The Standard Template Library (STL), List, Vector, and String

A container is an object that is designed to hold other objects. Examples are

- lists,
- vectors,
- maps,
- hashmaps.

A string is also a kind of container.

Doubly Linked Lists

- Syntax: `std::list<X>` . X is the type of elements in the list.
- Copy constructor, assignment, are defined having object semantics.
- Move operators are defined.
- Constructor with initializer list is defined.
- Default constructor: Construct empty list.
- No `<<` defined. (If you try to print a list, you will see a pretty unpleasant error message)

Linked Lists

```
std::list< unsigned int > l;

l = { 1, 2, 3, 4, 5 };

std::list< unsigned int > l2 = l;
   // List has object semantics, which means that the
   // list is copied.
```

Linked Lists (2)

1. Elements can be efficiently inserted/deleted (independent of size of list, linear in size of object) everywhere in the list. (At the beginning, at the end, in the middle)
2. Elements can be moved (within a single list, or between different lists of the same type) in constant time, by pointer manipulation, even when the object cannot be copied.
3. Elements can be accessed in constant time through iterators, but through indices only in linear time.
4. When the objects are small, list is space inefficient.
5. Due to random placement in memory, accessing list elements may cause many cash misses.

Back/Front

```
X& front( );
const X& front( ) const;
   // First element in list.

X& back( );
const X& back( ) const;
   // Last element in list.

pop_front( );
pop_back( );
   // Remove first/last element from list.
```

Back/Front (2)

```
void push_front( const X& );
   // Insert X at front.

void push_back( const X& );
   // Insert X at end.
```

Iterators

I want to explain what is an iterator without mentioning the word 'pointer':

An iterator is pretty much the same as an index. The two differences are:

- The element can be obtained without mentioning the container.
- One cannot do calculations on iterators. Iterators can be only compared, they can be increased/descreased.

Iterator versus Index

```
// Indexed version:
for( unsigned int i = 0; i != s. size( ); ++ i )
{
   std::cout << s[i] << "\n";
}

// Iterator version:
for( std::list< int > :: const_iterator
          p = s. begin( ); p != s. end( ); ++ p )
{
   std::cout << *p << "\n";
}
```

(Note that `std::list` cannot be indexed.)

Iterators

```
// Index:
unsigned int i = s. size( );
while( i != 0 )
{
   -- i;
   std::cout << s[i] << "\n";
}
// Iterator:
unsigned p = s. end( );
while( p != s. begin( ))
{
   -- p;
   std::cout << *p << "\n";
}
```

Pretty Printing with Index

```
std::cout << "{";
for( unsigned int i = 0; i != s. size( ); ++ i )
{
   if( i != 0 )
      std::cout << ",";
   std::cout << " " << s[i];
}
std::cout << "}";
```

Pretty Printing with Iterator

```
std::cout << "{";
for( std::list<int> :: const_iterator p = s. begin( );
        p != s. end( );
        ++ p )
{
   if( p != s. begin( ))
      std::cout << ",";
   std::cout << " " << *p;
}
std::cout << " }";
```

Non-const Iterators

If you want to assign to contents of the iterator, use `iterator` (without `const_`).

```
for( std::list<int> :: iterator p = s. begin( );
          p != s. end( );
          ++ p )
{
    *p = 0;
}
```

Auto

If you don't like the long names of iterator types, use `auto`.

```
for( auto p = s. begin( ); p != s. end( ); ++ p )
   std::cout << *p;
```

Problem: Overloading rules will prefer the most general version, i.e. `begin( )` where possible, `begin( ) const` where necessary. If you want `const_iterator`, where `iterator` is possible, use

```
for( auto p = s. cbegin( ); p != s. cend( ); ++ p )
   std::cout << *p'
```

