Dalhousie University
Faculty of Computer Science

CSCI 3132 – Object Orientation and Generic Programming

Week 9 – Generic Programming and STL

Template Functions & Template Classes

What is Generic Programming

- *Programming paradigm for developing efficient, reusable software that
  - Focuses on finding commonality among similar implementations of the same algorithm
  - Provides suitable abstractions so that a single, generic algorithm can cover many concrete implementations.

*Ref: generic-programming.org

What is Generic Programming

- Writing "templates" of source codes
  - Parameters unspecified at the time of class definition
  - Programs evaluated at compile time
  - Instantiation of the templates provide information to the compiler for the type it should use to create the class out of the template

Template – Example

- Templates can be:
  - Function templates that behave like functions can accept many different kinds of arguments
  - Example:

```
 template <class T>
T max(T x, T y){
       if (x < y)   return y;
       else         return x;
}
```

    - Compare this with
      `#define max(a,b)   ((a) < (b) ? (b) : (a))`
  - Class templates that are often used to make generic containers
  - Example: Linked list container of the STL library
    - list<*type*> has a set of standard functions associated with it that work, regardless of what the type is.
    - *more on these later*

Templates

- Advantages:
  - Already looked at while studying templates
  - Type-safety, code reusability, …
- Disadvantages:
  - Most compilers have poor support for templates
  - Difficult to make sense of error messages
    - Executed code is generated by the compiler and is not present in the original code
  - Compiler generates extra code for each use of template (template instantiation)
    - Indiscriminate use may result in code bloat and large executables
    - Separate compilation of template definitions and template function declarations not implemented by most compilers

Overloading with Function Templates

- Function templates may be overloaded
  - Provide different parameter lists
- Example:
  - ```
    template <class T>
    T prod (T x, T y)
    {   return x*y;   }
    T prod (T x, T y, T z)
    {   return x*y*z; }
    ```
    ```

Overloading with Function Templates

- Non-template version of a function can be used with a template version
  - Used if implementation of a function is different for different data types (e.g. char and int)
- Example:

```
#include <iostream>
#include <string>
using namespace std;
char* prod(char* x, char *y) {
        char * p = new char[2];
        p[0]=*x;    p[1]=*y;
        return p;
 }
template <class T>
T prod(T x, T y) { return x*y; }
int main() {
        int x = 1, y = 2;  char a='A', b='B';
        cout << prod(x, y) << " " << prod(&a, &b) << endl;
        return 0;
}
```

Class Templates

- Templates may also be used to create generic classes
  - Allow you to create one general version of a class without having to duplicate code to handle multiple data types
  - Example: SimpleArray class instead of IntArray, FloatArray, DoubleArray etc.

Class Templates

- Declaring a class template similar to declaring a function template
- Example:

```
template <class T>
class Point {
private:
   T ptX;
   T ptY;
public:
   Point() {  ptX = 0; ptY = 0;  }
   Point(T x, T y) {  ptX = x; ptY = y;  }
   …
};
```

- When defining objects of the template class, the type parameter must be specified
  - Example: `Point<int> pt;`

Exercise

- Complete the point template class shown earlier, by writing and testing the following functions:
  a) Copy constructor for the Point class
  b) Functions GetX and GetY to get x and y coordinates
  c) Function SetPoint and ShowPoint to set and display the values of x and y coordinates
  d) Appropriate overloaded operators that can be used to multiply the point with a scalar

Exercise 1 – Solution

```
//a) Copy Constructor
Point(const Point & pt) {
   ptX = pt.ptX;
   ptY = pt.ptY;
}

//b) GetX and GetY
T GetX(const Point & pt) { return pt.ptX; }
T GetY(const Point & pt) { return pt.ptY; }

//c) SetPoint and ShowPoint
void SetPoint(T x, T y) { ptX = x; ptY = y; }
void ShowPoint() {
      cout << "(" << ptX << ", " << ptY << ")\n";
}
```

Exercise 1 – Solution

