STL

Standard Template Library
Good reference: `http://www.sgi.com/tech/stl/`

Contain standard C++ data containers and algorithms.

Containers:
- **string** // **`basic_string<char>` stores text**
- **vector** // **random access sequence, dynamic size**
- **list** // **sequential access sequence, dynamic size**
- **deque** // **random access sequence, dynamic size**
- **set** // **unique collection of keys only**
- **map** // **unique collection of key-value pairs**
- **pair** // **collection of two values**

Adapted containers:
- **stack**
- **queue**
- **priority_queue**

More containers:
- **multiset**
- **multimap**
- **hash_set**
- **hash_multiset**
- **hash_map**
- **hash_multimap**

Using STL

```
// To get access to container-name in the library
#include <container-name>

// To declare objects of container-name
// storage-type is what you want to store in the container
// <storage-type> is the template specification

std::container-name<storage-type> variable-name-1;
std::container-name<storage-type> variable-name-2;

// OR

using std::container-name;
container-name<storage-type> variable-name-1;
container-name<storage-type> variable-name-2;

// OR

using namespace std;
container-name<storage-type> variable-name-1;
container-name<storage-type> variable-name-2;
```

Iterators

A standard way to iterate over a collection of values in some container. Iterators use the operators ++ (increment, go to next value) and * (dereference, get current value).

Iterators can be compared using operators == or !=

To declare an iterator:
```
container-type::iterator
container-type::const_iterator
container-type::reverse_iterator
container-type::const_reverse_iterator
```

To retrieve a forward iterator, use member functions:
```
// point out first position in collection
begin()
// point out position just after last
end()
```

To retrieve a reverse iterator, use member functions:
```
// point out last position in collection
rbegin()
// point out position just before first
rend()
```

To iterate constant containers you must use const_iterator. Forward and reverse iterators can not be mixed.

Some iterators support strides, like +9 positions.

Show example figure of where begin - end pint to...

Iterator examples

```
// convenient to have
typedef vector<int> array;

array mydata;
...

for (array::iterator i = mydata.begin();
     i != mydata.end(); ++i)
{
   cout << (*i) << endl;
}

// worse option, not generic
for (unsigned int i = 0;
     i != mydata.size(); ++i)
{
   cout << mydata[i] << endl;
}

// modeled after C-programmers solution
// note similarity of code inside loop
const int SIZE = 100;
int mydata[SIZE];

for (int* i = mydata;
     i < (mydata + size); ++i)
{
   cout << (*i) << endl;
}

// pointers to C-arrrays is like iterators !

// what if we change container type to list ?
```

Iterator in coming C++11 standard

```
// convenient to have
typedef vector<int> array;

array mydata;

...

for (auto i = mydata.begin();
     i != mydata.end(); ++i)
{
   cout << (*i) << endl;
}

...

for (int& d : mydata)
{
   cout << d << endl;
}
```

Egenskaper

Automatisk minneshantering!

string:
- sekvens av tecken
- stöder hopslagning med `+`, sökning med `find`

vector:
- ligger i följd i minnet
- snabb random access till alla element
- snabb insättning sist men ej först

deque:
- mycket lik vector
- snabb insättning först och sist

list:
- snabbt lägga till först och sist
- utspridda i minnet (dubbellänkad lista)
- enkel stegning framår och bakåt, men ej random access

map:
- associativ container
- lagrar `pair<key-type, value-type>`
- automatisk insättning vid []-sökning av nyckel
- normalt implementerad som träd
- nyckeln kan ej ändras, utan får först tas bort

set:
- lik map, men lagrar endast nyckel

Algorithms

#include <algorithm>

Non-mutating algorithms:
- for_each
- find // first value
- count
- mismatch
- equal
- search // first subsequence

Mutating algorithms:
- copy
- swap
- transform
- replace
- fill
- generate
- remove
- unique
- reverse
- rotate
- random_shuffle
- random_sample
- partition
- sort
- ...

All algorithms use iterators to specify work range!
Adapters like `back_inserter` can be used...

Example

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main()
{
  vector<int> array;

  copy(istream_iterator<int>(cin),
       istream_iterator<int>(),
       back_inserter(array));

  vector<int> array_copy(array.size());
  copy(array.begin(), array.end(),
       array_copy.begin());

  sort(array.begin(), array.end());

  vector<int>::iterator new_end;
  new_end = unique(array.begin(), array.end());
  array.erase(new_end, array.end());

  cout << "Array: " << endl;
  copy(array.begin(), array.end(),
       ostream_iterator<int>(cout, "\n"));

  cout << "Array copy: " << endl;
  copy(array_copy.begin(), array_copy.end(),
       ostream_iterator<int>(cout, "\n"));

  return 0;
}
```