You should use this, in order to make clear that container will not be changed.

```
std::cout << "how many numbers do you want to add ?";
unsigned int nr;
std::cin >> nr;
for( unsigned int i = 0; i < nr; ++ i )
{
   std::cout << "please type " << i;
   std::cout << "-th number: ";

   list. push_back(0);
   std::cin >> list. back( );
}

// Numbers can be added with code on previous page.
```

Erasing and Inserting in the Middle

```
iterator l.  erase( iterator p );
   // Delete the element at p, and return the
   // iterator behind p.
```

and inserted by:

```
iterator l. insert( iterator p, x );
   // Insert x at position p, and return the new
   // iterator, which now holds x.
```

Vectors

Vectors and lists are quite similar things. A list is implemented by a chain of cells that are connected with pointers. A vector is implemented by an array that is allocated on the heap.

- Vectors allow indexing, using either [ ] or `at( )`.
- Vectors are more space efficient. They are likely to be local which can be an advantage for small elements. They may be harder to allocate.
- Vectors do not support inserting/erasing in the middle. (But overwriting an element is possible.) Using `push_back( )` or `pop_back()` on a vector spoils all existing iterators of this vector. Be careful with that.

Implementation

The implementation of `std::vector` is pretty much like in the stack class

```
template< class X, class A = std::allocator<X>>
class vector
{
   X* tab;
   unsigned int current_size;
   unsigned int current_capacity;
};
```

`std::vector` uses low level tricks to make sure that the area in the range `X[ current_size ... current_capcity ]` is not initialized (not constructed).

The vector resizes when `current_size == current_capacity`.

Resizing

`current_capacity` is usually increased/decreased in powers of two.

It is possible to set `current_capacity` by yourself using `reserve( )`. Do this if you know in advance how big the vector will be.

If one sets the borders for resizing properly, resizing is not too expensive.

- When `current_capacity` is increased, it is doubled.
- When `current_size` $<$ `current_capacity`/4, then `current_capacity` is divided by 2.

Potential Method (Amortized Complexity)

Assign to each occurrence of vector v a potential **phi** of type **int**. Whenever we do a `push_back` or `pop_back`, proceed as follows:

- Announce `push_back( )` with a cost of 3. If we are not resizing, then 1 is used for the immediate push back, 2 is added to **phi**.

  When resizing, we have **phi** $\geq$ `current_size` which we can use for moving/copying the objects.

- Announce `pop_back()` with a cost of 2. If we are not resizing, then 1 is used for the immediate pop back, 1 is added to **phi**. When resizing, we have **phi** $\geq$ `current_size`, which can be used for moving/copying the objects.

It can be shown that **phi** is never negative. Hence a vector can `push_back( )/pop_back( )` in constant time.

Moving?

When the `std::vector<X>` resizes, it could use a moving constructor.

It will do this only when `X( X&& )` exists, and is guaranteed not to throw any exceptions.

In order to do this, declare it as

```
X( X&& ) noexcept;
```

Of course, the moving constructor should really throw no exceptions. This is possible in general when the moving constructor does not allocate on the heap.

One can use `std::is_nothrow_move_constructible<X>::value` to check it. (Or put a print statement in the moving constructor, and see if it is used.)

Strings

Strings should always be preferred over character arrays. A string is almost the same as `std::vector<char>`

It has indexing, and iterators with comparison.

The iterators are obtained as follows:

```
std::string::iterator std::string begin( );
std::string::const_iterator std::string begin( ) const;

std::string::iterator std::string end( );
std::string::const_iterator std::string end( ) const;
```

In addition to this, strings have the operators `+`, `+=`, `==`, `!=`, `<`,`>`, `<=`, `>=` defined.

Strings

```
std::string s = "one two three";
   // Converts const char* to std::string.

s += ' ';
s += "four five six";
std::cout << s;
for( std::string::const_iterator
        p = s. begin( );
        p != s. end( );
        ++ p )
{
   std::cout << *s;
}
```

Strings

Strings should of course be preferrred over C-style character arrays.