to read or write more than one file, we must clear() the stream before using it to read from another file
  - avoids being in an error state

stringstream

- **stringstream**: Stream object that reads or writes a string.
  - In addition to the normal iostream operations, it also defines an overloaded member named str.
  - Calling str with no arguments returns the string to which the stringstream is attached.
  - Calling it with a string attaches the stringstream to a copy of that string.
- Stringstreams provide conversions and/or formatting

Standard Template Library

Standard Template Library (STL)

abstract data type: Type whose representation is hidden. To use it, we need to know only what operations the type supports.

Container: A type whose objects hold a collection of objects of a given type.

Examples: vector, list

class template: A blueprint from which many potential class types can be created.

To use a class template, we must specify what actual type to use. When we create e.g. a vector, we must say what type this vector will hold (vector<int>, vector<string> ...)

Size Types

size_t: Machine-dependent unsigned integral type defined in cstddef header that is large enough to hold the size of the largest possible array.

size_type: Unsigned Type defined by the string and vector classes that is capable of containing the size of any string or vector, respectively.

difference_type: A signed integral type defined by vector that is capable of holding the distance between any two iterators.

Sequential Containers

Sequential Containers I

- **Container**: A type that holds a collection of objects of a given type.
  - Each library container type is a template type.
  - To define a container, we must specify the type of the elements stored in the container.
  - The library containers are variable-sized.
- **sequential container**: A type that holds an ordered collection of objects of a single type.
  - Elements in a sequential container are accessed by position.

Sequential Containers II

- **Vector**: Flexible-size array
  - Elements in a vector are accessed by their positional index.
  - We add elements to a vector by calling push_back or insert.
  - Adding elements to a vector might cause it be reallocated, invalidating all iterators into the vector.
  - Adding (or removing) an element in the middle of a vector invalidates all iterators to elements after the insertion (or deletion) point.

Sequential Containers III

- **Deque**: Double-ended queue
  - Elements in a deque are accessed by their positional index.
  - Like a vector in all respects except:
    - it supports fast insertion at the front and at the end of the container
    - it does not relocate elements as a result of insertions or deletions.

Sequential Containers IV

- List: Doubly linked list.
  - Supports only bidirectional sequential access
  - Supports fast insertion (or deletion) anywhere in the list.
  - Adding elements does not affect other elements in the list; iterators remain valid when new elements are added.
  - When an element is removed, only the iterators to that element are invalidated.

Sequential Containers V

- **forward_list**: Singly linked list
  - Supports only sequential access in one direction
  - fast insertion (or deletion) at any point in the list.
- **array**: Fixed-size array
  - Supports fast random access.
  - Cannot add or remove elements.

Defining a sequential container

- Initializing container elements
  - As a copy of another container
  - As a copy of a range of elements
  - Allocating and initializing a specified number of elements (only sequential containers)
- Constraints on types a container can hold
  - Element type must support assignment
  - We must be able to copy objects of the element type
- Containers of containers
  - vector< vector<string> > lines;

Iterators

Iterator: A type that can be used to examine the elements of a container and to navigate between them.

off-the-end iterator: The iterator returned by end.

- It is an iterator that refers to a nonexistent element one past the end of a container.

Iterator Ranges

- **iterator range**: A range of elements denoted by a pair of iterators.
  - The first iterator refers to the first element in the sequence, and the second iterator refers one past the last element.
  - If the range is empty, then the iterators are equal (and vice versa—if the iterators are equal, they denote an empty range).
  - If the range is non-empty, then it must be possible to reach the second iterator by repeatedly incrementing the first iterator.
  - By incrementing the iterator, each element in the sequence can be processed.
- **left-inclusive interval**: A range of values that includes its first element but not its last.
  - Typically denoted as [i, j) meaning the sequence starting at and including i up to but excluding j.

