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 2024
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 are *true*?

a) "Magic numbers" are also known as "literals".

b) The following line of code will produce an output of 6: `cout << 12.5/2;`

c) `string::size_type` represents a kind of magic number.

d) A `long double` has more precision than a `float`.

e) Primitive types do not have constructors.

1.2 For this question, ignore the instruction given in the box at the start of the paper regarding `include` files and the standard namespace. Here, the included header files and namespace directives for each code snippet are explicitly specified.

The Boost libraries provide a `shared_ptr` in the `boost` namespace (in `boost/smart_ptr.hpp`). The STL provides a `shared_ptr` in the `std` namespace (in `<memory>`). Which of the following programs are correctly written so that `a_ptr` is an instance of Boost's `shared_ptr` class, and the program will compile?

a)
```
#include <iostream>
#include <boost/smart_ptr.hpp>
#include <memory>
using namespace std;
using namespace boost;

int main()
{
  shared_ptr<int> a_ptr(new int{6});
  cout << *a_ptr << endl;
}
```

b)
```
#include <iostream>
#include <boost/smart_ptr.hpp>
#include <memory>
using namespace std;

using SmartPtr = boost::shared_ptr<int>;

int main()
{
  SmartPtr a_ptr(new int{6});
  cout << *a_ptr << endl;
}
```

c)
```
#include <iostream>
#include <boost/smart_ptr.hpp>
using std::cout;
using std::endl;

int main()
{
  shared_ptr<int> a_ptr(new int{6});
  cout << *a_ptr << endl;
}
```

d)

```
#include <iostream>
#include <boost/smart_ptr.hpp>
#include <memory>
using namespace boost;

int main()
{
  shared_ptr<int> a_ptr(new int{6});
  std::cout << *a_ptr << std::endl;
}
```

e)

```
#include <iostream>
#include <boost/smart_ptr.hpp>
#include <memory>

int main()
{
  boost::shared_ptr<int> a_ptr(new int{6});
  std::cout << *a_ptr << std::endl;
}
```

1.3 Which of the following statements are *true*?

a) A vector uses more memory than an array of an equivalent size.

b) The following program demonstrates correct use of the vector class.

```
int main()
{
  vector<int> f;
  f.reserve(5);
  f[0] = 12;
  f[1] = 5;

  return 0;
}
```

c) A vector's capacity is always greater than its size.

d) Accessing an element using `v[i]` is faster than using `v.at(i)`.

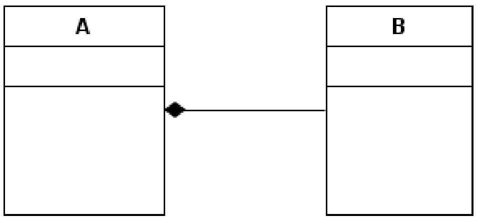
e) Using `push_back` is the most efficient way to add an element to a vector.

1.4 Which of the following statements, concerning UML diagrams, are *true*?

a) UML is an acronym for “Universal Modelling Language”.

b) A class diagram presents the dynamic view of a software design.

c) It is possible to implement the relationship shown in the diagram below by having class `A` have a `unique_ptr` of type `B` as a data member.



d) The diagram above is a type of object diagram.

e) The diagram above implies that an object of class `B` cannot exist without an object of class `A`.

1.5 Which of the following statements are *true*?

a) Given two equally viable design alternatives for a software system, one should favour the simpler alternative over the more reusable alternative.

b) It is prudent to create a specific, individual solution for a specific problem, and to do this a number of times, before attempting to create a single, general solution.

c) C++ templates are used in order to produce solutions that are less general and more constrained.

d) A general solution tends to be flexible.

e) `raylib` is a good example of a library that is reusable.

1.6 Which of the following statements are *true*?

a) There is never a case where the compiler does not know which function to execute for a given line of code.

b) A virtual method table is used to support static dispatch in programming languages.

c) Static dispatch and early binding go hand-in-hand.

d) A normal function call is replaced with a pointer to the function’s implementation during the build process.

e) Virtual tables or *vtables* are defined for each object of a class.

1.7 Which of the following statements are *true* about a *default* constructor?

a) It is a constructor which has no parameters.

b) It is constructor with parameters which all have default arguments.

c) The default constructor of the base class in a class hierarchy is always called when constructing one of the derived classes in the hierarchy.

d) One cannot provide a default constructor for a class which has at least one pure virtual function.

e) The compiler does not provide a default constructor for a class if the programmer writes their own constructor.

1.8 Examine the following C++ program:

```
 1  int main()
 2  {
 3    int a = 10;
 4    int &b = a;
 5
 6    cout << &b << endl;
 7    a++;
 8    cout << &a << endl;
 9    b = b - 1;
10    cout << b << endl;
11    cout << a << endl;
12    cout << a++ << endl;
13
14    return 0;
15  }
```

Which of the following statements are *true*?

a) Line 6 will output the memory address of variable a.

b) Lines 6 and 8 do not produce the same output.

c) The output of line 10 is 9.

d) The output of line 11 is 11.

e) The syntactical meaning of the “&” symbol on line 4 is different to that on line 6.

