hrs	/ /20		Exams Office Use Only

University of the Witwatersrand, Johannesburg

Course or topic No(s)	ELEN3009
Course or topic name(s) Paper Number & title	Software Development II
Examination/Test* to be held during month(s) of (*delete as applicable)	November 2023
Year of Study (Art & Sciences leave blank)	Third
Degrees/Diplomas for which this course is prescribed (BSc (Eng) should indicate which branch)	BSc(Eng)(Elec)
Faculty/ies presenting candidates	Engineering and the Built Environment
Internal examiners and telephone number(s)	Dr SP Levitt x77209
External examiner(s)	Mr A Levien
Special materials required (graph/music/drawing paper) maps, diagrams, tables, computer cards, etc)	Computer card for multiple-choice questions

Time allowance

Course Nos	ELEN3009	Hours	3

Instructions to candidates
(Examiners may wish to use this space to indicate, inter alia, the contribution made by this examination or test towards the year mark, if appropriate)

1. Read instructions on page 1 of exam
2. Available marks: 110 - Full marks: 100
3. Closed-book exam
4. Basic scientific calculator permitted

Internal Examiners or Heads of School are requested to sign the declaration overleaf

1. As the Internal Examiner/Head of School, I certify that this question paper is in final form, as approved by the External Examiner, and is ready for reproduction.

2. As the Internal Examiner/Head of School, I certify that this question paper is in final form and is ready for reproduction.

(1. is applicable to formal examinations as approved by an external examiner, while 2. is applicable to formal tests not requiring approval by an external examiner—Delete whichever is not applicable)

Name:_____ Signature:_____

(THIS PAGE NOT FOR REPRODUCTION)

Instructions

- Answer *all* questions. The questions do not carry equal weight.
- You may assume that all code snippets given include the required header files and the directive: `using namespace std;`
- Pencil may be used.
- 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.
- Reference sheets are provided in an appendix.

Question 1 [Total Marks: 40]

For each of the multiple choice questions below, there may be *more than one correct answer*. Mark the *correct answers* on the multiple choice card that is provided. Each question counts for five marks.

A *negative marking scheme* is used so any incorrect answers which are selected for a particular question will lower your mark for that question. Note, however, that you cannot score less than zero for any single question.

1.1 Which of the following statements concerning unit tests are *true*?

a) Tests should not contain branching logic because then they cannot be run in any order.

b) Tests can be written to indirectly test the code within `private` member functions.

c) For every `public` class member function there should be a corresponding test.

d) A test that takes a second to run would be considered a fast test.

e) AAA stands for Anticipate, Act, Assert.

1.2 Which of the following statements are *true*?

a) `getStudentVector` is an example of an intention-revealing function name.

b) A monolithic class is a class which has too many responsibilities.

c) Data classes are classes which contain only data members and no member functions.

d) `char c = 'a';` is an example of violating the DRY principle.

e) The Refused Bequest code smell means that child classes need functionality that is not offered by the parent class.

1.3 Given the following program, which of the statements concerning the program are *true*?

```
1  int main()
2  {
3     auto greeting = new string{"Hello"};
4
5     greeting->append("!");
6     delete greeting;
7
8     auto number = new int{3};
9     delete greeting;
10
11    cout << *number << endl;
12    delete number;
13
14    return 0;
15 }
```

a) `greeting` is known as a raw pointer.

b) The output of this program is undefined.

c) This program contains a memory leak.

d) Replacing the blank line 7 with:

```
greeting = nullptr;
```

results in a correctly formed program.

e) Using the original program given above, if line 3 is replaced with:

```
auto greeting = make_unique<string>("Hello");
```

we can remove line 6, line 9, and line 12, resulting in a program which has no memory leaks.

1.4 Which of the following statements concerning Git and GitHub are *true*?

a) Git is a centralised version control system

b) Assuming that you have a linear commit history, changing any commit in the history will cause the commit hashes of all of the subsequent commits to change.

c) The `rebase` command allows you to combine commits together.

d) The `commit` command pushes code contained in a local repository up to a remote repository.

e) A pull request on GitHub is primarily used for code review.

1.5 Which of the following statements are *true*?

a) If a program's behaviour is undefined, it means that the program's behaviour is not completely predictable.

b) During compilation, a compiler may remove statements containing undefined behaviour.

c) In C++ dereferencing a null pointer is undefined behaviour.

d) Programming languages which enforce well-defined behaviour are often less efficient than those that do not.

e) Disabling compiler optimizations is a good approach to obtaining predictable behaviour from undefined operations.

