CSCI251/CSCI851 Autumn-2020
Advanced Programming (**SGPd**)

C++ good practice:
Part D

On we go …

- The last set finished at the end of Part I of the textbook.
- In this set we begin with Part II, which starts on page 307.
- Part II deals with the C++ library.

Page 310-311: IO classes

- For IO processing there are 3 header files you are likely to use:
  - `iostream` : General input/output streams.
  - `fstream` : File streams.
  - `sstream` : String streams.
- The types within those are related through inheritance, so for example, `ifstream` and `istringstream` are derived from `istream`.
  - The interaction with the derived types is the same as with the base type.

- IO objects, so stream objects, don’t have copy or assign.
- So you cannot do something like

```
ofstream out1, out2;
out1=out2;
```

- This also means that when we want to pass streams, we must do so by reference.
- Furthermore reading or writing an IO objects changing the state so the reference cannot be `const`.

Page 313: IO Library Condition State

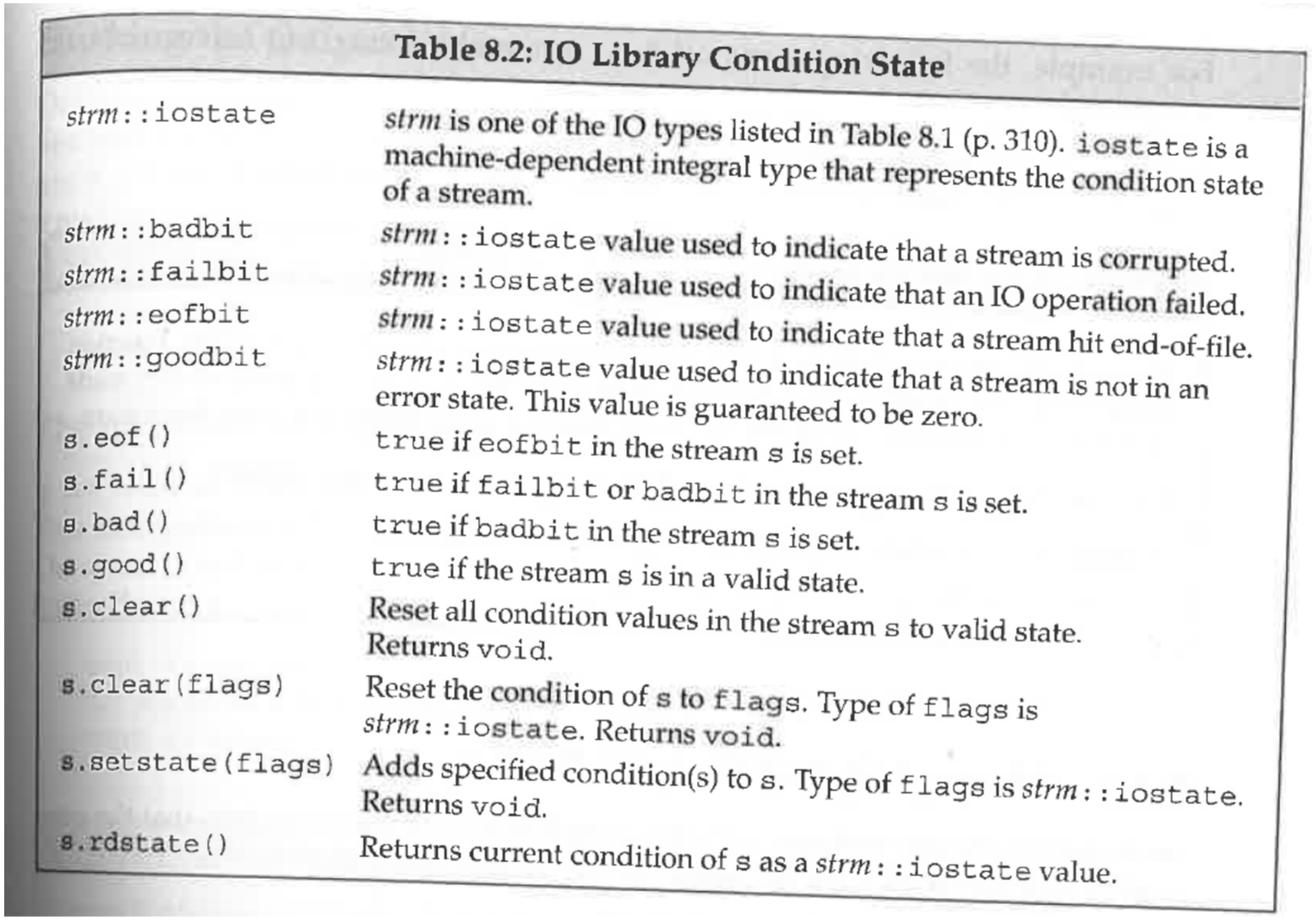

Table 8.2: IO Library Condition State

strm::iostate	*strm* is one of the IO types listed in Table 8.1 (p. 310). iostate is a machine-dependent integral type that represents the condition state of a stream.
strm::badbit	*strm*::iostate value used to indicate that a stream is corrupted.
strm::failbit	*strm*::iostate value used to indicate that an IO operation failed.
strm::eofbit	*strm*::iostate value used to indicate that a stream hit end-of-file.
strm::goodbit	*strm*::iostate value used to indicate that a stream is not in an error state. This value is guaranteed to be zero.
s.eof()	true if eofbit in the stream s is set.
s.fail()	true if failbit or badbit in the stream s is set.
s.bad()	true if badbit in the stream s is set.
s.good()	true if the stream s is in a valid state.
s.clear()	Reset all condition values in the stream s to valid state. Returns void.
s.clear(flags)	Reset the condition of s to flags. Type of flags is *strm*::iostate. Returns void.
s.setstate(flags)	Adds specified condition(s) to s. Type of flags is *strm*::iostate. Returns void.
s.rdstate()	Returns current condition of s as a *strm*::iostate value.

- This is a copy of Table 8.2 in the textbook.

Page 315: Buffer flushing

- Warning: Buffers aren’t flushed if a program crashes.
