### The Standard Template Library (STL)

#### STL Components
- **Containers**
  - classes for objects that contain data
- **Iterators**
  - "pointers" into containers
- **Generic algorithms**
  - functions that work on different types of containers
- **Adaptors**
  - classes that "adapt" other classes
- **Allocators**
  - objects for allocating space

#### STL Vector Container
The STL vector mimics the behavior of a dynamically allocated array and also supports automatic resizing at runtime.

**vector declarations:**
- `vector<int> iVector;`
- `vector<int> jVector(100);`
- `cin >> Size; vector<int> kVector(Size);`

**vector element access:**
- `jVector[23] = 71;`
- `int temp = jVector[41];`
- `int jFront = jVector.front();`
- `int jBack = jVector.back();`

**vector reporters:**
- `cout << jVector.size();`
- `cout << jVector.capacity();`
- `cout << jVector.max_capacity();`
- `if (jVector.empty()) . . .`

#### STL Sequences Containers

**vector<T>**
- random access, varying length, constant time insert/delete at end

**deque<T>**
- random access, varying length, constant time insert/delete at either end

**list<T>**
- linear time access, varying length, constant time insert/delete anywhere in list

#### STL Iterators

- **Values in Maps**
- **Insert() Member Function**
- **Inserting in Maps and Multimaps**
- **Insert() Member Function**
- **Finding Data in a Map**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **More on Iterators**
- **Other Iterators**

#### STL Iterator Operations
- **Insertion into Vector Objects**
- **Deletion from Vector Objects**
- **Range Deletion Example**
- **Const Iterators**
- **Container Comparison**
- **Relational Comparison Example**
- **Subscripting in Maps**

#### STL Deque Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **Finding Data in Multimaps**

#### STL List Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**

#### STL Sorting Containers
- **Sorted Container**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**

#### STL Associative Containers
- **Sorted Container**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**

#### STL Deque Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **Finding Data in Multimaps**

#### STL List Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **Finding Data in Multimaps**

#### STL Sorting Containers
- **Sorted Container**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**
- **Sorted Containers**

#### STL Deque Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **Finding Data in Multimaps**

#### STL List Container
- **Map Example**
- **Finding Data in a Multimap**
- **Finding Data in Multimaps**
- **Finding Data in Multimaps**
Vector Constructors

The `vector` template provides several constructors:

```
vector<T> V;       //empty vector
vector<T> V(n, value); //vector with n copies of value
vector<T> V(n);     //vector with n copies of default for T
```

The `vector` template also provides a suitable deep copy constructor and assignment overload.

---

STL Vector Indexing

In the simplest case, a `vector` object may be used as a simple dynamically allocated array:

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < MaxCount; Count++) {
    iVector[Count] = Count;
}
```

However, the usage above provides neither runtime checking of the vector index bounds, or dynamic growth. If the loop counter exceeded the capacity of the vector object, an access violation would occur.

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
for (int Count = 0; Count < 2*MaxCount; Count++) {
    cout << iVector[Count];
}
```

Use of the `at()` member function cause an exception in the same situation.

---

STL Iterators

An `iterator` is an object that keeps track of a location within an associated STL container object, providing support for traversal (increment/decrement), dereferencing, and container bounds detection. (See Stroustrup 3.8.1 – 3.8.4)

An iterator is declared with an association to a particular container type and its implementation is both dependent upon that type and of no particular importance to the user.

Iterators are fundamental to many of the STL algorithms and are a necessary tool for making good use of the STL container library.

Each STL container type includes member functions `begin()` and `end()` which effectively specify iterator values for the first element and the "first-past-last" element.
**STL Vector Iterator Example**

```cpp
string DigitString = "4568228458720501289";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
    BigInt.push_back(DigitString.at(i) - '0');
}
vector<int> Copy;
vector<int>::iterator It = BigInt.begin();
while (It != BigInt.end()) {
    Copy.push_back(*It);
    It++;
}
```

This could also be written using a for loop, or by using the assignment operator.

**STL Iterator Operations**

```cpp
string DigitString = "4568228458720501289";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
    BigInt.push_back(DigitString.at(i) - '0');
}
vector<int>::iterator It = BigInt.begin();
int FirstElement = *It;
It++;
It = BigInt.end();
It--;
int LastElement = *It;
```

Each STL iterator provides certain facilities via a standard interface:

**Insertion into Vector Objects**

Insertion at the end of the `vector` (using `push_back()`) is most efficient.

