Introducing STL

The Standard Template Library

- A number of template classes designed to provide implementations of collections
  - Designed for performance over safety
  - Designed for use with *values*, so require copy and assignment to be implemented efficiently in element types
  - May expect efficient equality and < operators to be present in element types
- Basis of generic programming support in C++
  - Separate *algorithm* definitions (eg. sort, search) from collections
  - Define iterators in a generic way

Vector

- Basic collection type
  - Like a smart array
  - Resizes automatically when needed
- Implements a single dimensional collection
  - Can have vector of vectors
- Use appropriate header file to bring declarations into program

```
#include <vector>

using namespace std;
```

Vector Example

- Populate a vector of int values then calculate sum and average

```
vector<int> intvec;
for ( int i = 0; i < num; i++ )
{
        int oneval;
        cout << "Enter value " << i << ": ";
        cin >> oneval;
        intvec.push_back( oneval );
}
int sum = 0;
for ( i = 0; i< intvec.size(); i++ )
        sum += intvec.at(i);
cout << "Sum is " << sum << ", average is " << sum / num << endl;
```

Defines a vector for holding simple ints.

push_back adds an element to the end of the vector.

size gives the number of items in the vector. Both at and [ ] return the item at a particular position, but at performs bounds checking.

Vector Capacity

- Vector will grow if needed
  - If push_back() function detects end of underlying array
  - Needs reallocation and copy of elements
- Possible to allocate an initial capacity for the vector
  - reserve() function
- Possible to query current capacity of the vector
  - capacity() function

```
intvec.reserve( 100 );
...
cout << "Capacity: " << intvec.capacity() << endl;
```

Algorithms

- Algorithms provide a generic implementation of a number of common tasks for STL container types
- Example: for_each

```
#include <algorithm>
...
for_each( start_iterator, end_iterator, function_to_perform );
```

The function to be called for each iterated item.

Iterator representing the first item to perform the function with.

Iterator representing the item following the last item to perform the function with.

for_each Algorithm Example

- Display the square of each value in a vector (of int)

```
void iteratorTest()
{
        vector<int> intvec;
        intvec.push_back( 10 );
        intvec.push_back( 20 );
        intvec.push_back( 30 );
        for_each( intvec.begin(), intvec.end(), showSquare );
}

void showSquare( int i )
{
        cout << i * i << endl;
}
```

Function Pointer

```
100
400
900
```

Using Algorithms with Objects

- Algorithms are agnostic about their element types

```
void shower( Fraction f )
{
        f.ShowLine();
}

vector<Fraction> fracVec;
...

for_each( fracVec.begin(), fracVec.end(), shower );
```

A global function takes a Fraction and calls its member function

for_each uses the global function

Algorithms and Member Functions

- Algorithms can use member functions if required
  - mem_fun and mem_fun_ref allow this

```
#include <functional>
...

for_each( fracVec.begin(),
          fracVec.end(),
          mem_fun_ref( &Fraction::ShowLine ) );
```

mem_fun_ref takes the address of the member function which the global function previously called

Alternative for Loop

- Range based for loop
  - Available since C++11
  - Equivalent to STL for_each algorithm

```
vector<int> intvec;
intvec.push_back( 10 );
intvec.push_back( 20 );
intvec.push_back( 30 );

for( int n : intvec )
{
    cout << n * n << endl;
}
```

The body of the loop is executed with n representing each item in intvec in turn

Algorithms Based on Predicates

- Predicate is a function returning bool value
  - Can be used to control an algorithm

```
...
        vector<int> intvec;
        intvec.push_back( 10 );
        intvec.push_back( 20 );
        intvec.push_back( 30 );
        int num = count_if( intvec.begin(), intvec.end(), isAdult );
        cout << "There were " << num << " adults\n";
}
bool isAdult( int i )
{
        return i >= 18;
}
```

Predicate function. If it returns true, the item is included in the count.

Iterators

- Iterator is an object designed to manage traversal of a collection
  - Collection classes define iterators that operate with that collection
- Iterator is like a specialised pointer to an element in the collection
  - Overloaded * operator available to access the object
  - Overloaded ++ operator available to move to next object
- Collections have overloaded functions to return iterators
  - begin() returns iterator initialised to point to first element
  - end() returns iterator initialised to point past the last element

Iterator Example

- Populate vector, then show elements

```
vector<int> intvec;
intvec.push_back( 10 );
intvec.push_back( 20 );
intvec.push_back( 30 );

vector<int>::iterator it = intvec.begin();

cout << *it << endl;

cout << *(++it) << endl;

cout << *(it + 1) << endl;
```

Creates and fills the vector.

'it' is a local variable holding the iterator returned by begin.

Returns the item at the iterator position.

Moves the iterator on one and returns that item.

