Class Test 2025: 1 Hour 30 Minutes – 35 marks

Instructions

- Answer *all* questions. The questions do not carry equal weight.
- For questions which require you to write source code, note that:
  - You only need to specify `#include`'s if specifically asked.
  - For classes, you can give the implementation entirely in the header file, unless directed otherwise.
  - Marks are not awarded solely for functionality but also for good design, making appropriate use of library functions, following good coding practices, and using a modern, idiomatic C++ style.
  - Your code must be easily understandable or well commented.
  - You may use pencil but then you forfeit the right to query the marks.
- Reference sheets are provided.

Question 1 [Total Marks: 10]

Read the description of the STL's `replace` algorithm below and write a set of `doctest` unit tests to verify its behaviour.

> `replace(first, last, old_value, new_value)`
>
> Assigns `new_value` to all the elements in the specified range that compare equal to `old_value`. The function uses `operator==` to compare the individual elements to `old_value`. No value is returned.

In terms of marking, one mark is allocated for each test name, and one or more marks are allocated for the test bodies. The actual number depends on the complexity of the test.

Question 2 [Total Marks: 14]

In figure skating, and other sports, performances are scored by a number of judges. In order to eliminate outliers the highest and lowest scores awarded are discounted. The remaining scores are then averaged to produce the final score.

a) Write a function which accepts a vector of floating point scores and returns the final score according to the method described above. Importantly, *you may not use any looping structures* in this function. (9 marks)

b) Write a simple program which accepts scores from the user which are assumed to be in the range of 1 to 10. The user must type in a “0” to complete entering scores. Error checking for invalid user input is not required.

Once the user has completed entering scores, your program must display all the scores that have been entered as well as the final score which must be calculated using your function from part a). (5 marks)

Question 3 [Total Marks: 11]

a) In Figure 1, *main* is a branch name. *C0* and *C1* represent commits numbered in the order that they are made, and the asterisk shows that HEAD is pointing to the *main* branch. The sequence of Git commands that produced this Git history graph is:

```
git commit
git commit
```

It is assumed that files are added before committing.

Give the *final* Git history graph that results from issuing the following further sequence of commands:

```
git branch feature-1
git branch feature-2
git commit
git switch feature-1
git commit
git commit
git merge main
git switch main
git merge feature-1
git switch feature-2
git commit
```

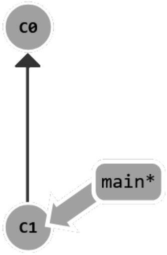

Figure 1: Git history: initial state

(8 marks)

b) Explain what the command `git fetch` does.

(3 marks)

[3 Questions — Available Marks: 35 — Full Marks: 35]

Please fill in the question numbers on the front page of your script.

<string> class

Assume the following declarations:

```
string s, s1, s2;
char c;  char* cs;
string::size_type i, start, len, start1, len1, start2, len2, pos, newSize;
int num;
```

Methods and operators

Constructors and destructors

`string s;`	Creates a string variable.
`string s(s1);`	Creates s; initial value from s1.
`string s(cs);`	Creates s; initial value from cs.

Altering

`s1 = s2;`	Assigns *s2* to *s1*.
`s1 = cs;`	Assigns C-string *cs* to *s1*.
`s1 = c;`	Assigns char *c* to *s1*.
`s[i] = c;`	Sets ith character. Subscripts from zero.
`s.at(i) = c;`	As subscription, but throws `out_of_range` if *i* isn't in string.
`s.append(s2);`	Concatenates s2 on end of s. Same as s += s2;
`s.append(cs);`	Concatenates cs on end of s. Same as s += cs;
`s.assign(s2, start, len);`	Assigns s2[start..start+len-1] to s.
`s.clear();`	Removes all characters from s
`s.insert(start,s1);`	Inserts s1 into s starting at position *start*.
`s.erase(start,len);`	Deletes a substring from s. The substring starts at position *start* and is *len* characters in length.