- If you are putting output statements in for debugging you need to make sure they are flushed or you may be mislead as to the crash point.
- Options for flushing:
  - `endl` : Newline then flushes.
  - `flush` : Just flushes.
  - `ends` : Null then flushes.

- Hopefully you will remember that `cerr` flushes immediately, `cout` doesn’t.
- We can use a stream manipulator to change cout to flushing immediately.

```
cout << unitbuf;
```

- … and turn it back again …

```
cout << nounitbuf;
```

Pages 315-316: Tying streams

- Reading from an input stream that is tied to an output stream also results in the output stream being flushed.
  - Such tying makes sense for interactive systems.
  - The objects `cout` and `cin` are tied together.
- Tying is done using `tie`, an overloaded function with two forms:
  - Using `tie` with no argument returns a pointer to the `ostream` the object is currently tied to, or the null pointer if the stream isn't tied.
  - `x.tie(&o)` ties the stream `x` to the output stream `o`, as long as `x` isn't already tied to a stream since each stream can be tied to at most one output stream.
    - Multiple streams can be tied to the same output stream.

Page 317: C++11 file names

- We have previously seen something like:

```
ifstream in(ifile);
```

- … with `ifile` the name of a file.
- Pre C++11 `ifile` had to be a C-string, so a C-style character array.
- From C++11 on, `ifile` can be a `string`.

Page 318: fstream → close

- When an fstream object is destroyed, such as when it goes out of scope, close is called automatically on the file it is tied to.
- Here goes code for processing a list of files, with the file being closed each loop when input goes out of scope.

```
 string line;
for (auto p = argv+1; p!= argv +argc;++p) {
        ifstream input(*p);
        cout << endl;
        if (input){
                getline(input,line);
                cout << string(*p) << " : 1st line :" << endl;
                cout << line << endl;
        }
        else
        cerr << "couldn't open " << string(*p) << endl;
}
cout << endl;
```

Page 319-320: File modes

- If you want to preserve existing data in a file, open the `ofstream` in either `app` or `in` mode explicitly.
- When `open` is used, the mode is always set.
  - If it's not explicit, it's implicit with the default used.
    - `ifstream`: default `in`.
    - `ofstream`: default `out`.
    - `fstream`: default `in` and `out`.