```
//d) Appropriate overloaded operators: *, = and *=
Point & operator* (T val1) {
      ptX = ptX * val1;
      ptY = ptY * val1;
      return *this;
}
Point & operator= (const Point & p) {
      ptX = p.ptX;
      ptY = p.ptY;
      return *this;
}
Point & operator*= (const T & i) {
      ptX *= i;
      ptY *= i;
      return *this;
}
```

Exercise 1 – Testing the Template Class

```
int main() {
        int x = 2, y = 5;
        double dx = 2.2, dy = 3.3;

        Point<int> pt;       //note the type parameter
        pt.SetPoint(x, y);
        pt *= 2;
        pt.ShowPoint();

        Point<double> dpt;
        dpt.SetPoint(dx, dy);
        dpt = dpt * 2;
        dpt.ShowPoint();
}
```

Exercise 2 – Template Functions with Static variables

What is the output of the following code and why?

```
#include <iostream>
template <class T>
void GrowShape(const T&g){
        static T size = 1;
        size *= g;
        std::cout<<size<<std::endl;
}
int main(){
    int x = 2;
    float y = 1.1;
    GrowShape(x);
    GrowShape(x);
    GrowShape(y);
    GrowShape(y);
    GrowShape(x);
}
```

```
2
4
1.1
1.21
8
```

Exercise 3 – Template Class with Static variables

What is the output of the following code and why?

```
#include <iostream>
template <class T>
class shape{
protected:
    T dim;
public:
    shape(const T&d):dim(d){}
    void GrowShape(const T&g){
        static int count = 0;
        dim *= g;
        count++;
        std::cout<<count<<": "<<dim<<std::endl;
    }
};
int main()
{
    shape<int> s(1);
    s.GrowShape(2);
    s.GrowShape(3);
    s.GrowShape(4);
    shape<float> f(1.0f);
    f.GrowShape(1.1f);
    f.GrowShape(2.2f);
}
```

```
1: 2
2: 6
3: 24
1: 1.1
2: 2.42
```

Generic Programming

Generic Programming – History

- Originally pioneered by ML (MetaLanguage, also called LISP with types) in 1973
  - Permits writing common functions or types that differ only in the set of types on which they operate when used.
- Such entities are known as:
  - Generics in most languages including Ada, Java, C#, VB, .NET
  - Parametric polymorphism in ML, Scala, Haskell
  - Templates in C++ and D

Ref: Wikipedia – Generic Programming

Examples of Generic Programming

- Standard Template Libraries (STL)
  - Generic C++ library of container classes, algorithms and iterators
  - Almost every component in the STL is a template
    - Can be instantiated to contain any type of object
  - STL algorithms are decoupled from the STL container class
    - Usually implemented as global functions
  - STL iterators are generalization of pointers
    - Iterators make it possible to decouple algorithms from containers

Concepts

- Consider the following piece of code:

```
template <class InputIterator, class T>
InputIterator find      (InputIterator first,
                         InputIterator last,
                         const T& value) {
      while (first != last && *first != value)
            ++first;
      return first;
}
```

  - What is the set of types that can be correctly substituted for the `InputIterator` parameter?
    - int *, double * but not int or double
  - find(…) implicitly defines a set of requirements on types

Concepts

- Types substituted in **find** must be able to provide certain operations, for example:
  - Compare two objects (first != last) of the type
  - Increment an object of that type (++first)
  - Dereference an object of that type to obtain an object that it points to (*first != value)
- Such a set of type requirements is called a concept
  - A type conforms to a concept if it satisfies all of those requirements for that concept

Concepts

- Using concepts makes it possible to write programs that separate interface from implementation
  - While creating the **find** template function, the author only has to consider the interface provided by the concept (in this case input iterator)
    - No worries about implementation of every possible type that conforms to that concept
  - While using the find function, one only needs to ensure that the passed arguments are models of the input iterator
- Programming in terms of concepts rather than specific types makes it possible to reuse and combine software components together

STL

Standard Template Library

Components of STL

- What is STL?
  - “The Standard Template Library provides a set of well structured generic C++ components that work together in a seamless way.
- What does STL consist of:
  - Containers
    - Objects that store other objects (its elements) and have methods for accessing its elements
  - Iterators
    - Implementation of the Iterator pattern that provides sequential access to the elements of an aggregate object without exposing the object’s underlying representation
  - Algorithms
    - Components that perform algorithmic operations on containers
    - Operate over iterators rather than containers

Containers

STL Container Classes

- STL includes the following containers:
  - Sequential containers
    - Templates supporting sequentially organized storage
    - support sequential access, may also support random access
  - Associative containers
    - Templates containing a key associated with a value
    - Not necessarily indexed with sequential integers
  - Adapters
    - Adapt a (sequential) container to provide a specific functionality or interface

STL Container Classes – Sequential

- Sequential containers
  - vector
    - A dynamic array that grows and shrinks at the end
    - Constant time insert/delete at the end
    - Provides indexed storage
    - Supports random access iterators
  - list
    - (Doubly) linked list, with data elements linked together through pointers
    - Constant time insert/delete at any point in the list
    - No random access iterator
  - deque
    - Double ended queue
    - Constant time insert/delete at both ends
    - Supports random access iterators

STL Container Classes – Associative

- Associative – support direct lookup via complex key-values
  - set
    - Ordered collection of unique keys
  - map
    - Container with key-value pairs
  - multiset and multimap
    - Set that can support multiple equivalent (non-unique) keys / key-value pairs

STL Container Classes – Adapters

- Adapters
  - stack
    - Container providing Last-in First-out (LIFO) access
    - Usually adapts (built on top of) a deque container
  - queue
    - Container providing First-in First-out (FIFO) access
    - Usually adapts a deque container
  - Priority-queue
    - Container providing constant time lookup of the default (largest) element at the expense of logarithmic insertion/extraction
    - Usually adapts a vector container

The Vector Container

Vector

- Similar to arrays but with dynamic length
  - Length handled automatically by the container
  - Consume more memory than arrays as they keep extra storage for growth
  - Elements can be accessed using offsets
- Compared to other dynamic sequence containers:
  - Element access is very efficient
  - Adding or removing elements from its end is relatively efficient
  - Perform poorly for operations involving insertion and deletion of elements at positions other than the end

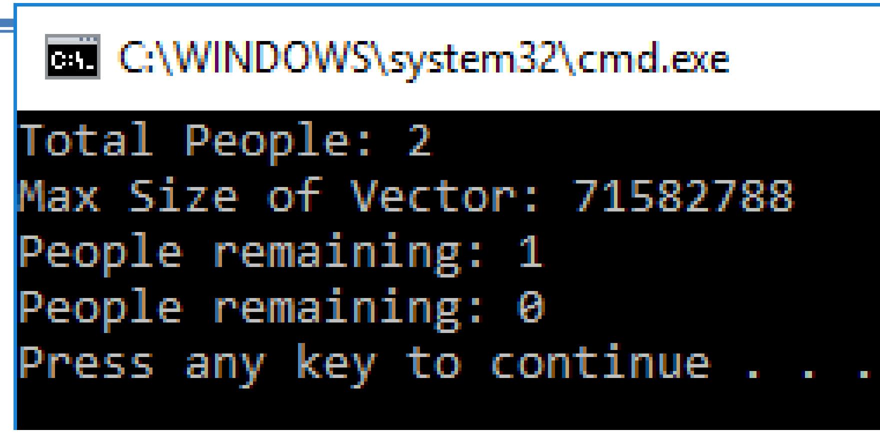
Vector – Member Functions

- Some frequently used member functions
  - Iterators:
    - begin, end, rbegin, rend
  - Capacity:
    - size, max_size, resize, empty, capacity
  - Element access:
    - operator [ ], at, front, back
  - Modifiers:
    - assign, push_back, pop_back, insert, erase, swap

Vector – Example 1

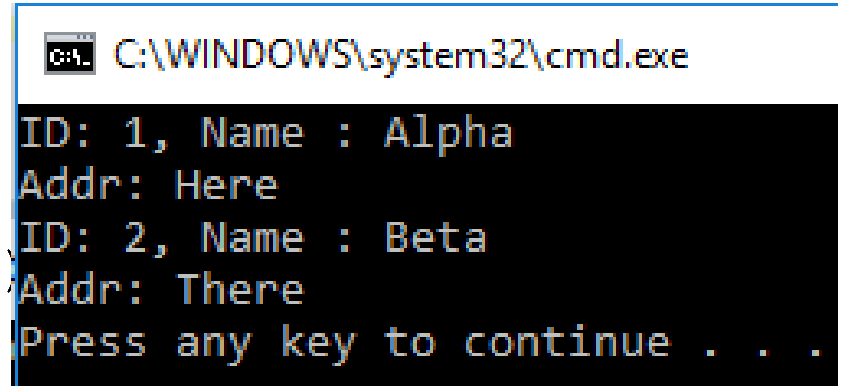
```
#include<vector>
int main(){
   vector <People> myPeople;
   People p1(1,"Alpha","Here");
   myPeople.push_back(p1);
   People * p2 = new People(2, "Beta", "There");
   myPeople.push_back(*p2);
   cout << "Total People: " << myPeople.size()<<endl;
   cout << "Max Size: " << myPeople.max_size() << endl;