Access

`cs = s.c_str();`	Returns the equivalent c-string.
`s1 = s.substr(start, len);`	s[start..start+len-1].
`c = s[i];`	ith character. Subscripts start at zero.
`c = s.at(i);`	As subscription, but throws `out_of_range` if *i* isn't in string.

Size

`i = s.length();`	Returns the length of the string.
`i = s.size();`	Same as s.length()
`i = s.capacity();`	Number of characters s can contain without reallocation.
`b = s.empty();`	True if empty, else false.
`i = s.resize(newSize, padChar);`	Changes size to *newSize*, padding with *padChar* if necessary.

Searching

All searches return `string::npos` on failure. The `pos` argument specifies the starting position for the search, which proceeds towards the end of the string (for "first" searches) or towards the beginning of the string (for "last" searches); if `pos` is not specified then the whole string is searched by default.

`i = s.find(c, pos);`	Position of leftmost occurrence of char *c*.
`i = s.find(s1, pos);`	Position of leftmost occurrence of *s1*.
`i = s.rfind(s1, pos);`	As find, but right to left.
`i = s.find_first_of(s1, pos);`	Position of first char in s which is in s1 set of chars.
`i = s.find_first_not_of(s1, pos);`	Position of first char of s not in s1 set of chars.
`i = s.find_last_of(s1, pos);`	Position of last char of s in s1 set of chars.
`i = s.find_last_not_of(s1, pos);`	Position of last char of s not in s1 set of chars.

Comparison	
`i = s.compare(s1);`	<0 if s<s1, 0 if s==s1, or >0 if s>s1.
`i = s.compare(start1, len1, s1, start2, len2);`	Compares s[start1..start1+len1-1] to s1[start2..start2+len2-1]. Returns value as above.

`b = s1 == s2`
`also > < >= <= !=`
	The comparison operators work as expected.

Input / Output	
`cin >> s;`	>> overloaded for string input.
`getline(cin, s);`	Next line (without newline) into s.
`cout << s;`	<< overloaded for string output.
Conversions	
`num = stoi(s);`	Converts string s to integer num
`s = to_string(num);`	Converts integer num (or any numeric type) to string s

Concatenation

The + operator is overloaded to concatentate two strings.

```
s = s1 + s2;
```

Similarly the += operator is overloaded to append strings.

```
s += s2;
s += cs;
s += c;
```

ELECTRICAL AND INFORMATION ENGINEERING
University of the Witwatersrand, Johannesburg
Software Development II

Reference Sheets

1 The `vector` container

Declarations
Assume that `T` is some type (eg, `int`).

```
T e;
vector<T> v, v1;
vector<T>::iterator iter, iter2, beg, end;
(use vector<T>::const_iterator or vector<T>::reverse_iterator if appropriate)
int i, n, size;
bool b;
```

Constructors and destructors

`vector<T> v;`	Creates an empty vector of `T`s.
`vector<T> v(n);`	Creates vector of `n` default values.
`vector<T> v(n, e);`	Creates vector of `n` copies of `e`.
`vector<T> v(beg, end);`	Creates vector with elements copied from range `[beg, end)`.
`vector<int> v{1,2,3};`	Creates vector with elements 1, 2 and 3.
`v.~vector<T>();`	Destroys all elements and frees memory.

Size

`i = v.size();`	Number of elements.
`i = v.capacity();`	Max number of elements before reallocation.
`i = v.max_size();`	Implementation max number of elements.
`b = v.empty();`	True if empty (same as `v.size() == 0`).
`v.reserve(size);`	Increases capacity to `size` before reallocation.

