Throughout this file I will use \( C \) to be a class name [and template resolution type for a templated type]; I will use \( T \) for the name of the templated type.

## I Iterators

- an abstraction to allow a programmer to go through the elements [in some order] of a data structure without knowing the implementation. You can think that an iterator locates a value.
- if a class \( C \) supports iterators, it defines a type \( C::\text{iterator} \) which can be used to define iterator variables.
- the class overloads =, ==, <, and > to mean the obvious things
- for an iterator variable \( x \), it overloads the operator * so that \(*x\) means “the value at location \( x \)”
- for an iterator variable \( x \), it overloads the operator ++ so that \( x++ \) changes \( x \) to be the location of the next value in the order
- the class defines a method \( C::\text{iterator \ begin()} \) to be the iterator of the begining of the order, and a method \( C::\text{iterator \ end()} \) to be the location right after the end of the order. [Yes, technically it is the location after the end.]
- with this a generic loop to look at all the elements in a variable \( x \) of a class \( C \) can be written as:
  ```cpp
  for (\( C::\text{iterator} \ i \ = \ x.\text{begin}(); \ i \ < \ x.\text{end}(); \ i++ \)) \{ ... \}
  ```
  In the body, you can use \(*i\) to see the values
- the class will also define reverse iterators:
  a type \( C::\text{reverse\_iterator} \), the overloaded operator --, methods \( \text{rbegin()} \) and \( \text{rend()} \) to let you traverse the list in the opposite order.

## II Pair

- this class takes 2 templates: \( T_1 \) for the first item in the pair and \( T_2 \) for the second
- this is really defined as a \text{struct} so its data fields are public. The data field names are \text{first} and \text{second}
- you need to: \texttt{#include <utility>}
- there is one method:
  ```cpp
  \text{pair}<T_1, \ T_2> \ \text{make\_pair} \ (\text{const} \ T_1&, \ \text{const} \ T_2&);
  ```
III Containers

- the word “container” is STL’s name for a “data structure” as an Abstract Data Type; it “contains” values

- every STL container defines types:
  
  $C::\text{iterator}$ and $C::\text{reverse\_iterator}$ as mentioned in section I above.
  
  $C::\text{value\_type}$ which is the type $T$
  
  $C::\text{reference}$ which is a reference to the type $T$

- every container defines methods:
  
  $C::C()$; the default constructor
  
  $C::C(const\ T\&);$ the default copy constructor, invoked when a variable is initialized from another variable in the same class
  
  $C::~C()$; the default destructor
  
  $C\&\ C::\text{operator=}\ (\const\ C\&);$ assignment
  
  $C::\text{iterator}\ \text{begin()}\,,\ C::\text{iterator}\ \text{end()}\,,\ C::\text{iterator}\ \text{rbegin()}\,,\ and\ C::\text{iterator}\ \text{rend()}$ as mentioned in section I above.

- every container defines all the comparison operators; for vars $C\ a,\ b$;
  
  $a == b$ means they have the same length and all the items are the same
  
  $a < b$ means that “a” and “b” agree to some point, and either “a” ends at that point or the value of “a” < the value of “b” at that point
  
  the others are defined by the usual algebraic rules

- there are several methods that look at the size of the container
  
  $C::\text{size\_type}\ C::\text{size}\ (\text{void});$ = the number of items in the container
  
  $\text{bool}\ C::\text{empty}\ (\text{void});$ = is the container empty?

III.1 Sequence Containers

- in a sequence container, the order in which you enter the values matters; they are “in sequence”

- it makes sense to talk about “just in front of” and “the items between”; there is a methods:
  
  $C::\text{iterator}\ C::\text{insert}\ (C::\text{iterator}\ \text{bef},\ C::\text{value\_type}\ \&\ \text{val});$ to insert the value “val” just in front of “bef”
  
  $C::\text{iterator}\ C::\text{insert}\ (C::\text{iterator}\ \text{bef},\ C::\text{size\_type}\ nVal,\ C::\text{value\_type}\ \&\ \text{val});$ to insert “nVal” copies of the value “val” just in front of “bef”
C::iterator C::insert (C::iterator bef, C::iterator fst, 
C::iterator lst);

[the value of “bef” is in the destination, the values of “fst” and “lst” are in the 
source] to insert the values from “fst” to “lst” into the destination just in front of
“before”

in all 3, the return value [an iterator] is the location of the new values

• there are also methods to delete values:

C::iterator C::erase (C::iterator pos); to remove the value at “pos”
C::iterator C::erase (C::iterator fst, C::iterator lst);

to remove all values from “fst” upto but not including “lst”

• there are three types of sequence containers: vector, deque, list

III.1.a vector

• A vector is a sequence container designed to allow fast insertion/deletion at the end
but slower insertion elsewhere. Think “array”.