Page 326-328: Sequential container overview

- The default container is `vector`.
- There are some guidelines in the lecture notes.
- One important point it that if you are unsure…
  - Write your code using only operations common to both vectors and lists, so use iterators not subscripts and avoid random access.
  - These lets you easily go to list or vector depending on what you end up deciding.

Pages 331- 334: Iterators and Containers

- The iterator range, as specified by a pair of iterators referencing elements of the same container, is pivotal in the use of containers and iterators.
  - It's left inclusive: [`begin`, `end`) …

  … so `end` points to one past the last element.
- When you don't need write access, you `cbegin` and `cend`.
  - Here the c is for const.

Pages 334-336: Defining and initalising a container

- When initialising a container as a direct copy of another one, the container type and element types have to match.
- So the following works …

```
list<string> authors={“Alice”, “Bob”};
list<string> list2(authors);
```

- But the following wouldn’t…

```
deque<string> dq2(authors);
```

- Note the list initialization for `authors`, allowed from C++11.

to `vector<T>`: constructors

- There are a fair few options, listed without the allocator here.
- Default, with an empty vector.

```
vector<int> v0;
```

- Initialisation with values …

```
vector<int> v1(10, -1);  // 10 ints set to -1
```

- Initialisation without values …

```
vector<int> v2(10);  // 10 ints, default set to 0.
```

- By copying from another suitable `vector<T>`.

```
vector<int> v3(v2);
vector<int> v4=v2;  // equiv. to above
```

- Iterator based construction:

```
vector<int> v5(v4.begin(),v4.end());
```

- List initialisation, from C++11.

```
vector<string> words = {"one",
"two", "red", "blue"};
vector<int> numbers{1,2,3,4,5,6,7};
```

Pages 340-341: Comparing container

- A container is compared by an element by element comparison.
  - Equal iff same length and elements all the same pairwise.
  - The comparison is based on comparing the first unequal elements of the container.
  - If the containers differ in size but every element of the shorter is equal to the corresponding element of the larger, the shorter is deemed less.
- You can only use a relational operator, >, >=, <, <=, if that operator is defined for the element type the container is storing.

Pages 341-349: Sequential Container Operations

- Container element are copies:
  - When an object is used to initialize a container, or insert something into the container, it's a copy in the container and relationship is gone.
  - This means changing the element in the container doesn't effect the original object, and vice versa.
  - Using `emplace` passes values to a constructor that's used to make an object in the container directly.
- Insertions for `vector`, `string`, or `deque`.
  - Existing iterators, references and pointers into the container may be invalidated.
    - Deletions will call problems too, as would other resizing operations.
  - Legal to insert anywhere anyway, but it could be expensive.

- Calling `front` or `back` on an empty container is a bad thing!
- STL container member functions that remove elements don’t check their argument(s).

Page 353-355: Warnings re: Iterators

- If an iterator (or pointer or reference) has been invalidated, such as through a resizing operator, using it will result in a run-time error.
  - To avoid this happening it’s a good idea to minimize the region in which the iterator needs to be used.
    - In particular when you are adding or removing elements, the iterator returned by end is usually invalidated so in loops that add/remove elements recalls to end are needed, not a stored iterator.

Page 356-359: Reserving memory

- It’s possible to reserve memory in some containers.
- This doesn’t change the number of elements stored, so the `size()` will be unchanged, but does change how many elements can be stored before space needs to be reallocated.
  - `capacity()` and `reserve()` work for `string`, `vector`.

```
vector<int> vInt;
vInt.push_back(1);
cout << vInt.capacity() << " " << vInt.size() << endl;
vInt.reserve(100);
cout << vInt.capacity() <<  " " << vInt.size() << endl;
```

- The C++11 function `shrink_to_fit()` requests a reduction in the capacity to equal the size.
  - But the implementation but choose to ignore this.
- Implementations may allocate different sizes.
- The function `resize()` requests a change in size, not capacity, meaning elements are initialized.