Invalidated Iterator

- **invalidated iterator**: An iterator that refers to an element that no longer exists.
  - Using an invalidated iterator is undefined and can cause serious run-time problems.
  - There is no way to examine an iterator whether it has been invalidated
- Hint: make range of code over which an iterator must stay valid as short as possible!

Sequence container Typedefs

Container-Defined Typedefs	
size_type	Unsigned integral type large enough to hold size of largest possible container of this container type
iterator	Type of the iterator for this container type
const_iterator	Type of the iterator that can read but not write the elements
reverse_iterator	iterator that addresses elements in reverse order
const_reverse_iterator	Reverse iterator that can read but not write the elements
difference_type	Signed integral type large enough to hold the difference, which might be negative, between two iterators
value_type	Element type
reference	Element's lvalue type; synonym for value_type&
const_reference	Element's const lvalue type; same as const value_type&

Sequence container operations

- Each container supports operations that let us
  - Add elements to the container: push_back, push_front, insert
  - Delete elements from the container: erase, clear, pop_back, pop_front
  - Determine the size of the container: size, resize, empty, max_size
  - Access elements from the container: back, front, at
- Container elements are copies!
- at() may throw out_of_range exception
- Using a subscript that is out-of-range or calling front or back for an empty container are serious programming errors!

Sequence container operations

- New members: emplace_back, emplace_front, emplace
  - Correspond to: push_back, push_front, insert
  - For push we pass on object of the element type that is then copied into the container
  - For emplace we pass the arguments to the constructor of the element type
  - A new element is constructed directly in space managed by the container

Relational operators

- To be able to compare two containers they must be the same kind of container and hold the same type of elements
- The operators work similar to the string relationals
  - If both containers are the same size and all the elements are equal, then the two containers are equal; otherwise they are not equal
  - If the containers have different sizes but every element of the shorter one is equal to the corresponding element of the longer one, then the shorter is considered to be less than the other
  - If neither container is an initial subsequence of the other, then the comparison depends on comparing the first unequal elements

Assignment and swap

- The assignment operator erases the entire range of elements in the left-hand container and then inserts the element of the right-hand container object into the left-hand container: c1 = c2
- The same holds for the assign operations
- The swap operation swaps the values of its two operands
  - It does not invalidate iterators, no elements are moved
  - After swap, iterators continue to refer to the same elements, although they are now in a different container
  - Since swap does not delete or insert any elements it is guaranteed to run in constant time

Library vector type

- **push_back**: Function defined by vector that appends elements to the back of a vector.
- Use iterators
  - begin(), end()
  - Iterator returned from end() can not be dereferenced
  - const_iterator vs. const iterator
  - Any operation changing the size of a vector makes existing iterators invalid
  - Prefer != instead of < in loops

How a vector grows

- The size of a vector is the number of elements in it
- The capacity of a vector is how many elements it could hold before new memory must be allocated
- Each implementation of vector is free to choose its own allocation strategy
  - It must provide reserve and capacity functions
  - It must not allocate new memory until it is forced to do so
  - How much memory then is allocated is up to the implementation

Deciding which container to use

- Rules of thumb
  - If the program requires random access to elements, use vector or deque
  - If the program needs to insert or delete elements in the middle of the container, use a list
  - If the program needs to insert or delete elements at front or back, but not in the middle, use a deque
  - If we need to insert elements in the middle of the container only while reading input and then need random access to the elements, consider reading into a list and then copying the list into a vector

String revisited

- String only operations
  - The substr function that returns a substring of the current string
  - The append and replace functions that modify the string
  - A family of find functions that search the string
  - A family of compare functions to perform lexicographical comparisons
- Numeric conversions
  - to_string(val)
  - stod(str), stoi(str), …

Container Adaptors I

- **Adaptor**: A library type, function, or iterator that given a type, function, or iterator, makes it act like another.
  - There are three sequential container adaptors: stack, queue, and priority_queue.
  - Each of these adaptors defines a new interface on top of an underlying sequential container.
- The type of the underlying container on which the adaptor is implemented is available via container_type
- Initialization by a container
  - deque<int> deq; stack<int> stk(deq);
  - stack<string, vector<string> > str_stk;