• put into program: #include <vector>

• There are 2 method for insertion/deletion at the end:
  void vector<T>::push_back (const T&);
  void vector<T>::pop_back (void);
  [Notice that the value popped is NOT returned]

• We also overload the array reference operator: T& vector<T>::operator []
  (vector<T>::size_type i);

III.1.b deque

• A deque is a sequence container designed to allow fast insertion/deletion at both the
front and the end, but slower insertion elsewhere. Think “double-ended queue”.

• put into program: #include <deque>

• deque has all the methods in vector

• it also has methods for insertion/deletion at the front:
  void deque<T>::push_front (const T&);
  void deque<T>::pop_front (void);
  [Notice that the value popped is NOT returned]
III.1.c list

- A list is a sequence container designed to allow equally fast insertion/deletion anywhere in the container. Think “doubly-linked linked-lists”.

- put into program: `#include <list>`

- It has the same methods as vector and deque

- It has methods to allow insertion of parts of another list at a designated position:
  
  ```cpp
  void list<T>::splice (list<T>::iterator pos, list<T>& x);
  which puts all of (different) list “x” before position “pos”
  
  void list<T>::splice (list<T>::iterator pos, list<T>& x, list<T>::iterator xPos);
  which puts the element at position “xPos” in “x” in front of “pos”
  
  void list<T>::splice (list<T>::iterator pos, list<T>& x, list<T>::iterator xFst, list<T>& xLst);
  which inserts the elements of “x” starting as “xFst” upto but not including “xLst” into the list in front of “pos”
  ```

III.2 Sorted Associative Containers

- These are containers for which the order of insertion is considered immaterial. Think sets, where 2 sets are the same in spite of the order.

- the search time is designed to be logarithmic. Think binary tree.

- the items consist of a key and a value; there are types:
  
  `C::key_type` = the type of the key
  `C::value_type` = the type of the value

- there are 2 constructors:
  
  ```cpp
  C::C();
  C::C(C::iterator fst, C::iterator lst); which initializes the container with
  the elements from “fst” upto but not including “lst” in another container of the
  same class and key/value types
  ```

- there are methods to insert and erase:
  
  ```cpp
  C::iterator C::insert (C::iterator hint, C::value_type& val);
  inserts the “val” in or near “hint”
  
  C::iterator C::insert (C::iterator first, C::iterator last);
  insert the values starting at “first” upto but not including “last” from another
  container of the same class and key/value types
  ```
size_type erase (const key_type& k);
    remove all items with that key; return the number removed

void erase (iterator pos);
void erase (iterator first, iterator last);

III.2.a set

• this is a container where each item is included only once

• put into program: #include <set>

• it brings in all the features in section III.2 above

• yet another insertion method
  pair<set<T>::iterator, bool>
  set<T>::insert (const set<T>::value_type& x);
  the returned iterator locates the inserted value, the returned bool is set if the value is a new value

• there is also a related class multiset which may require #include <multiset>; in a multiset you can have repeated items; the corresponding insert does not return a pair, but just an iterator

III.2.b map

• a map is a keyed lookup of information, where the key could be any type; compare to an array, where the key is an integer. Each key may occur only once.

• you might view the relation between a country name [a string] and its capital city; give the map the country name, it gives the name of the capital

• another example might be our second project, where we used a word as a key and the value was the number of times that word appeared in a passage

• put into program: #include <map>

• a map has 2 template variables; I will use K for the key type and C for the information type

• map defines the type value_type to be the pair of the key and the information

• it brings in all the features in section III.2 above

• yet another insertion method
  pair<map<K,T>::iterator, bool>
  map<K,T>::insert (const map<K,T>::value_type& x);
  the returned iterator locates the inserted value, the returned bool is set if the value is a new value
• since we can think of this as a (generalized) array, we overload operator[]:

\[ T & \text{map}\langle K, T \rangle \text{::operator[]} \text{(const map}\langle K, T \rangle \text{::key_type&)}; \]

• N.B., the array indexing operator returns the information; the * operator returns the pair of key and information

• there is a related class multimap where a key may occur more than once

---

### III.3 Container Adapter

• these restrict some of the above containers to common special cases

• there are 2: stack, queue

#### III.3.a stack

• a stack has a template variable for the values on the stack

• officially, there is a second one: the type of the container; it defaults to deque, which is usually ok. The container must support push_back and pop_back

• put into program: #include <stack>

• we add a method bool stack\langle T \rangle\text{::empty ()}; to check if stack is empty

• we add methods

\[
\text{void stack}\langle T \rangle\text{::push (T & )};
\text{void stack}\langle T \rangle\text{::pop (void)};
\]

• we add a method to look at the top of the stack:

\[
\text{const T & stack}\langle T \rangle\text{::top (void)};
\]

#### III.3.b queue

• there is a template variable for the values. As in stack, there is also an optional second template variable for the container; it defaults to deque

• put into program: #include <queue>

• there is a method bool queue\langle T \rangle\text{::empty ()}; to check for empty queue

• there are methods to add and remove items from the queue:

\[
\text{void queue}\langle T \rangle\text{::push (T & )};
\text{void queue}\langle T \rangle\text{::pop (void)};
\]
• there are methods to inspect the 2 ends of the queue:
  \[ T & \text{queue}< T > : \text{::front} (\text{void}); \]
  \[ T & \text{queue}< T > : \text{::back} (\text{void}); \]

