STL Containers & Iterators

Containers as a form of Aggregation

- Fixed aggregation
  - An object is composed of a fixed set of component objects
- Variable aggregation
  - An object is composed of a variable set of component objects
- Containers
  - An object exists in its own rights but it is able to hold other objects.

Generic Containers

- Container classes are building blocks used to create object-oriented programs, and they make the internals of a program much easier to construct.
- A container class describes an object that holds other objects.
- Container classes are so important that they were considered fundamental to early object-oriented languages.
- The C++ approach to containers is based on templates. The containers in the Standard C++ library represent a broad range of data structures designed to work well with the standard algorithms and to meet common software development needs.

Standard Template Library

- The standard template library (STL) contains
  - Containers
  - Iterators
  - Algorithms
- A ***container*** is a way stored data is organized in memory, for example an array of elements.
- ***Algorithms*** in the STL are procedures that are applied to containers to process their data, for example search for an element in an array, or sort an array.
- ***Iterators*** are a generalization of the concept of pointers, they point to elements in a container, for example you can increment an iterator to point to the next element in an array.

Containers in Software

- A container is usually instantiated as an object of container class
- A container class object encapsulates inside it a mechanism for containing other objects
- It also provides the necessary behaviour for adding, removing and accessing the objects it contains
- A container class gives the opportunity of reuse in different programs:
  - this frees the programmer from having to recreate complex data structures in every program to manage complex data structure

Some Containers Types

- Sequential containers: vector, list and deque; They store elements in client-visible order
- Associative containers: map, multimap, set and multiset
- Containers Adapters: queue, priorityqueue and stack

Sequence Containers

- A sequential container stores elements in a sequence. In other words each element (except for the first and last one) is preceded by one specific element and followed by another, **`<vector>`**, **`<list>`** and **`<deque>`** are sequential containers
- In an ordinary C++ array the size is fixed and can not change during run-time, it is also tedious to insert or delete elements. Advantage: quick random access
- **`<vector>`** is an expandable array that can shrink or grow in size, but still has the disadvantage of inserting or deleting elements in the middle

Sequence Containers

- **<list>** is a double linked list (each element has two pointers to its successor and predecessor), it is quick to insert or delete elements but has slow random access

- **<deque>** is a double-ended queue, that means one can insert and delete objects from both ends, it is a kind of combination between a **stack** (LIFO) and a queue (FIFO) and constitutes a compromise between a **<vector>** and a **<list>**

Associative Containers

- An associative container is non-sequential but uses a ***key*** to access elements. The keys, typically a number or a string, are used by the container to arrange the stored objects in a specific order, for example in a dictionary the entries are ordered alphabetically.

Associative Containers

- A **<set>** stores a number of items which contain keys. The keys are the attributes used to order the items, for example a set might store objects of the class Books which are ordered alphabetically using their title

- A **<map>** stores pairs of objects: a key object and an associated value object. A **<map>** is somehow similar to an array except instead of accessing its elements with index numbers, you access them with indices of an arbitrary type.

- **<set>** and **<map>** only allow one key of each value, whereas **<multiset>** and <**multimap**> allow multiple identical key values

List

- a standard doubly linked container
- supports constant-time insertion and deletion of elements at any point of the list
- Most list operation are identical to those of a vector
- However, list do not provide random access to elements
- **Insert(), erase()** run in constant time **which makes lists** Suitable for applications that perform many insertion and deletion

  –

The use of containers

- They give us control over collections of objects, especially dynamic objects
- Gives a simple mechanism for creating, accessing and destroying without explicitly programming algorithms to do these operation
- "**Iterator**" methods allow us to iterate throw the container

Container
add object **find object** **remove object** **isempty**

Requirements on elements in containers

Method	Description	Notes
Copy Constructor	Copy Constructor creates a new element that is “equal”to safely be destroyed without affecting the old one	Used every time you insert an element
Assignment operator	Replaces the content of an element with a copy of the source element	Used every time to modify an element
Destructor	Cleans up an element	Used every time you remove an element.
Default Constructor	Construct an element without any argument	Required only for some operations
operator ==	Compares two elements for equality	Required only for some operations
operator<	Determines if one element is less than another	Required only for some operations

A simple Example

- Here's an example using the set class template.
- This container, modelled after a traditional mathematical set, does not accept duplicate values.
- The following set was created to work with

Int Set

```
#include <iostream>                                    Intset.cpp
#include <iterator>
#include <set>
using namespace std;
int main(void)
{
  set<int> intset;
  for(int i = 0; i < 250; i++)
    for(int j = 0; j < 24; j++)
       intset.insert(j);
  cout <<"Set Size "<<intset.size()<<endl;
  cout <<"Contents"<<endl;
  copy(intset.begin(),intset.end(),
      ostream_iterator<int>(cout, ", "));
cout<<"\n";
}
```

Set

- The **`insert( )`** member does all the work:
- it attempts to insert an element and ignores it if it's already there.
- Often the only activities involved in using a set are simply insertion and testing to see whether it contains an element.
- You can also form a union, an intersection, or a difference of sets and test to see if one set is a subset of another.
- In this example, the values 0-24 are inserted into the set 250 times, but only the 25 unique instances are accepted.