Inserting elsewhere requires shifting data.

A `vector` object is potentially like array that can increase size. The capacity of a `vector` (at least) doubles in size if insertion is performed when `vector` is "full".

Insertion invalidates any iterators that target elements following the insertion point.

Reallocation (enlargement) invalidates any iterators that are associated with the `vector` object.

You can set the minimum size of a `vector` object `V` with `V.reserve(n)`.

**Insert() Member Function**

An element may be inserted at an arbitrary position in a `vector` by using an iterator and the `insert()` member function:

```cpp
insert(Y.begin(), Y.begin() + 1, m);
```

This is the worst case; insertion is always at the beginning of the sequence and that maximizes the amount of shifting.

There are overloads of `insert()` for inserting an arbitrary number of copies of a data value and for inserting a sequence from another `vector` object.
Deletion from Vector Objects

As with insertion, deletion requires shifting (except for the special case of the last element).

Member for deletion of last element: `V.pop_back()`

Member for deletion of specific element, given an iterator `It`: `V.erase(It)`

Invalidates iterators that target elements following the point of deletion, so

```cpp
j = V.begin();
while (j != V.end())
    V.erase(j++);
```

doesn’t work.

Member for deletion of a range of values: `V.erase(Iter1, Iter2)`

Range Deletion Example

```cpp
string DigitString = "0000028458720501289";
vector<char> BigChar;
for (int i = 0; i < DigitString.length(); i++) {
    BigChar.push_back(DigitString.at(i));
}
vector<char> Trimmed = BigChar;
vector<char>::iterator Stop = Trimmed.begin();
while (*Stop == '0') Stop++;
Trimmed.erase(Trimmed.begin(), Stop);
```

Note: be careful not to mix iterators for different objects; the results are usually not good…

Const Iterators

Constant iterator must be used when object is const – typically for parameters.

Type is defined by container class: `vector<T>::const_iterator`

```cpp
void ivecPrint(const vector<int> V, ostream& Out) {
    vector<int>::const_iterator It; // MUST be const
    for (It = V.begin(); It != V.end(); It++) {
        cout << *It;
    }
    cout << endl;
}
```

Container Comparison

Two containers of the same type are equal if:
- they have same size
- elements in corresponding positions are equal

The element type in the container must have equality operator. For other comparisons element type must have appropriate operator (<, >, …).

All containers supply a deep assignment operator. Also have `V.assign(fst, lst)` to assign a range to `v`. 
### Relational Comparison Example

```cpp
void ivecPrint(const vector<int> V, ostream& Out);
void StringToVector(vector<int>& V, string Source);

void main() {
    string s1 = "413098", s2 = "413177";
    vector<int> V1, V2;
    StringToVector(V1, s1);
    StringToVector(V2, s2);
    ivecPrint(V1, cout);
    if (V1 < V2) {
        cout << " < ";
    } else if (V1 > V2) {
        cout << " > ";
    } else {
        cout << " = ";
    }
    ivecPrint(V2, cout);
    cout << endl;
}
```

### STL Deque Container

**deque**: double-ended queue

- Provides efficient insert/delete from either end.
- Also allows insert/delete at other locations via iterators.
- Adds push_front() and pop_front() methods to those provided for vector.
- Otherwise, most methods and constructors the same as for vector.
- Requires header file `<deque>`.

### STL List Container

Essentially a doubly linked list.

- Not random access, but constant time insert and delete at current iterator position.
- Some differences in methods from vector and deque (e.g., no operator[])
- Insertions and deletions do not invalidate iterators.

### Associative Containers

- A standard array is indexed by values of a numeric type:
  - `A[0]...A[Size]`
  - dense indexing
- An associative array would be indexed by any type:
  - `A["alfred"], A["judy"]`
  - sparse indexing
- Associative data structures support direct lookup (“indexing”) via complex key values.
- The STL provides templates for a number of associative structures.
Sorted Associative Containers

The values (objects) stored in the container are maintained in sorted order with respect to a key type (e.g., a Name field in an Employee object).

The STL provides:

- `set<Key>`: collection of unique Key values
- `multiset<Key>`: possibly duplicate Keys
- `map<Key, T>`: collection of T values indexed by unique Key values
- `multimap<Key, T>`: possibly duplicate Keys

But of course the objects cannot be maintained this way unless there is some well-defined sense of ordering for such objects...

