OOP Final Solution

1 a) See lecture note on ADT

b)
```
for (list<list<int> >::iterator it1=a.begin();it1!=a.end();++it1) {
        for (list<int>::iterator it2=it1->begin();it2!=it1->end();++it2)
                cout << *it2;
        cout << endl;
}
```

2 a~e) See lecture note on ADT

f) There are two errors.

1 it<last should be replaced by it!=last, for input iterators don't support operator<.

2 it+=1 should be replaced by ++it or it++, for input iterators don't support operator+=.

3 a) There are four conditions:

(1) The ctor X::X(int) is non-explicit.

(2) The copy ctor is const-qualified, i.e. X::X(const X&);

(3) The copy ctor is non-explicit.

(4) The copy ctor is accessible (i.e. not private).

Comment

Under condition (1),

X a=7;

will be compiled to

X a=X(7);   // X(7) is an implicit conversion generated by the compiler.

which in turn requires that conditions (2) , (3), and (4) be satisfied by the copy ctor.   (N.B. VC++ fails to detect it. Try GNU C++.)

b) Method A

Define an explicit copy ctor.

You may define it publicly or privately, i.e.

class X { public: explicit X(const X&) {} };

or

class X { private: explicit X(const X&) {} };

This method disallows passing an X object by value. But, it still allows

X a(b);   where b is an X object

everywhere in the program (if it is defined publicly) or within the class scope (if it is defined privately).

Method B

Declare a copy ctor and provide no definition for it.

You may declare it publicly or privately, i.e.

class X { public: X(const X&); };

or

class X { private: X(const X&); };

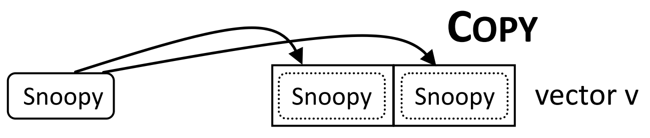
This method not only disallows passing an X object by value, but also disallows direct initialization:

X a(b);   where b is an X object.

4

```
Snoopy constructed              // 1
Snoopy copy-constructed         // 2
Snoopy copy-constructed         // 3
Snoopy destructed               // 4
Snoopy destructed               // 5
Snoopy destructed               // 6
```

Line	**Comment**
1	"Snoopy" is converted to a temporary str object by calling the str's ctor so as to invoke the vector<str>'s ctor, i.e. the declaration vector<str> v(2,"Snoopy"); becomes vector<str> v(2,**str("Snoopy")**)
2,3	Within the vector<str>'s ctor, the temporary str object referenced to by val is copy-construted to v[0] and v[1], respectively, i.e. for (**str*** it=start;it!=finish;++it) new (it) **str**(val); // call str's copy ctor



temporary str object   v[0]   v[1]

4	After completely constructing the vector v, the temporary str object is destroyed by a call to str's dtor.
5,6	On returning from main, the vector v is destroyed by the call v.~vector<str>() to the vector<str>'s dtor. In turn, within the vector<str>'s dtor, the str objects v[0] and v[1] are destroyed by calling the str's dtor: for (**str*** it=start;it!=finish;++it) it->~**str**(); // call str's dtor

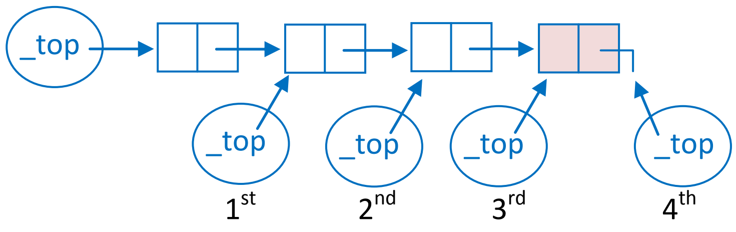
5 a) See lecture note on ADT

b) After the execution of the recursive call

stack(_top->succ).size()

the temporary stack object created by stack(_top->succ) will be destroyed, erasing the sublist of the list being traversed.

This is a disaster – the traversal will crash as soon as it tries to destroy a temporary stack object whose substack has already been destroyed.



temporary stack objects

When the recursion unwinds, the temporary stack objects are destroyed in the reverse order of their construction. Thus, the traversing process will crash on destroying the 2nd temporary stack obect, since the pink-colored node has already been deleted after destroying the 3rd temporary stack object.