The Copy Algorithm

- To print out the contents of the set we use, a special algorithm called **`copy`** In conjunction with the **`ostream_iterator`**
- We use the **`copy`** algorithm to remove the loop we could use to access the elements of the set.
- and the **`begin()`** and **`end()`** methods of the set to gather the extents of the set to copy.

iterator

- once we have the values to copy we need to copy it to some place which in this case is the **`ostream_iterator`** class template declared in the **`<iterator>`** header.
- An `ostream_iterator` is an Output Iterator that performs formatted output of objects of type T to a particular ostream.
- Output stream iterators overload their copy assignment operators to write to their stream.
- This particular instance of **`ostream_iterator`** is attached to the output stream **`cout`**
- It is possible to attach it to a file and get the results copied to a file

iterator

- Every copy assigns an integer from the set to `cout` through this iterator, the iterator writes the integer to `cout` and also automatically writes an instance of the separator string found in its second argument, which in this case contains a **,**
- It is just as easy to write to a file by providing an output file stream, instead of **`cout`**

More Complex example: countw.cpp

```
using namespace std;
void CountUniqueWords(const char* fileName)
{
ifstream source(fileName);
if(!source)
  {  cerr<<"error opening file\n";
     exit(EXIT_FAILURE);
  }
string word;
set<string> words;
while(source >> word)
      words.insert(word);
cout << "Number of unique words:"<< words.size() << endl;
copy(words.begin(), words.end(),ostream_iterator<string>(cout,
}
int main(int argc, char* argv[])
{
if(argc > 1) CountUniqueWords(argv[1]);
else
  cerr<<"Usage "<<argv[0]<<" file name"<<endl;
}
```

Iterators

- Iterator is a generation of pointer
- It is an object belonging to a class with the prefix * defined on it
- So that, if **p** in an iterator over a container, ***p** is an object in the container.
- You can think of iterator as pointing to a current object at any time

```
vector<float> L;
for(vector<float>::iterator p=L.begin(); p<L.end(); p++)
     cout << *p <<endl;
```

Different Iterators

- An iterator is an abstraction for genericity.
- It works with different types of containers without knowing the underlying structure of those containers.
- Most containers support iterators, so you can say

```
<ContainerType>::iterator
<ContainerType>::const iterator
```

- to produce the iterator types for a container.

Different Iterators

- Every container has a **`begin( )`** member function that produces an iterator indicating the beginning of the elements in the container, and an **`end( )`** member function that produces an iterator which is the past-the-end marker of the container.
- If the container is **`const`**, **`begin( )`** and **`end( )`** produce **`const_iterators`**, which disallow changing the elements pointed to (because the appropriate operators are **`const`**).

Iterators II

- All iterators can advance within their sequence (via operator++) and allow **`==`** and **`!=`** comparisons.
- Thus, to move an iterator **`it`** forward without running it off the end, you say something like:

```
while(it != pastEnd)
     {
      //Do something
       ++it;
     }
```

- where **`pastEnd`** is the past-the-end marker produced by the container's **`end( )`** member function.

Iterators III

- An iterator can be used to produce the container element that **`it`** is currently selecting via the dereferencing operator **`(operator*).`**
- This can take two forms. If **`it`** is an iterator traversing a container, and **`f()`** is a member function of the type of objects held in the container, you can say either:

```
(*it).f();
  or
it->f();
```

Shapes

- The following example shows how we can use the vector container and run-time polymorphism to access different objects

`shapes.cpp`

A different way of Iterating

shape2.cpp

```
ShapeIter begin = shapes.begin();
ShapeIter end = shapes.end();
while(begin !=end)
{
    (*begin)->draw();
    ++begin;
}
begin = shapes.begin();
end = shapes.end();
while(begin !=end)
{
   delete (*begin);
   ++begin;
}
```

for_each

- **for_each** allows us to call the same function for each of the elements in the container

```
for_each(shapes.begin(),shapes.end(),mem_fun(&Shape::draw));
```

- however we can only call for void methods (or using more complex structures) for pointers to containers or other unarray functions

Sorting

- the sort algorithm allows objects to be sorted.

`sorting.cpp`

Reverse Iterators

- A container may also be reversible, which means that it can produce iterators that move backward from the end, as well as iterators that move forward from the beginning.
- All standard containers support such bidirectional iteration.
- A reversible container has the member functions:
  - **`rbegin()`** (to produce a reverse iterator selecting the end)
  - **`rend()`** (to produce a reverse iterator indicating "one past the beginning").
- If the container is **`const`**, **`rbegin`**( ) and **`rend`**( ) will produce **`const_reverse_iterators.`**

reverseit.cpp

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main(int argc, char **argv)
{
 if(argc>1)
 {
  ifstream in(argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line))
     lines.push_back(line);
  vector<string>::reverse_iterator r=lines.rbegin();
  vector<string>::reverse_iterator end=lines.rend();
  for(r; r!=end; r++)
    cout << *r << endl;
   }
}
```

Vectors

- **`reserve()`** This method can be used to reserve a predefined amount of data space for the vector
- **`size()`** this reports how many elements in the vector
- **`erase(itor begin, itor end)`** erase a range of elements of the vector
- **`insert(itor where, value)`**

Converting Between sequences

`convert.cpp`

The Singleton Pattern

- The singleton pattern defines an object that can only exist once
- This is done by implementing the code in a particular way with the object knowing if it has been created
- If it has not it will create an instance of itself
- If it has been created it will return the instance.
- This pattern is a good way of storing global state data within a program.

Global.h

```
#include <iostream>
 class Global
{
    private :
        static Global* m_pinstance;
        int m_age;
        inline Global(){;}
        inline ~Global(){;}
        inline Global(const Global &_g){;}
  public :
        static Global* Instance();
        void setName( const std::string &_name );
        std::string getName();
        void setAge(int _age);
        int getAge();
};
```

Global.cpp

```
Global* Global::m_pinstance = 0;//initialize pointer

Global* Global::Instance()
{
  if (m_pinstance == 0)   // is it the first call?
  {
    std::cout<<"new instance\n";
    m_pinstance = new Global; // create sole instance
  }
  std::cout<<"existing object\n";
  return m_pinstance; // address of sole instance
}

void Global::setName(const std::string &_name)
{
  m_name=_name;
}
std::string Global::getName(){ return m_name;}
```