Advanced Programming in C++

2013

Standard Library and related

These exercises cover different parts of the C++ standard library. Their purpose is to support you in your studies to prepare for the computer examination. There are also some larger exercises given separately, which are highly recommended. Most of the programming exercises given here are supposed to lead to rather small solutions, while most of the exercises given separately, in most cases, lead to more comprehensive coding.

The exercises given here are divided into different topics, with respect to their main feature. Some exercises may therefore actually cover several topics.

- Standard containers and algorithms
- Function objects
- Iterators
- Numerics
- Manipulators (streams) Don’t prioritize!

Standard containers and algorithms

If nothing else is specified in the exercises, use standard algorithms, function objects (standard or defined by you), iterators, etc., rather than loops (**for**, **while**, **do**), ordinary functions, indexing, etc.

vector

1. Write a program which creates a vector<**char**> which stores the letters a-z in order. Print the elements in the vector in order and in reverse order using std::copy.

2. Create a vector<string> and read a list with names of car brands into the vector. Sort the list using std::sort and print the result using std::copy. There is a given text file with car brands, named cars_of_the_world.txt.

3. Rewrite exercise 2. Instead of printing all cars using std::copy(), construct a loop for printing all car brands in the list in exercise 2, which begins with a certain letter, e.g all car brands beginning with 'A'. Try both the ordinary **for** loop, with iterators, and the range based **for** loop.

4. Rewrite exercise 2. Instead of printing all cars, print just those fulfilling a predicate. Define a binary predicate, which takes two strings and checks if the first character of the first string is equal to *any* of the characters in the second string, regardless of case. Use std::copy_if and your predicate to print all car brands beginning with, e.g. A, B or C.

5. Rewrite exercise 2. Before printing the cars, their names are to be lower-cased. Define a function object class to lower case the letters in a string. Use std::transform() and such a function object to iterate over all cars and lower-case their names.

6. Implement, using standard functions such as make_heap, pop_heap and push_heap, but not sort_heap, the sorting method Heapsort on a std::vector. Heapsort algorithm:

- the content of the vector is first ordered as a max-heap
- then, repeatedly, until the size of the heap is reduced to one, swap the first and last element in the remaining heap part of the vector – reduce the heap size by one – restore heap order.
- the values in the vector are now stored in increasing order.

7. The following code is found in a program, where the vector v is populated with int values in some way, and then sorted as shown.

```
vector<int> v;
sort(v.begin(), v.end(), greater<int>());
```

Add code to then read one integer value at a time from cin, search for the value in v with std::binary_search and print if a value was found or not.

8. This is an exercise on using a variety of standard library components. Write a program which does the following, mainly by using standard algorithms, function objects, etc., when possible:

a) Read word from a text on file and store the words in the text in a std::vector. Use std::copy to read from the file and insert into the vector.

b) Sort the words in alphabetic order, and print the sorted words.

c) Remove all duplicates, and print the remaining unique words.

d) Sort again, but this time with respect to word length, keeping the alphabetic order for words of equal length. Print the result.

e) Count the number of words which are longer than a certain length, e.g five letters. The portion of words which exceeds this word length is printed, e.g as a percentage of the total number of words in the text read (this can be measure of the complexity of a text.)

f) Remove certain words (e.g a, and, but, do, if, in, is, its, not, of, or, that, the, to). Print the words that now remains. The words to remove can be stored in, e.g., a vector.

list

1. Modify the program in exercise 8, so the words are stored in containers of type std::list instead of std::vector. This will reveal some differences concerning std::vector and std::list.

2. Write a program that reads pairs of positive integer numbers from a text file and calculates the greatest common divisor (GCD) for each pair. Each pair is written on a separate line in the input text file and you may assume that input files always contains full pairs.

Define a fully featured function object class for GCD.

Define a an ordinary function (you can make it a template though) for three column output, given an output stream, an output field width, and iterators for iterating over three input ranges of (supposed to have) equal length.

Use three lists, one list for the first numbers of the pairs, another for the second numbers of the pair, and a third for the GCDs.

map

1. Write a function make_xref(), which, given an input stream, a vector <string> with given words, and an empty map<string, vector<int>>, creates a cross-reference in the map for the given words. The input stream is supposed to be a text with words and separating white space only. When all words in the input stream have been read, the map shall, for each given word, store the lines in the input where the word was found.

Also write a function print_xref(), for printing the cross-reference map created by make_xref(). print_xref() shall take an output stream and the cross-reference map. If the given words were *algorithms*, *containers*, *functions*, and *objects*, the output from print_xref() could be as follows.

```
algorithms  2, 7, 8.
containers  1, 4, 5.
functions  9.
objects  2, 7.
```

This exercise is supposed to show how standard library components and some own coding can be combined to solve a rather complicated operation in a fairly concise way.

Modification: Replace the vector for storing line numbers with a std::set.

2. In a map cars, declared as

```
map<string, list<string>> cars;
```

the keys are supposed to be names of countries, and for each country there is a list of the car brands produced in that country. To add a car brand for at certain country, which of the following operations is most suitable?

a) cars.insert(country, car);
b) cars[country] = car;
c) cars.insert(country).insert(car);
d) cars[country].push_back(car);

set

1. Write a program which lists all distinct words in a file in alphabetic order. In this case, a word is a sequence of letters, and words are delimited by arbitrary sequences of characters that are not letters. Use std::set to store the words.

2. Overload **operator**<< to output the elements of a set, one per line. Use std::copy and a stream iterator.

Note: In exercise 3-7 standard algorithms, such as set_union, set_intersection, and other standard set operations shall not be used if they directly implement the operation in question.

3. Overload **operator**+ to compute the union of two sets A and B, $A \cup B$, i.e. the set of all elements that are members of both A and B.