Strict Orderings

STL makes assumptions about orders in sort functions and sorted associative containers.

Logically we have a set $S$ of potential key values.

Ideally, we want a strict total ordering on $S$:

- For every $x$ in $S$, $x = x$.
- For every $x, y, z$ in $S$, if $x < y$ and $y < z$ then $x < z$.
- For every $x$ and $y$ in $S$, then precisely one of $x < y$, $y < x$, and $x = y$ is true.

Actually, can get by with a weaker notion of order:

Given a relation $R$ on $S$, define relation $E$ on $S$ by:

$x E y$ iff both $x R y$ and $y R x$ are false.

Then a relation $R$ is a strict weak ordering on $S$ if $R$ is transitive and asymmetric, and $E$ is an equivalence relation on $S$.

Example Order

```cpp
class Name {
  public:
    string LName;
    string FName;
};

class LastNameLess {
  public:
    bool operator()(const Name& N1, const Name& N2) {
      return (N1.LName < N2.LName);
    }
};
```

Using `LastNameLess`,

Zephram Alonzo < Alfred Zimbalist
Alonzo Church is equivalent to Bob Church

Notice that equivalence defined this way is not the same as `operator==`.

Special Function Objects

If there is an `operator<` for a class $T$ then you can use the special template `less<T>` (implicitly) to build order function objects.

```cpp
<functional>
```

When an ordering is required, the default STL implementation is built around the `less<T>` functional, so you don't have to do anything special...
Both `set` and `multiset` templates store key values, which must have a defined ordering.

`set` only allows distinct objects (by order) whereas `multiset` allows duplicate objects.

```cpp
set<int> iSet; // fine, built-in type has < operator
set<Employee> Payroll; // class Employee did not
                   // implement a < operator
```

However, a suitable operator can be provided:

```cpp
bool Employee::operator<(const Employee& Other) const {
    return (ID < Other.ID);
}
```

Set Example

```cpp
#include <functional>
#include <set>
#include "Employee.h"

void EmpsetPrint(const set<Employee> S, ostream& Out);
void PrintEmployee(Employee toPrint, ostream& Out);

void main() {
    Employee Ben("Ben", "Keller", "000-00-0000");
    Employee Bill("Bill", "McQuain", "111-11-1111");
    Employee Dwight("Dwight", "Barnette", "888-88-8888");

    set<Employee> S;
    S.insert(Bill);
    S.insert(Dwight);
    S.insert(Ben);

    EmpsetPrint(S, cout);
}
```

```cpp
void EmpsetPrint(const set<Employee> S, ostream& Out) {

    int Count;
    set<Employee>::const_iterator It;

    for (It = S.begin(), Count = 0; It != S.end();
     It++, Count++)
        PrintEmployee(*It, cout);
}
```

Hm…
**Multiset Example**

```cpp
void main() {
    list<char> L = list("dogs love food");
    // copy list to multiset
    multiset<char> M;
    list<char>::iterator i = L.begin();
    while (i != L.end()) M.insert(*i++);
    // copy multiset to list
    list<char> L2;
    multiset<char>::iterator k = M.begin();
    while (k != M.end()) L2.push_back(*k++);
    cmultisetPrint(M, cout);
}
```

**Set/Multiset Member Functions**

- **Insert and Erase**
  - by value:
    ```cpp
    S.erase(k); // k is a Key variable
    M.erase(k); // erase all copies of value
    ```
  - at iterator:
    ```cpp
    S.erase(i); // i an iterator
    M.erase(i); // erase only value *i
    ```

- **Accessors**
  - `find(Key)` - returns iterator to an element with given value, equals `end()` if not found
  - `lower_bound(k)` - returns iterator to first position where `k` could be inserted and maintain sorted order
  - `upper_bound(k)` - iterator is to last such position

---

**Maps and Multimaps**

Associative "arrays" indexed on a given Key type.

- **map** requires unique Keys (by def of order)
- **multimap** allows duplicate Keys

A map is somewhat like a set that holds key-value pairs, which are only ordered on the keys.

A map element can be addressed with the usual array syntax: `map1[k] = v`

However: the semantics are different!

**Values in Maps**

An element of a map is a pair of items: `pair<const Key, T>`

Once a pair has been inserted, you can only change the T value.

The pair class has public member fields `first` and `second`.