Method 1

The easiest way to avoid the undesired erasion is to nullify the stack object after the execution of the recusrive call.

```
stack::size_type stack::size() const
{
    if (_top==NULL) return 0;
    else {
        stack sub(_top->succ);
        size_type sz=1+sub.size();
        sub._top=NULL;        // nullify the stack object
        return sz;
    }
}
```

Method 2

Alternatively, we may distinguish two kinds of stack objects – those created by public ctors are erasable, and those created by the private ctor aren't. To do so, we have to add a private data member

```
bool erasable;
```

to the stack class, and modify the ctors and dtor as follows:

```
stack::stack() : _top(NULL), erasable(true) {}
stack::stack(node* t) : _top(t), erasable(false) {}
stack::~stack() { if (erasable) while (!empty()) pop(); }
```

Finally, the definition of size() remains unchanged.

Comment

Both methods are inefficient, as they take O(*n*) extra time and space to create and destroy the temporary stack objects, where *n* is the number of elements in the stack.

Best solution

Add a private member function to the stack class:

```
stack::size_type stack::sz(node* t) const
{
    return t==NULL? 0: sz(t->succ)+1;
}
```

and then define

```
stack::size_type stack::size() const { return sz(_top); }
```

6 a) See lecture note on ADT

If T1 is a built-in type,

```
pair(){}
```

will not initialize first, but

```
pair() : first(),second() {}
```

will zero-initialize first.

Similarly, if T2 is a built-in type, they are also different.

b) See lecture note on ADT

7 a) See lecture note on ADT

b) Modify the code by replacing the test

this!=&rhs

by

!(*this==rhs)

or

this!=&rhs || !(*this==rhs)   // This is better.

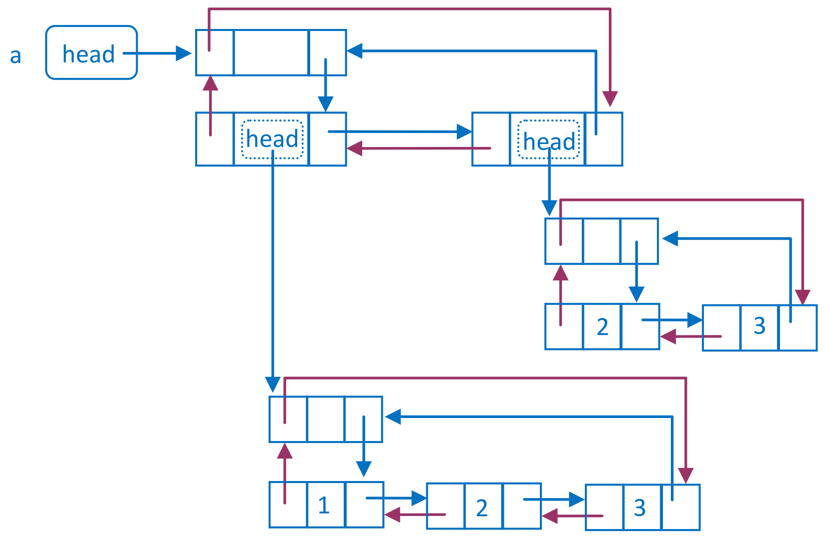
Note

If we also define

bool operator!=(const string& lhs,const string& rhs) { return !(lhs==rhs); }

we may use the test *this!=rhs instead.

8 a)



b) (++a.begin())->insert(++(++a.begin())->begin(),7);

Note:

$$\underbrace{\texttt{(++a.begin())}}\texttt{->insert(}\overbrace{\texttt{++}\overbrace{\underbrace{\texttt{(++a.begin())}}\texttt{->begin()}}^{\text{an iterator points to y1}}}^{\text{an iterator points to y2}}\texttt{,7);}$$

an iterator points to the 2nd int list (y1, y2, ...)

9 a) See lecture note on OOP

N.B. As far as this class lattice is concerned, both (1) and (3) may be private or protected.

b) See lecture note on OOP

10 a)
```
stackPS(const stackPS& rhs)
: stack(rhs),stackP(rhs),stackS(rhs)
{}
```

b) First of all, the expression **`new stackPS`** invokes **`stackPS`**'s default ctor

```
stackPS::stackPS() : stack(),stackP(),stackS() {}
```

whose execution relies on three upcasts:

1 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stack`**'s ctor

2 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stackP`**'s ctor

3 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stackS`**'s ctor

Next, three more casts for the execution of the remaining code:

`stackPS` $\rightarrow$ **`stackP&`** in **`stackP& s=*new stackPS;`**

`stackP` $\rightarrow$ **`*stackS`** in **`s.push(2);`**

`stackP` $\rightarrow$ **`*stack`** in **`s.stackP::push(2);`**

11 We have to define our own copy ctor.

```
stack::stack(const stack& rhs)
: _top(stk)
{
   for (int* it=rhs.stk;it<rhs._top;++it) {
      *_top=*it; _top++;
   }
}
```