   while(!myPeople.empty()) {
      myPeople.pop_back();
       cout << "People remaining: " << myPeople.size()<< endl;
   }
   return 0;
}
```



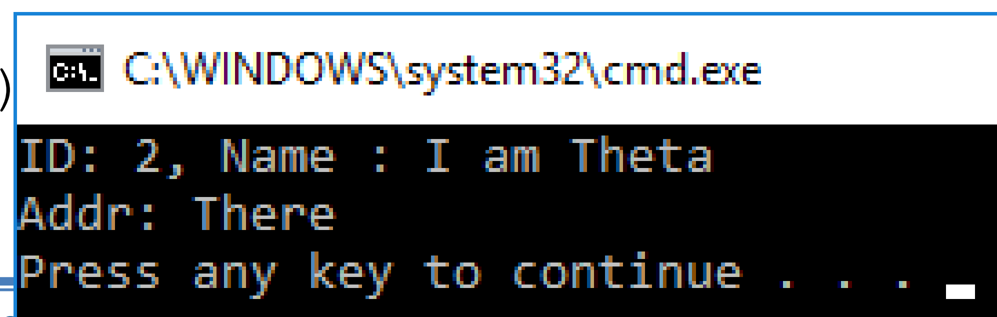
Vector – Example 2 – Using an Iterator

```
#include<vector>
int main(){
   vector <People> myPeople;
   People p1(1,"Alpha","Here")
   myPeople.push_back(p1);
   People * p2 = new People(2, "Beta", "There");
   myPeople.push_back(*p2);
   vector<People>::iterator it;
   for (it = myPeople.begin();
        it != myPeople.end(); it++) {
          it->DisplayProfile();
   }
   return 0;
}
```



Vector – Example 3

```
#include<vector>
int main(){
	vector <People> myPeople;
	People p1(1,"Alpha","Here");
	myPeople.push_back(p1);
	People * p2 = new People(2, "Beta", "There");
	myPeople.push_back(*p2);
	vector<People>::iterator it;
	it = myPeople.end() - 1;
	string nm = "I am Theta";
	it->SetName(nm);
	it->DisplayProfile()
	return 0;
}
```



Vector – Exercise

- Create a vector containing 10 integers in seq
  - Display the integers on console using iterator
- Assign this vector in reverse order to another vector
  - Display the integers on console using indices
- Delete all integers from the vector

Vector

- 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).

Vector Invalidation