Altering

`v = v1;`	Assigns `v1` to `v`.
`v[i] = e;`	Sets i-th element. Subscripts from zero.
`v.at(i) = e;`	As subscript, but may throw `out_of_range`.
`v.push_back(e);`	Adds `e` to end of `v`; expands `v` if necessary.
`v.pop_back();`	Removes last element of `v`.
`v.clear();`	Removes all elements.
`v.assign(n, e);`	Replaces existing elements with `n` copies of `e`.
`v.assign(beg, end);`	Replaces existing elements with copies from range `[beg, end)`.
`iter2 = v.insert(iter, e);`	Inserts a copy of `e` at `iter` position, and returns its position.
`v.insert(iter, n, e);`	Inserts `n` copies of `e` starting at `iter` position.
`v.insert(iter, beg, end);`	Inserts all elements in range `[beg, end)` starting at `iter` position.
`iter2 = v.erase(iter);`	Removes element at `iter` and returns position of next element.
`iter2 = v.erase(beg, end);`	Removes range `[beg, end)` and returns position of next element.

Access

`e = v[i];`	i-th element. No range checking.
`e = v.at(i);`	As subscript, but may throw `out_of_range`.
`e = v.front();`	First element. No range checking.
`e = v.back();`	Last element. No range checking.

Iterators

`beg = v.begin();`	Returns iterator to first element.
`end = v.end();`	Returns iterator to *after* last element.
`beg = v.rbegin();`	Reverse iterator to first (in reverse order) element.
`end = v.rend();`	Reverse iterator to after last (in reverse order) element.
`auto diff = v.end() - v.begin();`	Compute position difference/number of elements.
`iter++;`	Advance to next element.
`iter += 2;`	Advance 2 elements.
`iter--;`	Go back to the previous element.
`iter -= 3;`	Go back 3 elements.
`if (iter < iter2) { };`	Check if `iter` precedes `iter2`
`if (iter > iter2) { };`	Check if `iter` succeeds `iter2`
`if (iter == iter2) { };`	Check if `iter` equals `iter2`
`if (iter != iter2) { };`	Check if `iter` is not equal to `iter2`

2 The doctest Framework

```
TEST_CASE("This is a test case")
{
    CHECK(expression);          // assertion passes if expression is true
    CHECK_FALSE(expression);    // assertion passes if expression is false
    CHECK_THROWS(expression);   // assertion passes if an exception of
        any type is thrown by expression
    CHECK_NOTHROW(expression);  // assertion passes if no exception is
        thrown by expression
    CHECK_THROWS_AS(expression, exception_type); // assertion passes if
        expression throws an exception of exception_type
    CHECK(doctest::Approx(left) == right);  // assertion passes if left
        is approximately equal to right (floating point comparison)
}
```

Listing 1: doctest framework: syntax and assertions

3 The STL Algorithms

3.1 Numerical Algorithms

The following tables summarize commonly used algorithms from <numeric>. The parameter names used in the signatures are explained below. The algorithms given here do not support parallelization.

Function Parameters	
first, last	Iterators denoting a half-open input range [first,last).
first1, last1, first2	Two-range variants: [first1,last1) with a second range starting at first2.
result	Iterator to the beginning of the destination/output range.
val, init	Initial value or starting value used by the algorithm.
binary_op	Binary operation to combine values (see below).

Common Binary Operation Function Objects (in <functional>)	
std::plus<>	Addition (a + b).
std::minus<>	Subtraction (a - b).
std::multiplies<>	Multiplication (a * b).
std::divides<>	Division (a / b).
std::modulus<>	Modulo (a % b).
std::logical_and<>	Logical AND (a && b).
std::logical_or<>	Logical OR (a b).

Numerical Algorithms

`accumulate(first, last, init)`	Returns the sum of the elements in `[first,last)` added to the value `init`: $\sum a_i + i$
`accumulate(first, last, init, binary_op)`	Accumulates using `binary_op` instead of `+`.
`inner_product(first1, last1, first2, init)`	Returns the sum of element-wise products over `[first1,last1)` and the range starting at `first2`, added to `init`: $\sum a_i \cdot b_i + i$
`inner_product(first1, last1, first2, init, binary_op1, binary_op2)`	Uses `binary_op2` for element-wise combine (replaces multiplying the pairs) and `binary_op1` to accumulate (replaces summing the results).
`iota(first, last, val)`	Fills the range `[first,last)` with sequentially increasing values starting at `val`, then `val+1`, `val+2`, …
`partial_sum(first, last, result)`	Writes the running (prefix) sums of `[first,last)` to `result`: for inputs $\{a_0, a_1, \ldots\}$, outputs $\{a_0, a_0{+}a_1, a_0{+}a_1{+}a_2, \ldots\}$.
`partial_sum(first, last, result, binary_op)`	Uses `binary_op` instead of `+` to form prefix results.
`adjacent_difference(first, last, result)`	Writes the first element unchanged, then the pairwise differences: $\{a_0, a_1{-}a_0, a_2{-}a_1, \ldots\}$.
`adjacent_difference(first, last, result, binary_op)`	Uses `binary_op` to combine adjacent elements instead of subtraction.

