Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 12 | The Standard Template Library II

The **`string`** as a STL container

In Lecture 8, the discussion of advanced string operations was postponed because they are implemented using the STL algorithm library, as part of the C++ language. The **`string`** can be viewed as a sequential container, similar to the vector. Indeed, **`string`** and **`vector<char>`** provide similar capabilities. Consider the following program.

```
string alph = "the quick brown fox jumps over the lazy dog";
string::iterator sib = alph.begin();
string::iterator sie = alph.end();
string::iterator sf = find(sib, sie, 'x');
unsigned nx = sf - sib; //  nx == 18
```

For convenience, the **`string`** class provides a number of member functions that are commonly used to search and process strings. The following are among the more useful.

- Find the location of the start of a specified substring – return **`string::npos`** if not found.

```
unsigned sindx = alph.find("fox");// sindx = 16
sindx = alph.find("bat");
bool not_found = sindx == string::npos; // not_found == true
```

- Push a character onto the end of the string.

```
alph.push_back('.');
//alph == "the quick brown fox jumps over the lazy dog.";
```

- Copy a substring starting at **`index`**, of **`n`** characters .

```
string subs = alph.substr(4,  5); // subs == "quick"
                          ^    ^
                       index   n
```

It is emphasized that the full power of the algorithm library can also be applied to the **`string`** through its iterators.

The **`sort`** algorithm

In the last lecture the **`find`** and **`find_if`** algorithms were used to illustrate the interaction between containers, iterators and algorithms. The **`find`** algorithm is fairly easy to implement using a simple **`for`** loop and many programmers would probably not bother with STL to execute a **`find`** operation on a simple container like the **`vector`**.

The situation is much different in the case of algorithms for sorting the contents of a vector. The typical programmer will likely not know how to write an efficient sorting algorithm, and even a simple sorting routine presents significant programming difficulties. Thus most programmers will resort to using the STL **`sort`** algorithm.

The signature of the templated sort algorithm is as follows.

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last)
```

The iterators **`first`** and **`last`** specify the set of elements that are to be sorted. Thus only a portion of the container may be sorted, but typically **`first`** is the iterator returned by **`begin()`** and **`last`** is the iterator returned by **`end()`**.

The operation of **`sort`** is demonstrated in the following simple example.

```
#include <vector>
#include <algorithm>
vector<int> v(5);
v[0]=5;
v[1]=1;
v[2]=10;
v[3]=8;
v[4]=6;
sort(v.begin(), v.end());
//after sort  v[0]=1; v[1]=5; v[2]=6; v[3]=8; v[4]=10
```

With this version of sort[1], it is assumed that the elements of the container can compute the comparison operator **`<`** , i.e. *less than*. Note that the result of **`sort`** is to arrange the elements in ascending order. Also the power of a RandomAccessIterator is necessary to efficiently sort.

Since the **`string`** is a sequence of **`char`** values, it is possible to apply **`sort`** as for any sequence container. From the earlier string example,

```
string alph = "the quick brown fox jumps over the lazy dog";
string::iterator sib = alph.begin();
string::iterator sie = alph.end();
sort(sib, sie);
cout<< alph << endl;
```

The program prints out the following.

```
        abcdeeefghhijklmnoooopqrrsttuuvwxyz
^^^^^^^^
```

The space **`char`** is lower in value than the printing characters, thus the 8 spaces come first in sorted order. To gain more flexibility in sorting, STL provides another version of the sort interface,

[1] Note that it is not necessary to explicitly declare
`sort<vector<int>::iterator>(v.begin(), v.end());`

```
template <class RandomAccessIterator, class
Compare> void sort(RandomAccessIterator first,
                   RandomAccessIterator last, Compare comp)
```

The extra template argument, `Compare`, is used to specify a *predicate* on two container elements that compares them and returns a result that can be cast to a `bool` value, i.e., `true` or `false`. In the final sorted sequence the predicate, `comp` will return `true` for each pair of sequential elements. As an example, this relation holds for an ascending numerical sequence and the comparison provided by **<**, as in the previous example. That is, $\forall_{0 \le i < n-1} v_i < v_{i+1}$.

This more flexible form of `sort` is illustrated by the following example.