Container Adaptors II

- **stack**: Adaptor for the sequential containers that yields a type that lets us add and remove elements from one end only.
  - Operations: pop, top, push
  - Possible underlying container: vector, list, deque
- **queue**: Adaptor for the sequential containers that yields a type that lets us add elements to the back and remove elements from the front.
  - FIFO storage and retrieval
  - Operations: front, back
  - Possible underlying container: list, deque

Container Adaptors III

- **priority_queue**: Adaptor for the sequential containers that yields a queue in which elements are inserted, not at the end but according to a specified priority level.
  - By default, priority is determined by using the less-than operator for the element type.
  - Possible underlying container: vector, deque

Associative Containers

Pair Type

- **Pair**: Type that holds two public data members named first and second.
  - The pair type is a template type that takes two type parameters that are used as the types of these members
- Example
  - pair<string, string> testpair; testpair = make_pair(“str1”, “str2”);
  - cout << testpair.first << “ “ << testpair.second << endl;

Associative Containers

- **associative container**: A type that holds a collection of objects that supports efficient lookup by key.
  - Containers are set, map, multiset, multimap
  - Elements are ordered by key
- **associative array**: Array whose elements are indexed by key rather than positionally.
  - We say that the array maps a key to its associated value.
- **strict weak ordering**: Relationship among the keys used in an associative container.
  - In a strict weak ordering, it is possible to compare any two values and determine which of the two is less than the other.
  - If neither value is less than the other, then the two values are considered equal.

Map

- **Map**: Associative container type that defines an associative array.
  - Like vector, map is a class template.
  - A map, however, is defined with two types: the type of the key and the type of the associated value.
  - In a map a given key may appear only once.
  - Each key is associated with a particular value.
  - Dereferencing a map iterator yields a pair that holds a const key and its associated value.

Additional typedefs

- **key_type**: Type defined by the associative containers that is the type for the keys used to store and retrieve values.
  - For a map, key_type is the type used to index the map.
  - For set, key_type and value_type are the same.
  - The key type must define the < operator (strict weak ordering!)
- **mapped_type**: Type defined by map and multimap that is the type for the values stored in the map.
- **value_type**: The type of the element stored in a container.
  - For set and multiset, value_type and key_type are the same.
  - For map and multimap, this type is a pair whose first member has type const key_type and whose second member has type mapped_type.

Map operations

- Subscripting a map behaves differently from subscripting a vector or string
- Using an index that is not already present adds an element with that index to the map
- The map subscript operator return a value of type mapped_type
- Further map operations
  - insert, count, find, erase

Set

- **Set**: Associative container that holds keys
  - In a set, a given key may appear only once
  - Also here keys are read-only
- set supports the same operations as map
  - Except: set has no subscript operator and does not define mapped_type

Multimap and Multiset

- **Multimap**: Associative container similar to map except that in a multimap, a given key may appear more than once.
- **Multiset**: Associative container type that holds keys
  - In a multiset, a given key may appear more than once
- Operations
  - m.lower_bound(k): returns an iterator to the first element with key not less than k
  - If key is not in container, lower_bound refers to first point at which this key could be inserted while preserving the element order within the container
  - m.upper_bound(k): returns an iterator to the first element with key greater than k
  - m.equal_range(k), returns a pair of iterators, first is lower_bound(k), second upper_bound(k)

Unordered Containers

- Associative containers that use hashing rather than a comparison operation on keys to store and access elements
- Performance depends on quality of hash function
- **unordered_map, unordered_set , unordered_multimap , unordered_multiset**

STL container reference

- STL Containers member map:
http://www.cplusplus.com/reference/stl/
- Container choice
http://adrinael.net/containerchoice.png

Generic Algorithms

Generic Algorithms I

- **Generic algorithms**: Type-independent algorithms.
- Requirements of an algorithm
  - We need a way to traverse the collection: we need to be able to advance from one element to the next
  - We need to be able to know when we have reached the end of the collection
  - We need to be able to compare each element to the value we want
  - We need a type that can refer to an element's position within the container or that can indicate that the element was not found
- Generic algorithms never execute container operations
  - they operate solely in terms of iterators and iterator operations
- First look for suitable container operations before using equivalent generic algorithms (especially for lists!)