Example: Using a standard function object with `accumulate`.

```
#include <numeric>
#include <functional>
#include <vector>

std::vector<int> v{2, 3, 4};
// Product of elements using std::multiplies
auto product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<>{});
// product is 24
```

3.2 General Algorithms

The following tables provide information on some of the algorithms which are available in <algorithm>. Parameters which are used in the function signatures are explained below. All of this information has been adapted from: http://www.cplusplus.com/reference/algorithm/.

Function Parameters	
first and last	Represent a pair of iterators which specify a range. The range specified is [first,last), which contains all the elements between first and last, including the element pointed to by first but not the element pointed to by last.
result	Represents an iterator pointing to the start of the output range.
val, old_value, new_value	Represent elements which are of the same type as those contained in the range.
pred	Represents a function which accepts an element in the range as its only argument. The function returns either true or false indicating whether the element fulfills the condition that is checked. The function shall not modify its argument. pred can either be a function pointer or a function object.

Non-Modifying Sequence Operations	
`all_of(first, last, pred)`	Returns `true` if `pred` returns `true` for all the elements in the specified range or if the range is empty, and `false` otherwise.
`any_of(first, last, pred)`	Returns `true` if `pred` returns `true` for any of the elements in the specified range, and `false` otherwise.
`none_of(first, last, pred)`	Returns `true` if `pred` returns false for all the elements in the specified range or if the range is empty, and `false` otherwise.
`for_each(first, last, fn)`	Applies function `fn` to each of the elements in the specified range. `fn` accepts an element in the range as its argument. Its return value, if any, is ignored. `fn` can either be a function pointer or a function object.
`find(first, last, val)`	Returns an iterator to the first element in the specified range that compares equal to `val`. If no such element is found, the function returns `last`. The function uses `operator==` to compare the individual elements to `val`.
`find_if(first, last, pred)`	Returns an iterator to the first element in the specified range for which `pred` returns `true`. If no such element is found, the function returns `last`.
`find_first_of(first1, last1, first2, last2)`	Returns an iterator to the first element in the range `[first1,last1)` that matches any of the elements in `[first2,last2)`. If no such element is found, the function returns `last1`. The elements in `[first1,last1)` are sequentially compared to each of the values in `[first2,last2)` using `operator==`.
`count(first, last, val)`	Returns the number of elements in the specified range that compare equal to `val`. The function uses `operator==` to compare the individual elements to `val`.
`equal(first1, last1, first2)`	Compares the elements in the range `[first1,last1)` with those in the range beginning at `first2`, and returns `true` if all of the elements in both ranges match, and `false` otherwise. The elements are compared using `operator==`.
`search_n(first, last, count, val)`	Searches the specified range for a sequence of successive `count` elements, each comparing equal to `val`. The function returns an iterator to the first of such elements, or `last` if no such sequence is found.
`binary_search(first, last, val)`	Returns `true` if any element in the specified range is equivalent to `val`, and `false` otherwise. The elements are compared using `operator<`. Two elements, `a` and `b` are considered equivalent if `(!(a<b) && !(b<a))`. The elements in the range *shall already be sorted* according to this same criterion (operator<). The function optimizes the number of comparisons performed by comparing non-consecutive elements of the sorted range, which is especially efficient for random-access iterators.
`min_element(first, last)`	Returns an iterator pointing to the element with the smallest value in the specified range. The comparisons are performed using `operator<`. An element is the smallest if no other element compares less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.
`max_element(first, last)`	Returns an iterator pointing to the element with the largest value in the specified range. The comparisons are performed using `operator<`. An element is the largest if no other element does not compare less than it. If more than one element fulfills this condition, the iterator returned points to the first of such elements.

