OOP10 MACHINE TEST

General information

1 Time 2010/6/17 6:30~10:30 pm

2 Score 4 problems; 100 points in total, excluding 5 bonus points

3 Sample test in rar or zip format

Problem	Score	Files
A	25	A.cpp
B	25	B.cpp
C	25	C.cpp, stackC.h
D	25	D.cpp, listD.h

$\Leftarrow$ 5 bonus points

4 General requirements

- Open course web site, lecture notes, homework solutions, and nothing else
- Use the stipulated algorithm, if any.
- Comments are not required.
- Suffice it to run the sample test. However, you shall present a general solution for each problem – any solution tailored for the sample test data will come to nought.
- Be honorable!
  Any activity unrelated to the test such as browsing the web, chatting on web, playing game, etc., is strictly prohibited.

Problem A

Generic function

Part A (10%)

Given

```
template<typename T>
struct even {
   bool operator()(const T& n) const { return (n&1)==0; }
};
```

Write an explicit specialization for **`T = string`** so that the function object
`even<string>()`
can be used to determine if a C++ string is of even length. For example,
`even<string>()("Snoopy")` evaluates to true (∵ the length is 6), and
`even<string>()("Pluto")` evaluates to false (∵ the length is 5).

Part B (15%)

Write the following generic function

```
template<typename ForwardIterator,typename Predicate>
ForwardIterator
satisfy(ForwardIterator begin,ForwardIterator end,Predicate p);
```

This function eliminates all the elements in the sequence **`[begin,end)`** that don't satisfy the Boolean-valued one-parameter predicate **`p`** and returns an iterator pointing to the end of the resulting sequence.
For example, let **`p`** be the function object **`even<int>()`** and apply it to the 15-element sequence

1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1

The result is the sequence that consists of the even numbers

2, 4, 6, 8, 6, 4, 2

The function does not actually erase any element from the original sequence. Physically, the resulting sequence still has 15 elements; but, logically, it has only 7 elements – the remaining 8 elements are meaningless.

Sample run

```
Test 1 ...
true false

Test 2 ...
2 4 6 8 6 4 2

Test 3 ...
Cubs Astros Mets Royals Giants Braves
```

Problem B

Substring generation

A substring of a string *s* contains a (possibly empty) sequence of characters of *s*. The characters may not be consecutive in *s*, but the order of the characters in *s* is preserved in the substring.
As an example, for *s* = "cafe", there are 16 substrings:
"" "e" "f" "fe" "a" "ae" "af" "afe" "c" "ce" "cf" "cfe" "ca" "cae" "caf" "cafe"

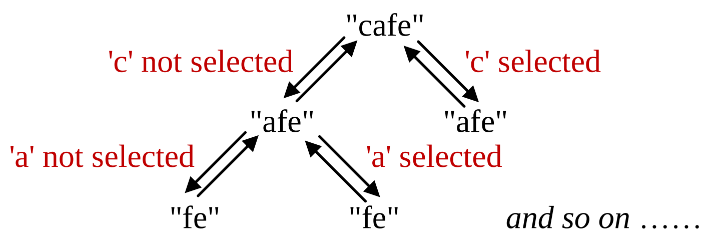
Write the following function

```
int substring(const string& s)
```

to generate all of the substrings of **`s`**. The function also returns the number of sub-strings generated.

Hints

1 You may define your own help function, if necessary.
2 Keep track of the characters (i.e. the substring) selected so far, e.g.



To this end, you may use STL stack, vector, or string to record the substrings. (N.B. Like STL list, C++ string also supports the functions **`insert`** and **`erase`**.)

Bonus (5%)

For 5 bonus points, you shall generate only *distinct* substrings.
As an example, for *s* = "sees", there are only 11 distinct substrings:
"" "s" "e" "es" "ee" "ees" "ss" "se" "ses" "see" "sees"

Sample run

```
Test 1 ...

e
f
fe
a
ae
af
afe
c
ce
cf
cfe
ca
cae
caf
cafe
16 substrings
```

an empty substring

Duplicate substrings are removed.

```
Test 2 ...

s
e
es
ee
ees
ss
se
ses
see
sees
11 substrings
```

Duplicate substrings aren’t removed.

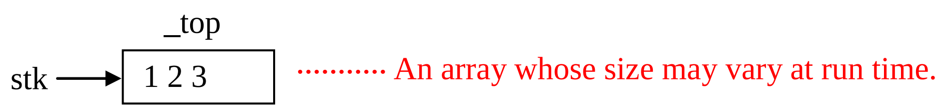
```
Test 2 ...

s
e
es
e
es
ee
ees
s
ss
se
ses
se
ses
see
sees
16 substrings
```

Problem C

Stack implementation

In this problem, you are asked to implement a stack by a dynamic array.



The size of the dynamic array shall follow the sequence 0, 1, 2, 4, 8, 16, 32, 64, …. That is, an empty stack has no storage. After pushing the 1st element onto an empty stack, the stack has an array of one element.

Afterwards, whenever an element is pushed onto a *full* stack,

1) a new array of double size is allocated,
2) the elements of the old array are copied to the new array,
3) the old array is destroyed, and
4) the element to be pushed is copied to the new array.

You need only write the blue-colored member functions, as shown below.

N.B. (1) Read other members carefully (2) Do not add any member to the class

```
template<typename T>
class stack {
public:
   typedef size_t size_type;
   typedef T value_type;
   stack();                          // 4%
   ~stack() { while (!empty()) pop(); operator delete[](stk); }
   void push(const value_type&);     // 17%
   void pop();                       // 4%
   value_type& top() { return stk[_top]; }
   const value_type& top() const { return stk[_top]; }
   size_type size() const { return _top+1; }
   bool empty() const { return size()==0; }
private:
   T* stk;
   size_type _top,_capacity;  //_capacity is the array size.
};
```

Sample run

```
Test 1 ...
0
z 26
y 25

Test 2 ...
Snoopy constructed
Pluto constructed
=======================
Snoopy copy-constructed
Snoopy copy-constructed
Snoopy destructed
Pluto copy-constructed
Pluto destructed
Snoopy destructed
=======================
Pluto destructed
Snoopy destructed
```

Problem D

List operation

Extend the template class **`list`** of HW#7 with a member function

```
list<T>& list<T>::operator-=(const list<T>&);
```

that "subtracts" a list from another in the following sense.
Let the lists a and b be

$a = (a_1, a_2, \ldots, a_m)$

$b = (b_1, b_2, \ldots, b_n)$

Then, the operation $a -= b$

1 erases all elements of list a that occur in list b
i.e. if $a_i = b_j$ for some j, erase a_i from list a

2 inserts all elements of list b (from b_1 to b_n, in that order) that are not in list a to the end of list a
i.e. if $b_i \neq a_j$ for all j, insert b_i at the end of list a

For example, let

$a = (6, 5, 6, 7, 8, 8, 3)$,

$b = (1, 2, 6, 8, 1, 3, 2, 1, 3)$,

then, after the execution of a **-=** b, list a becomes

$a = (5, 7, 1, 2, 1, 2, 1)$

because the blue-colored numbers are erased from list a, and the red-colored numbers are inserted at the end of list a.
Also, list b remains unchanged.
Notice that a self-application erases the entire list, i.e. for any list a, the execution of a **-=** a empties list a

$a = ()$

Sample run

```
5712121
683357
121216833
5712121
```