```
vector<int>::iterator j;
j = V.begin();
while (j != V.end())
V.erase(j++);
```

- The above code doesn’t work because the iterator is invalidated following the deletion point

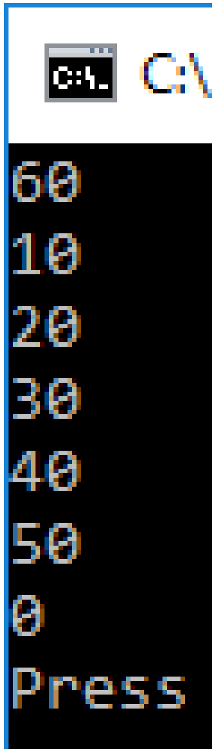
The Deque Container

Deque

- Deque is implemented as a vector of vectors
  - Think of it as several chunks of double-ended queues in memory, with a queue saving the location of each chunk
    - Uses multiple blocks of sequential storage
  - Empty spaces are available both at the front and the back for push_front() and push_back()
    - Doesn’t need to copy and destroy contained objects during a new storage allocation
- Deque has similar member functions as vector, with the addition of push_front(), pop_front() and other similar functions

Deque – Example

```cpp
#include <iostream>
#include <deque>
using namespace std;
int main(){
     deque<int> d = {10, 20, 30, 40, 50};
     d.push_front(60);
     d.push_back(0);
     for(int n : d) {
          cout << n << '\n';
     }
}
```



When to use Deque over Vector

- If you want to be able to add elements to both ends
- If the size of your collection is very large
  - Relocation not needed
  - Lesser chance of running out of memory as you do not need a contiguous memory block
- For other cases – use vector as it is simpler

Associative Containers

Associative Containers

- Set stores a collection of unique values
  - Similar to a mathematical set
  - Accepts only one copy of each element
  - Also sorts the elements in ascending order
    - Stores elements in a balanced tree data structure
  - The value of an element identifies itself
- Example:
  - Creating an index for a book
    - Want only one instance of each word
    - Want all words sorted for quick lookup

Set vs Other Containers

- Why Sets?
  - set::find() is much faster, and hence insertion is faster (log N)
- However, overhead is much more
  - STL provides several generic algorithm, e.g. lower_bound(first, last, x) that search in log N time
  - Both std::lower_bound and set::find take time proportional to log N, but the constants of proportionality are very different
    - "Using g++ on a 450 MHz Pentium III it takes 0.9 seconds to perform a million lookups in a sorted vector of a million elements, and almost twice as long, 1.71 seconds, using a set" [1]
    - "the set uses almost three times as much memory (48 million bytes) as the vector (16.8 million)" [1]
- So what container to use?
  - You can use whatever data structure is convenient, so long as it provides STL iterator
  - Usually, it's easiest and sufficient to use a C array, or a vector in most cases

[1] Matt Austern, "Why shouldn't you use set" [http://lafstern.org/matt/col1.pdf]

Vector – Searching in *logN* Example[1]

```
template <class vector, class T>
void insert_into_vector(vector& v, const T& t) {
      typename vector::iterator i
         = std::lower_bound(v.begin(), v.end(), t);
      if (i == v.end() || t < *i)
            V.insert(i, t);
}
```

- In the above example, of course, the vector needs to be sorted
  - Insertion is still proportional to *N,* as a vector needs to make space by shifting the elements
- This just illustrates the power of STL containers and algorithms, and the myriad options that are available to a programmer
  - Which one is used depends on the need of the program and the programmer's own preferences

[1] Matt Austern, "Why shouldn't you use set" [http://lafstern.org/matt/col1.pdf]

When to use SET[1]

- “Use a set when all or most of these conditions exist:
  - Collection is large enough that the difference between O(N) and O(logN) is important
  - The number of lookups is of a similar order to the number of insertions, so that insertion speed becomes relevant
  - Elements are inserted in random order
  - Insertions and lookups are not performed in distinct phases”
- “As a rule, always use the simplest data structure that meets your needs
  - More complicated a data structure, the less use it finds”

[1] Matt Austern, “Why shouldn’t you use set” [http://lafstern.org/matt/col1.pdf]

Maps

- In maps, one object is associated with another in an array-like fashion.
  - Instead of selecting an array element with a number, you look it up with an object.
- A map holds:
  - A key; what you look up by (mapname[key])
  - A value that results from the lookup with the key
- An iterator is used to move through the map
  - Dereferencing it, produces a pair object (key-value)
  - Members of a pair are accessed by selecting first or second

MAP – Example