Note: Use of basic `set` operations.

4. Overload **operator-** to compute B's complement relative to A, A–B, i.e. the set of all elements that are members of A but *not* of B.

Note: Use of basic `set` operations.

5. Overload **operator*** to compute the intersection of two sets A and B, $A \cap B$, i.e. the set of all elements that are members of both A and B.

Note: Use of basic `set` operations.

6. The set A is a subset of the set B, $A \subseteq B$, if each element in A is an member of B. The subset relation can also be defined in terms of intersection and equality: A is a subset of B if, and only if, the intersection of A and B, $A \cap B$, is equal to A. Use this definition to implement the subset relation $A \subseteq B$ as a function template `subset`.

Note: Easy if you have solved exercise 5.

7. Use the function template `subset` defined in exercise 6, to overload **<=** to compute the subset of two sets A and B, $A \cap B$.

Note: Simple addition to exercise 6.

multiset

1. Write a function which sorts the elements of a std::vector using a std::multiset.

stack

1. Define a queue by using only two std::stack (classic problem).

queue

1. Implement, using std::queue, the sorting method Radix Sort.

Function objects

1. Define both a *function* and a *function object class*, which returns the middle value of three values ("median-of-three"). Write a test program, which uses these, both directly and by giving them to a function, to be used from the function. Compare how this works for functions and for function objects.

2. Do the following on a container, e.g. a vector, with **int**s, using standard function objects, function adapters, algorithms, and maybe also some own function objects:

- **a)** Finds all values which are larger than a certain value.
- **b)** Finds all values which are *not* equal to certain value.
- **c)** Multiply all values with a certain factor.

Iterators

Many of the exercises for **Operator overloading** i the mixed exercises also include use of iterators.

1. Given the following **for** statement:

```
for (unsigned i = 0; i < line.size(); ++i)
  if (line[i] == ' ')
    space_count++;
```

Assume that line is a std::string. Rewrite the **for** statement using string iterators. Also rewrite using the range based **for** statement. It is also possible to count spaces using standard library components only.

2. Write a program which reads integer numbers from cin using a std::istream_iterator. The program shall print all even numbers, separated by spaces, to an output file, and all odd numbers, one per line, to another output file. Use std::ostream_iterators for printing to the output files.

Numerics

1. Write a program which reads a file of floating point numbers, creates a complex number (std::complex) of each pair read, and prints the complex numbers. If the number of floating point number in the file is odd, the last complex number shall be created with an imaginary part that is 0.0, and a warning shall be printed.

Manipulators

Only if you have time over and you want some odd exercises…

Note: Exercises 1-4 are related and supposed to be solved in order.

1. Define a simple manipulator tab, so a tab character can be written to an output stream as the example below shows. This is a simple exercise, see what basic_ostream::operator<< can offer.

```
cout << x << tab << y << tab << z << endl;
```

2. Define a parameterized manipulator tab, which can take an argument specifying how many tab characters to be printed to the output stream, according to the following example.

```
cout << tab(2) << x << endl;
```

Note 1: Parameterized manipulators need a bit more complicated solution, than non-parameterized. Do not use smanip, which the standard manipulators do, since smanip is implementation dependent, but instead implement tab straightforward as a class.

Note 2: Its not the intent that this parameterized version of tab is to work together with the unparameterized version of tab in exercise 1!

3. It is possible, with a more sophisticated solution than required to solve exercise 1 and 2, to make the manipulator tab usable both with and without an argument, e.g in accordance with the following example:

```
cout << tab(2) << x << tab << y << tab << z << endl;
```

The basic ideas how to solve this should be found in the solutions to exercise 1 and 2. Now it comes to combine this in some way to avoid the problem which should arise if you have tried to use the implementations of tab according to exercise 1 and 2 in one program (otherwise you have made a more advanced solution in these exercises than actually demanded).

4. This exercise incorporate the use of several constructs and techniques; class template, functions, overloading, callback. Generalize the solution for the manipulator tab you have acquired in exercise 3 above, so one can implement other manipulators which takes one argument, without having to copy and modify all code for the manipulator tab. If you do this, one could, e.g., without much effort make a parametrized variant of the manipulator endl:

```
cout << tab(2) << x << tab << y << tab << z << endl(2);
```

Note: Besides the manipulation itself (in the case of tab, to print a number of tab characters) it can, in the general case, also be of interest to be able to vary the type of the argument to the manipulator.

5. Define an output manipulator `based` which takes two argument, a number base (e.g between 2 and 16) and a number of type **int**, and prints the number the representation specified by the base. The manipulator is to be used, e.g. in the following way (and in this case print 1011, since the base 2 is specified):

```
cout << based(2, 11) << endl;
```

6. Define an output manipulator format for printing a value to an output stream right justified in an output field of a specified width, and also with the possibility to specify a fill character (if not specified space, ’ ’, shall be default). Example of usage:

```
cout << format(’a’, 4) << endl;            // "   a"
cout << format(999, 8, ’#’) << endl;       // "#####999"
cout << format(666, 6) << endl;            // "   666"
cout << format("foo", 6) << endl;          // "   foo"
```

Note: One can generalize, in analogy with exercise 4 above, to achieve reusable code for manipulators with up to three arguments.

7. Given the declarations:

```
struct X {
  int i;
  X(int);
  X operator+(int);
}

struct Y {
  int i;
  Y(X);
  Y operator+(X);
  operator int();
}

extern X operator*(X, Y);
extern int fun(X);
```

Which type conversions are performed in the following declarations and expressions:

```
X   x{ 1 };
Y   y{ x };
int i{ 2 };

i + 10
y + 10
y + 10*y
x + y + i
x*x + i
fun(4711)
fun(y)
y + y
11147 + y
```