1.6 Which of the following statements concerning a layered architecture are *true*?

a) Taking a layered approach is applying the Separation of Concerns design principle.

b) The Presentation Layer is typically constructed from existing open-source or commercial libraries and/or frameworks.

c) The Data Access Layer refers to the data store (file system, database, etc) of the application.

d) The standard Windows calculator application has all three layers present.

e) If the Domain/Logic Layer is dependent on the Data Layer, this implies that the Data Layer has no knowledge of the Domain/Logic Layer.

1.7 Which of the following options are *true*?

```
int main()
{
    try
    {
        auto numerator = 10;
        auto denominator = 0;
        auto result = numerator / denominator;
        cout << "Result: " << result << std::endl;
    }
    catch (...)
    {
        cout << "An exception occurred." << std::endl;
    }
    return 0;
}
```

a) The output of the above program is: `An exception occurred.`

b) In order to insert an element, `e`, into a vector, `v`, the following statement is used: `v.insert(iter, e);`
If `iter` does not point to an element in the vector or to the end of the vector, then this is known as a *pre-condition* violation.

c) If an exception is thrown inside a **`try`** block, and there is no corresponding **`catch`** block to handle it, then the program will continue executing after the **`try`** block.

d) Exceptions are a relatively expensive way (in terms of performance) of signalling errors because they involve unwinding the stack.

e) There can only be one **`catch`** handler for every try block.

1.8 Which of the following statements are *true*?

a) Static member functions cannot be overloaded.

b) Static member functions can only access static data members and other static member functions.

c) Static data members can only be accessed by static member functions.

d) Non-static data members can be accessed by static member functions.

e) The output of the following program is: `1 2 3`

```
class Player
{
private:
  int id;
  static int next_id;

public:
  int getID() { return id; }
  Player() { id = next_id++; }
};

int Player::next_id = 1;

int main()
{
  vector<Player> players(3);
  cout << players[0].getID() << " ";
  cout << players[1].getID() << " ";
  cout << players[2].getID() << " ";
  return 0;
}
```

Question 2 [Total Marks: 22]

Answer the following questions, given the `Person` class definition below.

```
class Person
{
public:
  Person(const string &name) : name_(name)
  {
    cout << "Creating a person" << endl;
  }

  Person(const Person &rhs) : name_(rhs.name_)
  {
    cout << "Copying a person" << endl;
  }

  string name() const { return name_; }

  void name(const string &new_name) { name_ = new_name; }

  ~Person()
  {
    cout << "Destructing a person" << endl;
  }

private:
  string name_;
};
```

Listing 1: `Person` class

a) Give the *entire* output of the following program. Note that the output of one line is indicated to assist you. (12 marks)

```
1  int main()
2  {
3      auto people = vector<Person>{};
4      people.push_back(Person{"Lerato"});
5      cout << people.size() << endl;
6      cout << people.capacity() << endl; // will print: 1
7      people.push_back(Person{"John"});
8
9      for (const auto &person : people)
10         cout << person.name() << endl;
11
12     return 0;
13 }
```

Listing 2: A `vector` of `Person`

b) Explain what a `shared_ptr` is, when it should be used, and how it works. (4 marks)

c) Give the *entire* output of the following program which now makes use of `shared_ptr`. As before, the output of one line is indicated to assist you. (6 marks)

```
int main()
{
  auto people = vector<shared_ptr<Person>>{};
  people.push_back(make_shared<Person>("Lerato"));
  cout << people.size() << endl;
  cout << people.capacity() << endl; // will print: 1
  people.push_back(make_shared<Person>("John"));

  for (const auto &person : people)
    cout << person->name() << endl;

  return 0;
}
```

Listing 3: A `vector` of `shared_ptr` to `Person`

Question 3 [Total Marks: 20]

a) Why is "switching on type codes" a code smell? (4 marks)

b) Refactor the code given in Listing 4 and Listing 5 to remove this code smell and produce the same output that is shown in Listing 6. Ensure that your solution can easily be extended to handle new types of employees, and new vacation and salary policies.