Pages 360-368 some string things …

- Lots of different ways to construct strings.
- Operation: `s.substr(pos, n)`
  - `n` characters starting from `pos`.
- Other operations: `assign`, `insert`, `erase`, `append`, `replace`.
  - Various overloaded forms of these, i.e. different argument sets.

- Searching operations: `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`
- The function `compare` can be used to check if two strings are equal.
  - Returns +ve, 0 (equal to), -ve.
- C++11 Conversions: `to_string`, `stoi`, `stol`, `stoul`, `stoll`, `stoull`, `stof`, `stod`, `stold`.
  - Failure to convert to a number results in an `invalid_argument` exception being thrown.
  - If it's a value that cannot be represented, `out_of_range` is thrown.

Generic algorithms

- Page 378: STL Algorithms never execute container operations, they operate in terms of iterators and iterator operations.
  - They never directly change the size of underlying containers, including adding or removing elements.
- Page 379: Best to use `cbegin()` and `cend()` with read-only algorithms.
  - Use `begin()` and `end()` if the plan is to use a returned iterator to change an element’s value, such as in `find`.

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

- Page 380: Those algorithms that take a single iterator for a 2nd sequence, so the 2nd or 3rd form above, **assume** that the 2nd sequence is at least as large as the first.
- Page 381: The sequences referenced through the iterators may refer to different kinds of container.
- Page 382: Algorithms that write to a destination iterator **assume** that destination is large enough to hold the number of elements that are going to be written.

Page 386: Passing a function to an algorithm

- Under the other args, we can use a predicate in place of <, for example.
- **Predicates**: Expression that can be called and that return a value that can be used as a condition.
  - Unary: Take a single parameter.
  - Binary: Take two parameters.
- To sort words by size … the stable_sort maintains alphabetical order within the same length words…

```
bool isShorter (const string &s1, const string &s2)
{   return s1.size() < s2.size();             }

sort (words.begin(), words.end(), isShorter);
stable_sort (words.begin(), words.end(), isShorter);
```

Pages 387-401: Lambdas

- Lambda expressions are callable functions, so can be used as predicates.
- The general syntax:

```
[capture list] (parameter list) -> return type { function body }
```

- Must include capture list and the function body, although the capture list may be empty:

```
auto f = [] { return 42; };
cout << f() << endl;
```

- Page 389: Lambdas with function bodies containing anything other than a single return statement that do not specify a return type return `void`.

- Page 390: A lambda being used in a function can only use a variable local to its surrounding function if the lambda captures that variable.
- So this is okay …

```
[size](const string &a)
    { return a.size() >= size; };
```

- … but this isn't …

```
[](const string &a)
    { return a.size() >= size; };
```

- … since `size` wasn't captured.

- Page 391: The capture list is used for local non `static` variables, lambdas can use local `static` variables and variables declared outside the function directly.
  - In particular this means something like `cout` doesn’t need to be passed as a variable.
- Page 393: When we capture a variable by reference, we must ensure that the variables exist at the time the lambda executes.
- Page 394: Keep your lambda captures simple!

Pages 397-401: Bind

- The function `bind` is new to C++11.
- Can be thought of “a general-purpose function adaptor”.
- Takes a callable object …
- … generates a new callable that adapts the parameter list of the original object.

```
auto newCallable = bind(callable, arg_list);
```

- The argument list contains placeholders for the arguments the `newCallable` needs.

Pages 401-402: Insert Iterators

- The three types of insert iterator are created using `back_inserter`, `front_inserter`, or `inserter`.
- We can only use `front_inserter` if the container has `push_front`, since that what a `front_inserter` uses.
- We can only use `back_inserter` if the container has `push_back`, since that what a `back_inserter` uses.

Pages 407-409: Reverse iterators

- Possibly add something here later.
- Not too important at the moment…

Back to generic algorithms

- Page 411: Many compilers won’t complain if we pass the wrong category of iterator to an algorithm ☹
- Page 415: Container specific algorithms: Use the list member versions in preference to the generic algorithms available for `lists` ***and*** `forward_lists`.