STL Part2 QUIZ

1.

We learnt that the STL `list` template class does not provide random access, i.e., it does not overload the subscript `operator []` like the STL `vector` class does. Why do you think that is the case, why could they not have overridden the subscript operator and make it take an integer index of the element one wants to access… in other words, why is the following not supported on a list:

```
std::list<int>  intList;

// add 150 elements to the list.
// Now access the 100th element

int value = intList[ 100 ];
// compile error, since [] is not overloaded
```

1. We learnt that the STL list template class does not provide random access, i.e., it does not overload the subscript operator [] like the STL vector class does. Why do you think that is the case, why could they not have overridden the subscript operator and make it take an integer index of the element one wants to access… in other words, why is the following not supported on a list:

```
std::list<int>  intList;
  // add 150 elements to the list.

  // Now access the 100th element

 int value = intList[ 100 ];
// compile error, since [] is not overloaded
```

Since list contains node that are linked to each other via a pointer (like in a list data structure), the nodes are not necessarily in contiguous memory, which means that to go from, say, the 1st node to, say, the 100th node, you have to traverse the pointer link from one node to the next (in other words, you cannot simply increment the pointer address like you can for an array or STL vector whose elements/nodes are allocated in contiguous memory). Since you have to traverse each element to get to the 100th node, or the Nth node in general, it means that the speed of access of a node is *dependent* on how far it is from the first node. However, the STL standard says that the subscript operator has to provide a constant complexity operation for access to any member. Since this is not possible in a list (for a reasonable implementation), the subscript operator is not overloaded for the STL list class.

2.

Why do you think that the `deque` class in STL does not provide the `reserve` or `capacity` methods, whereas the `vector` class does?

2. Why do you think that the deque class in STL does not provide the reserve or capacity methods, whereas the vector class does?

   When a deque object has to grow (i.e., when you add more elements than can fit in the space currently allocated for the deque), unlike a vector, it does not need to copy the existing elements over to the newly allocated space… think of like it has a bunch of "chunks of memory", and hence, it just creates a new chunk when more memory is needed.

   Let me make an attempt at an analogy here… if at a school, say grade 4 fills up with 25 students, and now they have another 20 students, they start another class (grade 4) where they put the overflow students (which is what a deque does)… they don't get a bigger room and put all 45 students in there (which is what a vector does… I can see teachers and parents hating vectors ☺).

   Since copying of existing elements into newly allocated memory is not what a deque class does, there isn't much reason for it to provide a reserve method, as the user does not have to worry about paying for copying of existing elements when the deque grows.

3.
What is wrong with the following code:

```
typedef map<int, MyClass *> TMyClassMap;

void PopulateMap(TMyClassMap & myMap )
{
    MyClass * mPtr = new MyClass (some args);
    myMap.insert (std::make_pair (mPtr->GetID (),
mPtr));

    MyClass m2 (some args);
    myMap.insert (std::make_pair (m2.GetID (),
&m2));
}

int main()
{
    TMyClassMap  myMap;

    PopulateMap( myMap );
    PrintAllElements( myMap );
    // assume this function prints all

    return 0;
}
```

3. What is wrong with the following code:

```
typedef map<int, MyClass *> TMyClassMap;

void PopulateMap(TMyClassMap & myMap )
{
    MyClass * mPtr = new MyClass (some args);
    myMap.insert (std::make_pair (mPtr->GetID (),
mPtr));

    MyClass m2 (some args);
    //m2 is on stack, will be destroyed
    //at end of this function when
    //it goes out of scope.

    myMap.insert(std::make_pair(m2.GetID (), &m2));
    // Oops.
    // We added its address, and its going
    // go out of scope and destroyed soon.
}
```

```
int main()
{
    TMyClassMap myMap;
    PopulateMap( myMap );
    // Poor and unsuspecting myMap doesn’t know
    // its holding a pointer to a deleted object…
its going to find out when someone accesses it, but
alas, will be too late then ☺

    PrintAllElements( myMap );
    // assume this function prints all

    return 0;
}
```

Notice that element m2 is created on the stack (and not the heap). So it is going to be deleted when it goes out of scope, which is at end of function PopulateMap…. But we added it to the map (by address), and when PopulateMap returns to main, the myMap object now contains a pointer to memory that has been deleted, and if you try to access that element, you will have undefined behavior (typically a program crash)…. You should try this out in your IDE.

.

4.

Why is the following not supported on STL list, but is supported on STL vector?

```
std::list< double > ld;

std::list<double>::const_iterator citer2 =
ld.begin();

citer2 = citer2 + 1; // this is a compile error
```

4.

Why is the following not supported on STL list, but is supported on STL vector?

```
std::list< double > ld;

std::list<double>::const_iterator citer2 =
ld.begin();

citer2 = citer2 + 1; // this is a compile error
```

Remember that we talked about the subscript operator not being overloaded on list class. For the same reason, the above is not supported on a vector. In other words, the + operator on the list iterator is not supported since if it was, you could say citer2 + 10 above and jump by 10 elements, in effect the same has having a subscript operator defined… but since that is not a constant time operation on lists, this is not allowed. You can use increment and decrement (like ++ citer2 or citer2 ++ or – citer2 or citer2--).

If you are thinking, well, citer2 + 1 is also moving only by 1 element, the problem is that if they overloaded the operator + here, then you could say + 5 instead of just +1, or +n in general, and this could not be caught at compile time… hence the + operator is simply not overloaded.

5.

What problem do you see in the code below?

```
void Foo()

{
    Base            bArray[ 2 ];
    Derived         dArray[ 2 ];

    Foo( bArray, sizeof(bArray)/sizeof(Base));

    Foo( dArray, sizeof(dArray)/sizeof(Derived));
}


void Foo( Base * bPtr, int numElements )
{
    int count = 0;

    while ( count < numElements )
    {
        ProcessObject( bPtr );

        bPtr ++;

        ++ count;
    }
}
```

Note: We will also run this example in the debugger.

6. What problem do you see in the code below

Header file: Derived.h

```
class Derived : public Base1, public Base2
{
public:
    Derived() {}

};
```

Header file: Composer.h

```
class Derived;  // forward declaration of Derived class
class Composer
{
public:
    Composer();

    const Base1 *   GetBase1() const
    {
        return (Base1*) mDerived;
    }

    const Base2 *   GetBase2() const
    {
        return (Base2*) mDerived;
    }

private:
    Derived *       mDerived;
};
```