```
#include <iostream>
#include <map>
#include <string>
#include <iterator>
using namespace std;
int main(){
      map<string, int> mpPlanets;
      mpPlanets.insert(make_pair("Mercury", 1));
      mpPlanets.insert(make_pair("Venus", 2));
      mpPlanets["Earth"] = 4;
      mpPlanets["Mars"] = 4;
      mpPlanets["Jupiter"] = 5;
      mpPlanets["Saturn"] = 6;
      mpPlanets["Uranus"] = 7;
      mpPlanets["Neptune"] = 8;
      mpPlanets["Earth"] = 3;
```

MAP – Example

```
map<string, int>::iterator it = mpPlanets.begin();
while (it != mpPlanets.end()){
        cout<<it->first<<" - "<<it->second<<endl;
                it++;
        }

if (mpPlanets.find("Earth") != mpPlanets.end())
        cout << "We found 'Earth`"<< endl;
if (mpPlanets.find("Sun") == mpPlanets.end())
        cout << "'Sun' not found" << endl;
return 0;
}
```

MAP – Example

C:\WINDOWS\system32\cmd.exe

```
Earth - 3
Jupiter - 5
Mars - 4
Mercury - 1
Neptune - 8
Saturn - 6
Uranus - 7
Venus - 2
We found 'Earth'
'Sun' not found
Press any key to continue . . .
```

MAP – Changing the Sorting Criteria

- The default sort order in set and map is ascending (lowest first)
  - Default order can be changed by using a custom sort order

```
struct RevOrder
{
      bool operator()(const string & left,
            const string & right) const
      {
            return (left > right);
      }
};
…
//when declaring a variable
map< string, int, RevOrder > mpPlanets;
```

MAP – Changing the Sorting Criteria

C:\WINDOWS\system32\cmd.exe

```
Venus - 2
Uranus - 7
Saturn - 6
Neptune - 8
Mercury - 1
Mars - 4
Jupiter - 5
Earth - 3
We found 'Earth'
'Sun' not found
Press any key to continue . . .
```

Choosing a Container

- A vector may used in place of a dynamically allocated array. Most commonly used container.
- A list allows dynamically changing size for linear access
- A set may be used when there is a need to keep data sorted and random access is unimportant
- A map should be used when data needs to be indexed by a unique non-integral key
- Use multiset or multimap when a set or map would be appropriate except that key values are not unique.

STL Iterators

STL Iterators

- Iterator is an object that
  - Keeps track of a location within an associated STL container object
  - Provides support for traversal (increment/ decrement), dereferencing, and container bounds detection
- An iterator is declared with an association to a particular container type
  - E.g. vector<int>::iterator iter;
- Each STL container type includes begin() and end() member functions that specify iterator values for the first and first past last element

Iterator Types

Container Class	Iterator Type	Container Category
vector	random access	sequential
deque	random access	
list	bidirectional	
set	bidirectional	associative
multiset	bidirectional	
map	bidirectional	
multimap	bidirectional	
stack	none	adaptor
queue	none	
priority_queue	none	

Iterator Types

Iterator Type	**Behavior**	**Supported Operations**
Random access	Move forward and backward Access values randomly Store and retrieve values	* = ++ –> == != –– + – [] < > <= >= += –=
Bidirectional	Move forward and backward Store and retrieve values	* = ++ –> == != ––
Forward	Move forward only Store and retrieve values	* = ++ –> == !=
Input	Move forward only Retrieve but not store values	* = ++ –> == !=
Output	Move forward only Store but not retrieve values	* = ++

Output Iterator

- Supports ++iter and iter++
- Also supports *iter = x to store data
- Cannot read from, or test with == or !=
- Not very useful on its own, but has two useful sub-classes
  - Insert iterator
    - Insert instead of overwriting
    - No need to increment the iterator
  - Ostream output iterator
    - Allows pointing to an output stream and insert elements
- Example:

