The Standard Template Library (STL)

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.

Linked Lists

- Syntax: `std::list< X >` . X is the type of elements in the list.
- Copy constructor, assignment, destructor are defined.
- Default constructor: Construct empty list.
- No `<<` defined. (If you try to print a list, you will see a pretty big error message)

Linked Lists

```
std::list< unsigned int > l;
   // This is the same as
   // std::list< unsigned int > l =
   //          std::list< unsigned int > :: list( );

std::list< unsigned int > l2 = l;
   // This is an implicit call of the
   // copy constructor. It is the same as
   // std::list< unsigned int > l2 =
   //    std::list< unsigned int > :: list( l2 );

   // A disadvantage of templates is that the
   // names are long.
```

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 one list, or between different lists of the same type) in constant time. (independent of size of list and size of object)
3. Elements can be accessed in constant time through iterators, but through indices only in linear time.
4. If you modify a vector, all its iterators become corrupted.

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 end.

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

Iterators

Iterators can be viewed as some kind of pointers to the elements of the container. The difference with ordinary pointers is that:

- They have more restricted type, and more restricted operations. This makes iterators more safe than pointers.
- They can be used without knowing how the container is implemented. This makes usage of iterators independent of the actual container used.

For the rest, iterators are pretty much like pointers.

Iterators

- There are two types of iterators: `iterator,` and `const_iterator`. An `iterator` can modify the object that it points to, a `const_iterator` cannot.
- `*p` produces the object that `p` points to. For an `iterator`, the result has type `X&`. For a `const_iterator`, the result has type `const X&`.
- `->` can be used in case `p` points to a structure.
- `==` and `!=` compare two iterators. Vector iterators also have `<,>,<=,>=`.
- `++,--` increase/decrease an iterator. Vector iterators also have `+,-,+=,-=`.

Iterators belonging to List

```
std::list<X> :: const_iterator
std::list<X> :: begin( ) const;
   // const_iterator pointing to first element,
   // if it is not equal to end().

std::list<X> :: iterator
std::list<X> :: begin( ) const;
   // iterator pointing to first element,
   // if it is not equal to end( ).
```

```
std::list<X> :: const_iterator
std::list<X> :: end( ) const;
   // const_iterator pointing one beyond
   // last element.

std::list<X> :: iterator
std::list<X> :: end( );
   // iterator pointing one beyond
   // last element.
```

Usage of Iterators

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

// Now sum contains the sum. This program also
// works for the empty list. In the empty list,
// begin( ) == end( ).
```

```
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

Elements can be erased by:

```
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 have random access, by using [] or at.
- Vectors are somewhat more space efficient. They are likely to be local which can be an advantage for small elements. They may be harder to allocate.

Vectors

Vectors offer the following operations:

```
X& operator [ ] ( unsigned int );
const X& operator [ ] ( unsigned int ) const;

X& at ( unsigned int );
const X& at ( unsigned int ) const;
   // The same as [ ] but at checks range.
```

Reverse Iterators

If you want to go through a list in reverse order, you could do this in the following way:

```
std::list< unsigned int > :: const_iterator
   p = list. end( );
while( p != list. begin( ))
{
   -- p;
   std::cout << *p;
}
```

Reverse Iterators (2)

Another, more intuitive way is to use a reverse iterator:

```
for( std::list< unsigned int > ::
                        const_reverse_iterator
        p = list. rbegin( );
        p != list. rend( );
        ++ p )
{
   std::cout << *p;
}
```

Maps

Maps are an associative container. They are useful for efficient lookup of elements. Maps are implemented by a binary search tree.

They are nice and useful, but there are some unpleasant complications in their use.

Maps

A map has an index type and a target type.

```
std::map< std::string, unsigned int > mp;
   // Creates map from std::string to unsigned int.

std::map< unsigned int, unsigned int > mp;
   // Map from unsigned int to unsigned int.

std::map< std::string, std::string > mp;
   // Map from string to string.
```

Map Iterators

Map iterators are pretty much like list and vector iterators, but the map contains pairs of elements.

