OOP Final solution

1 a) The STL **queue** class doesn't support iterator at all.

b) Since **d** references to a const deque, **it** shall be of the type
deque<int>::const_iterator
rather than
const deque <int>::iterator

c) The STL **vector** doesn't support **push_front**.

d) The auto_ptr **p** no longer owns the storage that contains a 7.

2 ① not a ctor at all
② a copy ctor
③ a ctor but not a copy ctor
④ a copy ctor

3 First of all, the code
string s="snoopy";
uses copy initialization. Thus, **"snoopy"** has to be converted to a temporary string object.
Now, the ctor in line 1 is illegal, because it can't reference to a temporary object.
Also, the ctor in line 3 is illegal, because it can't be used in copy initialization.

4 A uninitialized object can't be assigned.
The assignment **_data[i]=val** should be replaced by the placement new
new (_data+i) T(val)

5 a)
```
int* list::iterator::operator->() const
{
    return &operator*();     // or, &**this
}
```

b)
```
const list::iterator list::iterator::operator++(int)
{
    iterator old=*this;      // or, iterator old(*this);
    ++*this;                 // or, operator++();
    return old;
}
```

c) Because, to be consistent with built-in types,
`it++++;` (where **`it`** is of the class **`list::iterator`**)
should be disallowed.
On the other hand,
`++a.begin()` (where **`a`** is of the class **`list`**)
doesn't conflict with built-in type semantics and is sometimes useful, e.g.
`*++a.begin()` // access the 2nd element of the list **`a`**

6 a) **`it=it+1`**
is incorrect, as input iterators don't support **`+`**.
Replace it by **`++it`**or, less efficiently, by **`it++`**

b) 1) runs faster.
`list<int>::iterator::operator*()`, which returns a reference to the datum currently pointed to by the iterator, is two times faster than **`list<int>::reverse_iterator::operator*()`**, which returns a reference to the datum preceding to that pointed to by the reverse iterator.

7 a)
```
string operator+(const string& lhs,const string& rhs)
{
    string s(lhs); s+=rhs; return s;
    // or simply, return string(lhs)+=rhs;
}
```

b) It is ambiguous – all of the 3 overloading operator functions are viable, but none is the best viable.
To make it work, at least one of the two C-style strings must be converted to a **`string`** object, e.g.
`operator+(string("snoopy"),"pluto")`

8
```
template<typename T1,typename T2>
template<typename U1,typename U2>
pair<T1,T2>::pair(const pair<U1,U2>& p)
: first(p.first),second(p.second)
{}
```

9 (1) implementation
(2) interface and *mandatory* implementation
(3) interface and [auto] *default* implementation // auto may be omitted
(4) interface

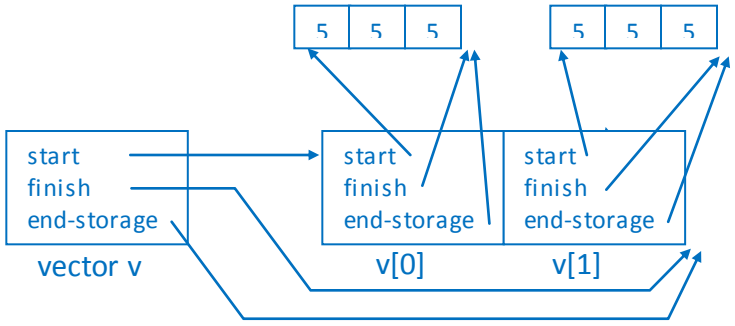
10 a) (1) protected
(2) public
(3) private
(4) public

b) For both classes, the special member functions, including default ctor, copy ctor, copy assignment operator, and dtor, shall all be protected.

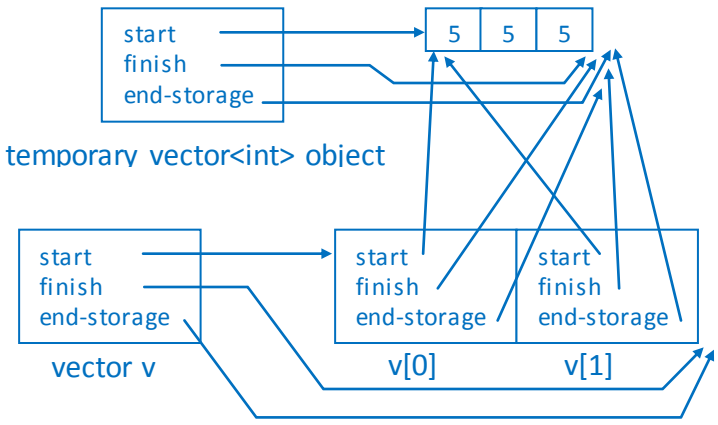
11 a)
```
stackPS::stackPS()
: stack(),stackP(),stackS()
{}
```

b) Four casts occur in the following order:
1 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stack`**'s ctor
2 upcast *this* pointer of **`stack`**'s ctor to *this* pointer of **`deque`**'s ctor
3 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stackP`**'s ctor
4 upcast *this* pointer of **`stackPS`**'s ctor to *this* pointer of **`stackS`**'s ctor

12 a)



b) The ctor call
```
vector<int>(3,5)
```
creates a temporary **`vector<int>`** object, which is then copy-constructed to **`v[0]`** and **`v[1]`** to yield the vector **`v`**. At this moment, if we use the implicitly generated copy ctor, the snapshot of the underlying structures of the vector **`v`** and the temporary object looks like:



After that, the temporary vector<int> object will be destroyed. Since **`v[0]`** and **`v[1]`** share the same storage as the temporary **`vector<int>`** object, they will also be accidently erased.

c) (1) **`rit->begin()`**
(2) **`rit->end()`**

13 The former yields the sequence 3, 4, 5, 6, 7, 8, 9.
But, the latter yields the sequence 3, 4, 5, 6, 7, 8, ?, ?, ?, 9 with three unoccupied slots.
It would be better to make use of the unoccupied slots, say
`*remove(b.begin(),b.end(),2)=9;`

14
```
class X {
public:
    X() : x(new int),single(true) {}
    X(int n) : x(new int[n]),single(false) {}
    ~X() { if (single) delete x; else delete [] x; }
private:
    int* x;
    bool single;
};
```