Modifying Sequence Operations	
`copy(first, last, result)`	Copies the elements in the range [`first`,`last`) into the range beginning at `result`. The function returns an iterator to the end of the destination range (which points to the element following the `last` element copied). The ranges shall not overlap in such a way that `result` points to an element in the range [`first`,`last`).
`transform(first, last, result, op)`	Applies the function `op` to each of the elements in the specified range and stores the value returned by `op` in the range that begins at `result`. `op` can either be a function pointer or a function object. The `transform` function allows for the destination range to be the same as the input range to make transformations *in place*. `transform` returns an iterator pointing to the element that follows the `last` element written in the `result` sequence.
`replace(first, last, old_value, new_value)`	Assigns `new_value` to all the elements in the specified range that compare equal to `old_value`. The function uses `operator==` to compare the individual elements to `old_value`. No value is returned.
`replace_if(first, last, pred, new_value)`	Assigns `new_value` to all the elements in the specified range for which `pred` returns `true`. No value is returned.
`fill(first, last, val)`	Assigns `val` to all the elements in the specified range. No value is returned.
`remove(first, last, val)`	Transforms the specified range into a range with all the elements that compare equal to `val` removed, and returns an iterator to the new end of that range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the elements that compare equal to `val` by the next element that does not, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state. The function uses `operator==` to compare the individual elements to `val`.
`remove_if(first, last, pred)`	Transforms the specified range into a range with all the elements for which `pred` returns `true` removed, and returns an iterator to the new end of that range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the elements for which `pred` returns `true` by the next element that does not, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state.

Modifying Sequence Operations	
`unique(first, last)`	Removes all but the first element from every consecutive group of equivalent elements in the specified range. The function does not alter the size of the container containing the range of elements. The removal is done by replacing the duplicate elements by the next element that is not a duplicate, and signalling the new size of the shortened range by returning an iterator to the element that should be considered its new past-the-end element. The relative order of the elements not removed is preserved, while the elements between the returned iterator and `last` are left in a valid but unspecified state. The function uses `operator==` to compare the pairs of elements.
`reverse(first, last)`	Reverses the order of the elements in the specified range. There is no return value.
`sort(first, last)`	Sorts the elements in the specified range into ascending order. The elements are compared using `operator<`. There is no return value.

Version 3.0, 2025 (`elen3009-doctest-algorithms.tex`)

<locale> Members

The <locale> header file includes functions for character classification. These are listed below.

`bool isalnum(char c)`	Returns true if the character tested is alphanumeric; false if it is not.
`bool isalpha(char c)`	Returns true if the character tested is alphabetic; false if it is not.
`bool iscntrl(char c)`	Returns true if the character tested is a control character; false if it is not.
`bool isdigit(char c)`	Returns true if the character tested is a numeric; false if it is not.
`bool isgraph(char c)`	Returns true if the character tested is alphanumeric or a punctuation character; false if it is not.
`bool isupper(char c)`	Returns true if the character tested is uppercase; false if it is not.
`bool islower(char c)`	Returns true if the character tested is lowercase; false if it is not.
`bool isprint(char c)`	Returns true if the character tested is a printable; false if it is not.
`bool ispunct(char c)`	Returns true if the character tested is a punctuation character; false if it is not.
`bool isspace(char c)`	Returns true if the character tested is a whitespace; false if it is not.
`bool isxdigit(char c)`	Returns true if the character tested is a character used to represent a hexadecimal number; false if it is not.
`char tolower(char c)`	Returns the character converted to lower case.
`char toupper(char c)`	Returns the character converted to upper case.