Generic Algorithms II

- Eliminating duplicates in a vector of strings called words
  - Step 1: sort (words.begin(), words.end());
  - Step 2: reorder words so that each word appears once in the front portion of words and return an iterator one past the unique element by unique

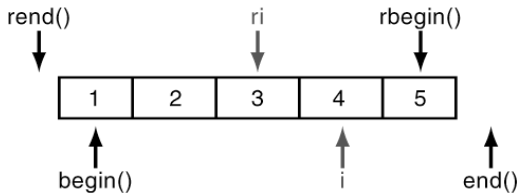
    vector<string>::iterator end_unique = unique(words.begin(), words.end());
  - Step 3: words.erase(end_unique, words.end());
- Note: algorithms never directly change the size of a container
  - If we want to add or remove elements, we must use a container operation

Generic Algorithms III

- **Predicate**: Function that returns a type that can be converted to bool.
  - Often used by the generic algorithms to test elements.
  - Predicates used by the library are either unary (taking one argument) or binary (taking two).

Iterators revisited I

- **insert iterator**: Iterator that uses a container operation to insert elements rather than overwrite them.
  - When a value is assigned to an insert iterator, the effect is to insert the element with that value into the sequence.
- **reverse iterator**: Iterator that moves backward through a sequence. These iterators invert the meaning of ++ and --.
- **off-the-end iterator**: An iterator that marks the end of a range of elements in a sequence.
  - The off-the-end iterator is used as a sentinel and refers to an element one past the last element in the range.
  - The off-the-end iterator may refer to a nonexistent element, so it must never be dereferenced.



Iterators revisited II

- **Inserter**: Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses insert to add elements just ahead of the element referred to by the given iterator.
- **back_inserter**: Iterator adaptor that takes a reference to a container and generates an insert iterator that uses push_back to add elements to the specified container.
- **front_inserter**: Iterator adaptor that given a container, generates an insert iterator that uses push_front to add elements to the beginning of that container.

Iterators categories I

- **iterator categories**: Conceptual organization of iterators based on the operations that an iterator supports.
  - Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories.
  - The algorithms use iterator categories to specify what operations the iterator arguments must support.
  - As long as the iterator provides at least that level of operation, it can be used.
  - For example, some algorithms require only input iterators.
  - Such algorithms can be called on any iterator other than one that meets only the output iterator requirements.
  - Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

Iterators categories II

1. **input iterator**: Iterator that can read but not write elements.
2. **output iterator**: Iterator that can write but not read elements.
3. **forward iterator**: Iterator that can read and write elements, but does not support --.
4. **bidirectional iterator**: Same operations as forward iterators plus the ability to use to move backward through the sequence.
5. **random-access iterator**: Same operations as bidirectional iterators plus the ability to use the relational operators to compare iterator values and the ability to do arithmetic on iterators, thus supporting random access to elements.

Structure of Generic algorithms

- Most of the algorithms take the forms
  - alg(beg, end, other parms);
  - alg(beg, end, dest, other parms);
  - alg(beg, end, beg2, other parms);
  - alg(beg, end, beg2, end2, other parms);
- Distinguishing versions that take a value or predicate
  - find (beg, end, val);
  - find_if (beg, end, pred);
- Distinguishing versions that copy from those that do not
  - reverse (beg, end);
  - reverse_copy (beg, end, dest);
- More information:
  - http://www.cplusplus.com/reference/algorithm/