```
std::map< std::string, unsigned int > mp;

... insert some elements.

std::map< std::string, unsigned int > :: const_iterator
     p = mp. begin( );

// Now *p has type
// std::pair< std::string, unsigned int >.
```

Pairs

Everything you need to know about pairs is on this slide:

```
// pair<X,Y> has two elements:

X first;
Y second;

// And it has a constructor: (Needed later)

std::pair<X,Y>::pair( const X& x, const Y& y );

... and a copy constructor, assignment, etc ...
```

That's all.

Now we know how to print the elements in a map:

```
// We print the elements in an
// std::map< std::string, unsigned int > :

for( std::map< std::string, unsigned int >
        :: const_iterator
         p = mp. begin( );
         p != mp. end( );
         ++ p )
{
   std::cout << ( p -> first );
   std::cout << " -> ";
   std::cout << ( p -> second );
   std::cout << "\n";
}
```

Lookup and Insertion

Unfortunately, lookup and insertion are not as nice as they ought to be. We start with lookup:

```
std::map<X,Y> mp;

X x;

// Now mp[x] is of type Y&.
// One can for example do

std::cout << mp [x] << "\n";
```

This probably looks fine, but ...

Problems with [ ]

When `x` is not defined in `mp`, then `mp[x]` will create an element, using the default constructor of `Y`.

This has two very unpleasant consequences:

1. Code involving [ ] will refuse to compile if Y has no default constructor.
2. `mp[x]` refuses to compile if `mp` is const.

As a consequence, the indexing operator [ ] can be only used in assignments, where type Y is small and has a default constructor.

On the other hand `mp[x]` is guaranteed to exist, which is sometimes useful.

Lookup using iterators

Instead of [ ], one can use

```
std::map<X,Y>:: const_iterator find( const X& ) const;
std::map<X,Y>:: iterator find( X& );
```

If X does not exist, then `find( )` returns `end( )`.

The following code compiles, also when Y has no default constructor:

```
const std::map< unsigned int, Y > & mp = ... ;
std::map< unsigned int, Y > :: const_iterator
              p = mp. find(k);
if( p != mp. end( ))
{
   std::cout << "The value for " << k << " is ";
   std::cout << ( p -> second ) << "\n";
}
else
{
   std::cout << "Unfortunately, no value for ";
   std::cout << k << " is present.\n";
}
```

Insertion using [ ]

Operator [ ] can be used for insertion, but a lot happens:

If one writes `mp[x] = y,` then

1. x is looked up. If it occurs, a reference to the current value is created. If it does not occur, a value is created using the default constructor of Y. A reference to the value is created.
2. The reference is passed to the assignment operator, which overwrites it with a copy of y.

As a consequence, one still needs a default constructor for Y, and possibly, first a value is created, which is immediately overwritten.

Therefore, [ ] should be used only when Y is a small type.

Insertion using insert( )

It is much nicer to use insert:

```
std::map<X,Y> mp;


X x;
Y y;
mp. insert( std::map<X,Y> :: pair(X,Y) );
   // Completely unproblematic.
```

Deletion can be done using
`erase( std::map<X,Y> :: iterator )`.

There is no way to delete with [ ].

Maps: The order

If you define `std::map<X,Y>`, the compiler needs to now which order to use for sorting the tree.

It will normally use $<$ on X.

Actually, the order is encoded in a third type Z. Z must have a default constructor, and any z of type Z must be applicable to elements of X.

```
std::map<X,Y,Z> mp;
// This compiles if the following code compiles:
Z z.
z(x1,x2);  // should return a bool.
```

Z must have a default constructor, and every z of type Z must be applicable on elements of X.

This is complicated, and you can look it up on page 485 of Bjarne Stroustrup, the C^{++} programming language, if you really want to understand it.

As a general rule, you should not define $<$ on X if there is no intuitive meaning.

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;
}
```

Other Containers

The STL has some more containers,
`set, multimap, priority_queue, bitset` , but the ones I discussed are the most useful.

Unfortunately, the STL has no hash map.

Conclusion

Use the STL. It is quite well-designed, and close to in terms of efficiency. If you use STL, you don't need to worry about memory management.