---

IV  Algorithms

• the STL defines a wide variety of algorithms applicable to many of the above classes

• note that these are NOT methods in any class; you do not use the dot notation with them

• most algorithms require the use of iterators; not require you to specify the typed variable itself, just an iterator to the data

• if you want to use these on your own classes, you will need to define your own iterators

• put into program: `#include <algorithm>`

IV.1  Query Algorithms

• these look for elements in a container or other properties; they do not change the container

• \[ C::\text{iterator} \text{find} (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}, T \text{val}); \]
  find the \text{val} between \text{fst} and \text{lst}

• \[ \text{void count} (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}, \text{const} T \text{val}, \text{int} & \text{n}); \]
  this counts the occurences of \text{val} between \text{fst} and \text{lst} and put the count into the variable \text{n}

• there are also generalizations of both of these:
  \[ C::\text{iterator} \text{find}_if (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}, \text{Pred} \text{pred}); \]
  \[ \text{void count}_if (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}, \text{Pred} \text{pred}, \text{int} & \text{n}); \]
  in both of these \text{pred} is a function that takes an iterator as the argument and returns a bool if the value at the iterator matches a condition

IV.2  Sort

• \[ \text{void sort} (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}); \]
  sort the list between \text{fst} and \text{lst}; 2 items that compare equal may wind up changing position

• \[ \text{void stable_sort} (C::\text{iterator} \text{fst}, C::\text{iterator} \text{lst}); \]
  sort the list between \text{fst} and \text{lst}; 2 items that compare equal are guaranteed to be in the same order as the original list
both of these have a form that lets you specify how the items are compared:

```cpp
void sort (C::iterator fst, C::iterator lst, bool comp());
void stable_sort (C::iterator fst, C::iterator lst, bool comp());
```

the function `comp` specifies how to compare 2 items

### IV.3 Binary Search

- this algorithm is “strange”; it does not tell you where the item is, just whether it is present or not
- there are 2 forms:
  ```cpp
  bool binary_search (C::iterator fst, C::iterator lst, const T& n);
  bool binary_search (C::iterator fst, C::iterator lst, const T& n, bool comp());
  ```

### IV.4 Set Functions

- these implement the standard set operations
- let $S_i = [fst_i, lst_i)$
  ```cpp
  bool includes (C::iterator fst1, C::iterator lst1, C::iterator fst2, C::iterator lst2);
  is $S_1 \supseteq S_2$ ?
  ```
- `C::iterator set_union (C::iterator fst1, C::iterator lst1, C::iterator fst2, C::iterator lst2, C::iterator result);` place $S_1 \cup S_2$ at result; the return value point to the location beyond the inserted elements
- `C::iterator set_intersection (C::iterator fst1, C::iterator lst1, C::iterator fst2, C::iterator lst2, C::iterator result);` place $S_1 \cap S_2$ at result; the return value point to the location beyond the inserted elements
- `C::iterator set_difference (C::iterator fst1, C::iterator lst1, C::iterator fst2, C::iterator lst2, C::iterator result);` place $S_1 - S_2$ (all elements in $S_1$ not in $S_2$) at result; the return value point to the location beyond the inserted elements

### IV.5 Computational

- must put in `#include <numeric>`
- in the next set, $T$ must be numeric; support $+$ and $*$
• \( T \) accumulate \((C::\text{iterator} \ F, C::\text{iterator} \ L, T \ I)\);
  return \( I + \sum_{i \in [F, L)} e_i \); add the elements between \( F \) and \( L \) and add \( I \)

• \( T \) inner_product \((C::\text{iterator} \ F1, C::\text{iterator} \ L1, C::\text{iterator} \ F2, T \ I)\);
  return \( I + \sum_{i \in [F, L)} (e_{1,i} \times e_{2,i}) \); multiply corresponding element, add the products, and add \( I \); think cross-products of vectors or the computation of a single element in the product of arrays

V Stream Iterators

• this lets a programmer step through an \texttt{iostream} object in a way that can be used in any of the STL algorithms

• there are 2 types: an \texttt{istream\_iterator\langle T \rangle} and an \texttt{ostream\_iterator\langle T \rangle}

V.1 \texttt{istream\_iterator}

• this can only be used as in input iterator; you can use it to get values; you cannot use it to write values

• the constructors:
  \texttt{istream\_iterator (istream \&);}  
  \texttt{istream\_iterator (const istream\_iterator \&);}  

• the usual methods: \*, ++ and ==

V.2 \texttt{ostream\_iterator}

• this can only be used as in output iterator; you can use it to write values; you cannot use it to read values

• the constructors:
  \texttt{ostream\_iterator (ostream \&, const char \*delim = 0);}  
  \texttt{ostream\_iterator (const ostream\_iterator \&);}  

• the usual methods: \*, ++ and ==