```
ostream_iterator<int> outIter( cout, "\n" );
...
copy( v.begin(), v.end(), outIter );
```

Input Iterators

```
vector<int> v;
vector<int>::iterator iter;

v.push_back(1);
v.push_back(2);
v.push_back(3);

for (iter = v.begin(); iter != v.end();iter++)
     cout << (*iter) << endl;
     //iterator being used to retrieve values
```

Forward Iterator

- Forward iterators combine input and output iterators
  - Can be used to read or write to a container
- Example

```
template <class ForwardIterator, class T>
void
replace(ForwardIterator first, ForwardIterator last,
        const T& oVal, const T& nVal){
   while (first != last){
      if (*first == oVal)
          *first = nVal;
      ++first;
   }
}
...
replace (vec.begin(), vec.end(), 1, 10);
```

Bidirectional Iterators

- Similar to forward iterators, but support the decrement operator

```
template <class BidirectionalIterator, class
   OutputIterator> OutputIterator
reverse_copy(BidirectionalIterator first,
             BidirectionalIterator last,
             OutputIterator result)
{
   while (first != last)
      *result++ = *--last;
   return result;
}
…
reverse_copy(lst.begin(),lst.end(),vec.begin());
```

Random Access Iterator

- Random access iterators allow values to be
  - Accessed by subscript
  - Subtracted from one another
  - Modified by arithmetic operators
- Random access iterators are similar to pointers

Iterators – Summary

Iterator Type	Produced by
Input	istream_iterator
Output	ostream_iterator inserter(), front_inserter(), back_inserter()
Bidirectional	list set and multiset map and multimap
Random access	Conventional pointers vector deque

STL Algorithms

Algorithms

- Several algorithms provided in the STL algorithm library, including:
  - Accessors
  - Numeric
  - Copying
  - Sorting

Accessor Algorithms

- access but do not change the contents
  - find
    - InputIterator find(InputIterator begin, InputIterator end, const T& value);
  - find_if
    - InputIterator find_if(InputIterator begin, InputIterator end, Predicate pred);
  - count
    - size_t count(InputIterator begin, InputIterator end, const T& value);
  - count_if
    - size_t count_if(InputIterator begin, InputIterator end, Predicate pred);

Numeric Algorithms

- max_element and min_element

```
ForwardIterator max_element(ForwardIterator
begin, ForwardIterator end)


ForwardIterator min_element(ForwardIterator
begin, ForwardIterator end)
```

- accumulate

```
T accumulate(InputIterator begin,
             InputIterator end, T initial_value)
```

Copying Algorithms

```
OutputIterator copy(InputIterator begin1,
          InputIterator end1, OutputIterator begin2)

OutputIterator remove_copy(InputIterator begin1,
          InputIterator end1, OutputIterator begin2,
          const T& value)

OutputIterator remove_copy_if(InputIterator begin1,
         InputIterator end1, OutputIterator begin2,
          Predicate pred)

OutputIterator transform(InputIterator begin1,
        InputIterator end1, OutputIterator begin2
          UnaryOperation op)
```

Sorting Algorithms

```
void sort(RandomAccessIterator begin,
          RandomAccessIterator end)

void sort(RandomAccessIterator begin,
          RandomAccessIterator end,
          Compare comp)

void partial_sort(RandomAccessIterator begin,
                  RandomAccessIterator middle,
                  RandomAccessIterator end)
```

References and Resources

- SGI
  - http://www.sgi.com/tech/stl/
  - STL source code download: http://www.sgi.com/tech/stl/download.html
- Microsoft - MSDN
  - https://docs.microsoft.com/en-us/cpp/standard-library/cpp-standard-library-reference
- Code Project
  - https://www.codeproject.com/KB/stl/
- CPlusPlus website
  - http://www.cplusplus.com/reference/
- Book
  - The C++ Standard Library: A Tutorial and Reference by Nicolai M. Josuttis