(16 marks)

```
class Employee
{
public:
  Employee(string name, string role, int years_worked = 0)
      : name_(name), role_(role), years_worked_(years_worked) {}

  string name() const { return name_; }
  string role() const { return role_; }
  int years_worked() const { return years_worked_; }

private:
  string name_;
  string role_;
  int years_worked_ = 0;
};
```

Listing 4: `Employee` class

```
class Company
{
public:
  void add_employee(const Employee &employee)
  {
    employees.push_back(employee);
  }

  void pay_employees() const
  {
    for (const auto &employee : employees)
    {
      auto vacation_days = 0;
      if (employee.role() == "Tester")
      {
        if (employee.years_worked() > 2)
          vacation_days = 20;
        else
          vacation_days = 15;

        cout << "Paying " << employee.name() << " a monthly salary of R" <<
            BASE_MONTHLY_SALARY_IN_RANDS << ". Employee has " <<
            to_string(vacation_days) << " days of leave." << endl;
      }
      if (employee.role() == "Manager")
      {
        cout << "Paying " << employee.name() << " a monthly salary of R" <<
            BASE_MONTHLY_SALARY_IN_RANDS * 3 << ". Employee has " << 30 << "
            days of leave." << endl;
      }
      if (employee.role() == "Intern")
      {
        cout << "Paying " << employee.name() << " a monthly salary of R" <<
            BASE_MONTHLY_SALARY_IN_RANDS * 1 / 3 << ". Employee has " << 0 <<
            " days of leave." << endl;
      }
    }
  }

private:
  vector<Employee> employees;
  const int BASE_MONTHLY_SALARY_IN_RANDS = 19000;
};

int main()
{
  Company company;
  company.add_employee(Employee{"Keabetswe", "Tester", 1});
  company.add_employee(Employee{"Ajay", "Manager", 2});
  company.add_employee(Employee{"Arnold", "Intern", 0});
  company.pay_employees();
  return 0;
}
```

Listing 5: A simple employee management system

```
Paying Keabetswe a monthly salary of R19000. Employee has 15 days of leave.
Paying Ajay a monthly salary of R57000. Employee has 30 days of leave.
Paying Arnold a monthly salary of R6333. Employee has 0 days of leave.
```

Listing 6: Output of the simple employee management system

Question 4 [Total Marks: 28]

You are developing a computer simulation in which a robot has to navigate a maze. The maze is represented by a square 5×5 grid of cells. Each cell in the grid is either a passage way or a wall. For the first development iteration, passage ways are to be represented by the character 'P' and walls are to be represented by the character 'W'.

The robot is able to move one cell at a time in four possible directions: North, East, South, and West. The robot can move along passages but is blocked by walls. For example, if the robot tries to move North from its current cell and the cell to the North is a passage then the move is successful. However, if the cell to the North is a wall then the move is blocked. Additionally, the robot may not move beyond the boundaries of the maze. A robot controller will be written to place the robot at a location in the maze and control its movements.

You may assume that the following constructor and data members exist for the `Maze` class.

```
1  class Maze
2  {
3  public:
4      Maze(const string& text_file_name)
5      { // DO NOT WRITE THIS CONSTRUCTOR - ASSUME THAT IT EXISTS
6        // the constructor reads the maze stored in the text file
7        // into the 2D character array: maze_
8      }
9
10 private:
11     const static auto WIDTH = 5;
12     const static auto HEIGHT = 5;
13     // create a two-dimensional array of chars using std::array
14     // the top-left corner of the maze is at maze_[0][0]
15     using Map = std::array<std::array<char, WIDTH>, HEIGHT>;
16     Map maze_;
17 }
```

Listing 7: The Maze class

a) Complete the `Maze` class, except for the constructor, and provide a `Robot` class as well as any other classes or types which are deemed necessary or useful for allowing the robot controller to work. You are *not* required to write the robot controller.

   Ensure that the implementation details of the `Maze` class are hidden, so that if there are implementation changes in future versions (to a matrix of ones and zeros, for example), the `Robot` class will not have to change.

   (17 marks)

b) Provide tests which verify the robot's movement in a North direction only. Clearly explain any assumptions that you make concerning the text file containing the maze map, and give the file contents for the maps used in your tests.

   (11 marks)

[4 Questions — Available Marks: 110 — Full Marks: 100]

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

<vector> class

Assume that `T` is some type (eg, int). Assume the following declarations:

```
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
```

Methods and operators

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.
`v.~vector<T>();`	Destroys all elems 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 ith element. Subscripts from zero.
`v.at(i)= e;`	As subscription, 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 position 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];`	ith element. No range checking.
`e = v.at(i);`	As subscription, 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();`	Returns reverse iterator to first (in reverse order) element.

`end = v.rend();`	Returns reverse iterator to *after* last (in reverse order) element.

<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;
```

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.

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;
```

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

Reference Sheets

1 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

2 Algorithms

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

Function Arguments	
`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.
`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.

Modifying Sequence Operations	
`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.

<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.