Returns the item after the iterator position without moving the iterator.

```
10
20
30
```

Iterators and Pointers

- Iterator may look and feel like a pointer
  - It is different, do not assume it can be passed to legacy code as a vanilla pointer
- Address of first element of a vector can be used as a vanilla pointer if necessary
  - &intvec[0]
  - Memory layout in a vector will be the same as in an array
- Altering contents of the vector may cause pointers and iterators to change
  - Since resizing and reallocation may be necessary

Using auto

- Iterator types can be complex
- Since C++11 the language has supported automatic type specification for variables
  - Based on initialising value
  - Does not compromise strong typing

```
auto i = 6;
```

i is declared as in int automatically as 6 is an int

```
vector<int>::iterator it = intvec.begin()

auto it = intvec.begin();
```

These two statements are equivalent

Sorting

- Sort algorithm allows elements of collection to be sorted
  - Uses < and == operators if defined
  - Can also use external predicate implemented as function object
- Step 1: populate vector of Fraction objects

```
vector<Fraction> fracVec;

for( int i = 0; i < 5; i++ )
{
        int n, d;
        cout << "Numerator: ";
        cin >> n;
        cout << "Denominator: ";
        cin >> d;
        Fraction f( n, d );
        fracVec.push_back( f );
}
```

Sorting

- Sort algorithm allows elements of collection to be sorted
  - Uses < and == operators if defined
  - Can also use external predicate implemented as function object
- Step 2: display then sort the vector, display the result

```
for_each( fracVec.begin(),
          fracVec.end(),
          mem_fun_ref( &Fraction::ShowLine ) );
cout << "----" << endl;

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

for_each( fracVec.begin(),
          fracVec.end(),
          mem_fun_ref( &Fraction::ShowLine ) );
```

sort can also take a 3rd argument which is a predicate to do the comparison

```
4/3
1/2
5/4
7/2
4/7
----
1/2
4/7
5/4
4/3
7/2
```

Some Collection Template Classes

- vector
  - Smart array, fast addition to end, fast retrieval of random element, insert and delete slow
- list
  - Doubly linked list, insert and delete fast, sequential traverse fast, random element access slow
- map, multimap, set, multiset
  - map manages items by key, multimap allows duplicate keys
  - set implements traditional set, multiset supports duplicates
- queue, dequeue, priority_queue
  - FIFO, add items to back, remove from front (dequeue supports add and remove from both ends), priority number affects pop in priority_queue
- stack
  - Based on deque, supports only push_back and pop_back

Collection Types

- Define broad properties of collections
  - Access patterns
- E.g. sort algorithm requires random access iterators
  - Vector, deque and string support random access
  - List containers do not support random access through iterators
- Associative containers do not support sort
  - set, multiset, map, multimap
  - Sorted by default

Collection Member Functions

- push_front(), push_back() may be used with non-associative collections
  - push_front() not supported by vector
  - front(), back() used to return reference to appropriate elements
- pop_front(), pop_back() remove items
  - do not return them
- erase() used to remove a range of items using iterators
  - clear() removes all items
  - remove() and remove_if() remove matching items from a list
- insert() adds items to collection
  - assign() replaces contents of a collection with new items

Example

- Add items to both ends of a deque, then assign to a list

```
deque<int> queue;
queue.push_front( 1 );
queue.push_back( 2 );
queue.push_front( 3 );
queue.push_back( 4 );

list<int> l;
l.assign( queue.begin(), queue.end() );
for_each( l.begin(), l.end(), writer );
```

```
void writer( int i )
{
    cout << i << endl;
}
```

```
3
1
2
4
```

Using a Set

- Construct Fraction objects, add to set, display set

```
set<Fraction> fractions;

for( int i = 0; i < 5; i++ )
{
    int n, d;
    cout << "Numerator: ";
    cin >> n;
    cout << "Denominator: ";
    cin >> d;
    Fraction f( n, d );
    fractions.insert( f );
}
for_each( fractions.begin(),
          fractions.end(),
          mem_fun_ref( &Fraction::ShowLine ) );
```

Fraction implemented operator< to support this

The fractions are held in the set in ascending order regardless of the order in which they are entered

```
Numerator: 3
Denominator: 4
Numerator: 2
Denominator: 3
Numerator: 7
Denominator: 4
Numerator: 1
Denominator: 4
Numerator: 1
Denominator: 9
1/9
1/4
2/3
3/4
7/4
```

Using a Map

- Map is constructed as a collection of pair objects
  - pair is a template containing 2 items, first is key, second is value

```
map<string, int>ages;

for( int i = 0; i < 5; ++i )
{
        pair<string, int> details;
        cout << "Name: ";
        cin >> details.first;
        cout << "Age: ";
        cin >> details.second;
        ages.insert( details );
}
```

This map will be used to store ages keyed by names

A pair is used to contain the name and age. This is inserted into the map

Retrieving from a Map

- Use overloaded [] operator
  - If no match, return "default" value of type

```
string name;
int age;
while( name != "quit" )
{
        cout << "Name: ";
        cin >> name;

        age = ages[name];
        cout << name << " is " << age << " years old " << endl;
}
```