Question 2 [Total Marks: 18]

a) A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed.

Provide the C++ code for an *integer* stack class which is implemented using a `vector`. Your class should include the following `public` functions:

- `void push(int element)`: Adds an element to the top of the stack.
- `void pop()`: Removes the element from the top of the stack.
- `int top()`: Returns the element at the top of the stack without removing it.
- `bool empty()`: Returns `true` if the stack is empty; `false` otherwise.
- `int size()`: Return the number of elements in the stack.

(8 marks)

b) Provide a set of tests for your class using the doctest framework. (10 marks)

Question 3 [Total Marks: 25]

a) Explain how a `shared_pointer` is implemented in C++. Make use of a drawing to aid your explanation. (5 marks)

b) Answer the following questions, given the code listings below.

```
class Person
{
public:
   Person(const string &name) : name_(name) {}
   string name() const { return name_; }

private:
   string name_;
};
```

Listing 1: `Person` class

```cpp
shared_ptr<Person> printName(shared_ptr<Person> ptr)
{
  cout << "Start printName" << endl;
  auto ptr_2 = ptr;
  cout << ptr_2->name() << " " << ptr_2.use_count() << endl;

  auto ptr_3 = ptr_2;
  cout << ptr_2->name() << " " << ptr_2.use_count() << endl;

  ptr_3 = nullptr;
  cout << ptr->name() << " " << ptr.use_count() << endl;

  ptr_2 = ptr = nullptr;
  // cout << ptr->name() << " " << ptr.use_count() << endl;

  cout << "End printName" << endl;
  return ptr_2;
}
```

Listing 2: `printName` function

```cpp
void printName2(shared_ptr<Person> &ptr)
{
  cout << "Start printName2" << endl;
  cout << ptr->name() << " " << ptr.use_count() << endl;

  // ptr = nullptr;

  cout << "End printName2" << endl;
  return;
}
```

Listing 3: `printName2` function

```cpp
int main()
{
  auto sandile_ptr = make_unique<Person>("Sandile");
  sandile_ptr = nullptr;

  auto jada_ptr = make_shared<Person>("Jada");
  cout << jada_ptr->name() << " " << jada_ptr.use_count() << endl;
  printName(jada_ptr);
  cout << jada_ptr->name() << " " << jada_ptr.use_count() << endl;

  auto karabo_ptr = make_shared<Person>("Karabo");
  printName2(karabo_ptr);
  cout << karabo_ptr->name() << " " << karabo_ptr.use_count();

  return 0;
}
```

Listing 4: The `main` function

i. The `main` program which is given in Listing 4 compiles and runs. Give its *entire* output. (9 marks)

ii. If line 14 in Listing 2 is uncommented will the program still compile and run? If your answer is "yes", give the output that will be produced by line 14. If your answer is "no", give an explanation why. (2 marks)

iii. Assuming the code listings are as given, if line 6 in Listing 3 is uncommented will the program compile and run? Give an explanation for your answer. (3 marks)

iv. On line 3 of Listing 4 `sandile_ptr` is declared. At which line will this pointer go out of scope? At which line will the pointee go out of scope? (4 marks)

v. Which will go out of scope first: `jada_ptr` or `karabo_ptr`? Explain your answer. (2 marks)

Question 4 [Total Marks: 27]

You are developing software for a store which rents out DVDs. Customers may rent more than one DVD at a time. The charge for a rental is determined by a formula that is based on the number of days that the DVD has been rented for, and the DVD's classification. Currently there are two classifications: New Release and Regular. The rental charges are as follows:

New Release DVDs A rate of R 30 per day is charged.

Regular DVDs R 20 is charged for a 1-day rental. The charge for each additional day is R 15.

Is is important to note that the classification of a DVD may change over time but this will not happen "mid-rental".

Customers earn "frequent renter points", whenever a DVD is rented. By default, one point is earned for each rental. However, an additional bonus point is earned when new releases are rented.

Your task is to model the above problem using an object-oriented approach and bearing in mind the need to create a flexible solution which can be easily extended to handle new classification types, and changes to the frequent renter point policy.

Provide all the *complete* classes and types which are necessary for producing a customer's statement based on the rentals that they have made. The statement has the form shown in Listing 5 and is stored in a `string` which is generated by the member function:

```
string Customer::getStatement() const
```

Note, you *do not* have to give the `main` function.

```
Rental statement for Jane Doe:
Furiosa: A Mad Max Saga       R 150
Super Size Me                 R 80

Total amount owed is R 230.

You earned 3 frequent renter points.
```

Listing 5: An example customer's statement. Both DVDs were rented for 5 days. Furiosa: A Mad Max Saga is classified as a new release, and Super Size Me is classified as a regular release.

[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;
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;
```

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.