To create a pair object to insert into a map use pair constructor:

```cpp
HourlyEmployee Homer("Homer", "Simpson", "000-00-0001");

pair<const string, Employee>(Homer.getID(), Homer)
```
Inserting in Maps and Multimaps

Insert value (can also insert using iterator):

```c++
map<string, Employee> Payroll;
Payroll.insert(pair<const string, string>(Homer.getID(), Homer));
```

A multimap allows duplicate keys:

```c++
multimap<string, string> mpl;
mpl.insert(pair<const string,string>("blue", "Jenny");
mpl.insert(pair<const string,string>("blue", "John");
```

Map Example

```c++
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <functional>
#include <map>
using namespace std;
#include "Employee.h"

void EmpmapPrint(const map<const string, Employee> S, ostream& Out);
void PrintEmployee(Employee toPrint, ostream& Out);

void main() {
    Employee Ben("Ben", "Keller", "000-00-0000");
    Employee Bill("Bill", "McQuain", "111-11-1111");
    Employee Dwight("Dwight", "Barnette", "888-88-8888");
    map<const string, Employee> S;
    // . . . continues . . .
    S.insert(pair<const string, Employee>(Bill.getID(), Bill));
    S.insert(pair<const string, Employee>(Dwight.getID(), Dwight));
    S.insert(pair<const string, Employee>(Ben.getID(), Ben));
    EmpmapPrint(S, cout);
    // . . . continues . . .
}
```

Finding Data in Map

```c++
// . . . continued . . .
S.insert(pair<const string, Employee>(Bill.getID(), Bill));
S.insert(pair<const string, Employee>(Dwight.getID(), Dwight));
S.insert(pair<const string, Employee>(Ben.getID(), Ben));
EmpmapPrint(S, cout);
// . . . continues . . .
```

Map Example

```c++
map<string,string> mp;
// insert some values
map<string,string>::iterator m_i;
m_i = mp.find("222-22-2222");
if (m_i != mp.end()) // do something with entry
    // insert some values
map<string,string>::iterator m_i;
m_i = mp.find("222-22-2222");
if (m_i != mp.end()) // do something with entry
    // manipulate the data entry, but not the key value:
    (*m_i).second //data value, may be changed
```
Map Example

```cpp
// . . . continued . . .
map<const string, Employee>::const_iterator It;
It = S.find("111-11-1111");
cout << (*It).second.getName() << endl;
// . . . continues . . .
```

Bill McQuain

Finding Data in a Multimap

The `find()` method is only guaranteed to find a value with the specified key.

- `lower_bound()` method finds first pair with the specified key
- `upper_bound()` method finds last pair with the specified key

Use an iterator to look at each of duplicate values.

Subscripting in Maps

The map template allows use of a subscript:

```
map[k] = t
```

(even if the key value isn't integral).

If no pair exists in the map with the key `k`, then the pair `(k, t)` is inserted.

If pair `(k, t0)` exists, then `t0` is replaced in that pair with `t`.

If no pair with key `k` exists in `mp` the expression `mp[k]` will insert a pair `(k, T())`.

This ensures that `mp[k]` always defined.

Subscripting is not defined for multimaps.

Map Example

```cpp
// . . . continued . . .
Employee Fred("Fred", "Flintstone", "888-88-8888");
Employee Homer("Homer", "Simpson", "123-45-6789");
S[Fred.getID()] = Fred;
S[Homer.getID()] = Homer;
EmpmapPrint(S, cout);
}
```

<table>
<thead>
<tr>
<th>Phone</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000-00-0000</td>
<td>Ben Keller</td>
</tr>
<tr>
<td>111-11-1111</td>
<td>Bill McQuain</td>
</tr>
<tr>
<td>123-45-6789</td>
<td>Homer Simpson</td>
</tr>
<tr>
<td>888-88-8888</td>
<td>Fred Flintstone</td>
</tr>
</tbody>
</table>
More on Iterators

There are several kinds of iterators, which correspond to various assumptions made by generic algorithms.

The properties of an iterator correspond to properties of the "container" for which it is defined.

Input iterators:
- Operations: equality, inequality, next (++j, j++), dereference (*j)
- No guarantee you can assign to *j: istream_iterator<char>

Output iterators
- Operations: dereference for assignment: *j = t, next (++j, j++)
- May not have equality, inequality: ostream_iterator<int>

Reverse Iterators
Adapted from iterators of container classes.

Containers define the types:
- reverse_iterator
- const_reverse_iterator

Containers provide supporting member functions:
- rbegin()
- rend()