STL Overview

### STL Components

- **Containers**: classes for objects that contain
- **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

### Sequential Access Containers

These templates provide data structures supporting sequentially-organized storage. Sequential access is supported, and in some cases, random access as well.

- **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 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() ) ...`

### 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.
### Vector Example

```cpp
#include <iostream>
#include <iomanip>
#include <vector> // for vector template definition
using namespace std;

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

Specify initial vector size.

Can access like an array...

Warning: the capacity of this vector will NOT automatically increase as needed if access is performed using the [] operator. See the discussion of member functions insert() and put_back().

### STL Vector Indexing

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

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

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

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

### STL Iterators

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.

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.

**Iterator**

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.

### STL Vector Iterator Example

```cpp
string DigitString = "45658228458720501289";
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++;
}
```

Obtain reference to target of iterator.

Advance iterator to next element.

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

Inserting with the `push_back()` member, BigInt will grow to hold as many digits as necessary.
STL Iterator Operations

Each STL iterator provides certain facilities via a standard interface:

```cpp
string DigitString = "45682245872051289";
vector<int> BigInt;
for (int i = 0; i < DigitString.length(); i++) {
    BigInt.push_back(DigitString.at(i) - '0');
}
vector<int>::iterator It;
It = BigInt.begin();
int FirstElement = *It;
It++;
It = BigInt.end();
It--;
int LastElement = *It;
```

- Create an iterator for `vector<int>` objects.
- Target the first element of `BigInt` and copy it.
- Step to the second element of `BigInt`.
- Now it targets a non-element of `BigInt`. Dereference will yield a garbage value.
- Back it up to the last element of `BigInt`.

STL Iterator Operations

STL Iterator Operations

STL Iterator Operations

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 an 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)`.

Insertion into Vector Objects

Insertion into Vector Objects

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)`

Deletion from Vector Objects

Deletion from Vector Objects

Deletion from Vector Objects

STL Overview
Range Deletion Example

```cpp
string DigitString = "00000028458720501289";
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.

Other STL Containers

The STL also provides two standard variants of the linear list:

stack<T>
queue<T>

Generally these conform to the usual expectations for stack and queue implementations, although the standard operation names are not used.

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 `Key`s
- `map<Key, T>`: collection of `T` values indexed by **unique** `Key` values
- `multimap<Key, T>`: possibly duplicate `Key`s

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 $S$:

1. For every $x$ in $S$, $x = x$.
2. For every $x$, $y$, $z$ in $S$, if $x < y$ and $y < z$ then $x < z$.
3. 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:

$$x E y \text{ iff both } x R y \text{ and } y R x \text{ are false}$$

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 $\text{operator}==$.

# Special Function Objects

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

When an ordering is required, the default STL implementation is built around the $\text{less}<T>$ 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>
using namespace std;
#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);
}
```

```cpp
000-00-0000 Ben Keller
111-11-1111 Bill McQuain
888-88-8888 Dwight Barnette
```
STL Overview

**Multiset Example**

```cpp
void main() {
  list<char> L = lst("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++);
  multisetPrint(M, cout);
}
```

![Multiset Example Diagram]

**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 elements 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):

```cpp
map<string, Employee> Payroll;
Payroll.insert(make_pair(Homer.getID(), Homer));
```

A multimap allows duplicate keys:

```cpp
multimap<string, string> mp1;
mp1.insert(make_pair("blue", "Jenny"));
mp1.insert(make_pair("blue", "John"));
```

Map Example

```cpp
#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(make_pair(Bill.getID(), &Bill));
    S.insert(make_pair(Dwight.getID(), &Dwight));
    S.insert(make_pair(Ben.getID(), &Ben));
    EmpmapPrint(S, cout);
    // . . . continues . . .
    000-00-0000 Ben Keller
    111-11-1111 Bill McQuain
    888-88-8888 Dwight Barnette

    EmpmapPrint(S, cout);
    // . . . continues . . .
}
```

Finding Data in Map

```cpp
// . . . continued . . .
S.insert(make_pair("Bill", Bill));
S.insert(make_pair("Dwight", Dwight));
S.insert(make_pair("Ben", Ben));
EmpmapPrint(S, cout);
// . . . continues . . .
```

Use `find(Key)` function to find entry by key:

```cpp
map<string, string> mp;
// insert some values
mp.insert(make_pair("222-22-2222", "name"));
mp.insert(make_pair("222-22-2222", "name2"));
if (mp.find("222-22-2222") != mp.end()) //do something with entry
```

Can manipulate the data entry, but not the key value:

```cpp
(*m_i).first //get key value, cannot be changed (const)
(*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 . . .
```

By Bill McQuain

Of course, the value of the iterator is not checked before dereferencing, so if the specified key value isn’t found (so the iterator equals S.end()), the subsequent dereference will blow up...

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: `mp[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);
// . . . continues . . .
```

By Bill McQuain

000-00-0000 Ben Keller
111-11-1111 Bill McQuain
123-45-6789 Homer Simpson
888-88-8888 Fred Flintstone
Map Example

```cpp
It = S.find("000-00-0000");
if (It != S.end())
    cout << (*It).second->getName() << endl;
It = S.find("000-00-0001");
if (It != S.end())
    cout << (*It).second->getName() << endl;
```

Ben Keller

This prints nothing. No record in the map matches the specified key value, so `find()` has returned the end marker of the map.

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 (i++), dereference (*j)
- No guarantee you can assign to *j: `istream_iterator<char>`

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

Other Iterators

Forward Iterators
- Operations of both input and output iterator
- Iterator value can be stored and used to traverse container

Bidirectional Iterators
- Operations of forward iterators
- Previous: --j, j--

Random Access Iterators
- Bidirectional operators
- Addition, subtraction by integers: r + n, r - n
- Jump by integer n: r += n, r -= n
- Iterator subtraction r - s yields integer

Reverse Iterators

Adapted from iterators of container classes.

Containers define the types:
- `reverse_iterator`
- `const_reverse_iterator`

Containers provide supporting member functions:
- `rbegin()`
- `rend()`
## Choosing a Container

<table>
<thead>
<tr>
<th>Container</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>A vector may be used in place of a dynamically allocated array.</td>
</tr>
<tr>
<td>List</td>
<td>A list allows dynamically changing size for linear access.</td>
</tr>
<tr>
<td>Set</td>
<td>A set may be used when there is a need to keep data sorted and random access is unimportant.</td>
</tr>
<tr>
<td>Map</td>
<td>A map should be used when data needs to be indexed by a unique non-integral key.</td>
</tr>
<tr>
<td>Multiset</td>
<td>Use multiset or multimap when a set or map would be appropriate except that key values are not unique.</td>
</tr>
<tr>
<td>Multimap</td>
<td></td>
</tr>
</tbody>
</table>

©2000 McQuain WD

Computer Science Dept. VA Tech January 2000

OO Software Design and Construction

Copyright 1997-2000