```
bool my_comp(unsigned const& a, unsigned const& b) {
  return a>b;
}

int main(){
  vector<int> v(5);
  v[0]=5;
  v[1]=1;
  v[2]=10;
  v[3]=8;
  v[4]=6;
  sort(v.begin(), v.end(), my_comp);
  //after sort v[0]=10; v[1]=8; v[2]=6; v[3]=5; v[4]=1
  ...
}
```

After the sort operation the relation **>** holds between adjacent pairs in the sequence. That is, the sort operation produced a descending sequence.

A templated comparison function can be implemented that provides a generic ordering relation for descending sort operations.

```
template <class T>
bool my_comp(T const& a, T const& b){
  return a>b;
}
...
sort(v.begin(), v.end(), my_comp<unsigned>);
```

This templated comparison function can be applied to any type `T` that implements the **>** operator. Of course if there is a type for which **>** is not implemented, a compiler error will result.

It is also possible to implement a more customized comparison function as shown in the following example, where a sort operation is implemented for a simple 2-d point class.

```
class point_2d {
public:
  point_2d(): x_(0), y_(0){}
  point_2d(double x, double y): x_(x), y_(y){}
  double x_;
  double y_;
};
```

The comparison function is defined on the relative distance of two points, **`a`** and **`b`**, from the origin. The 2-d coordinates are designated as public for ease of illustration. Note that **`less_2d`** is not templated in this example.

```
bool less_2d(point_2d const& a, point_2d const& b) {
  double adist_sq = a.x_*a.x_ + a.y_*a.y_;
  double bdist_sq = b.x_*b.x_ + b.y_*b.y_;
  return adist_sq < bdist_sq;
}
```

This function is used in sorting as follows.

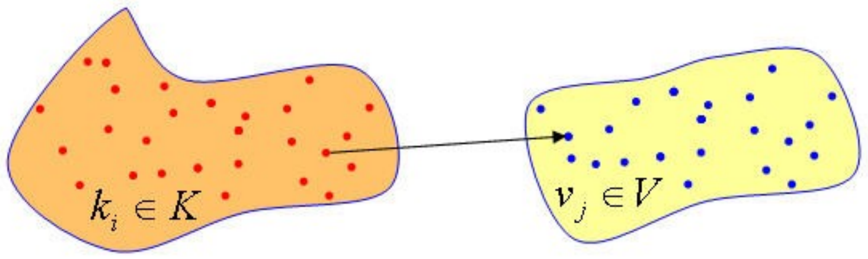
```
point_2d p1(1.0, -1.0), p2(3.1, 1.2), p3(-0.5, 0.75);
vector<point_2d> vp;
vp.push_back(p1);
vp.push_back(p2);
vp.push_back(p3);
sort(vp.begin(), vp.end(), less_2d);
```

After the sort the order of the points in **`vp`** is,

```
vp[0] {x_=-0.50000000000000000 y_= 0.75000000000000000 }
vp[1] {x_= 1.0000000000000000  y_=-1.0000000000000000 }
vp[2] {x_= 3.1000000000000001  y_= 1.2000000000000000 }
```

The associative container, **`set`**

Another category of STL container is the *associative* container, where a *value* data type is accessed through a second data type, the *key*. This arrangement is illustrated in the figure below.



Associative Container

The associative container provides a map between the set of keys and the set of values. The associative container has no concept of accessing a value element at a *position* corresponding to an unsigned integer index with the **`[index]`** operator. Instead, each value element has a associated, unique**,** *key* element by which the value can be accessed. Once assigned, a key is not changed, otherwise the associative link would be lost.

The simplest instance of an associative container is **`set<T>`**. In this case, the container *value* type and *key* type are the same and both equal to **`T`**. Thus, each element of the set is unique, since each value acts as its own key. Therefore unique elements can only be inserted or deleted from **`set<T>.`** Any attempt to assign set values as equal will result in an error.

The set key type **`T`** must support a comparison function, by default **`less<T>`**, that determines the order of a pair of elements. This function is necessary because **`set<T>`** is maintained in a sorted order to facilitate efficient retrieval of queries by *key*. However, this order is not important to enable associative access.

The following is a simple example of using the set.

```
# include <set> set<int> s;
s.insert(3); // s == {3}
s.insert(3); // s == {3} the insert fails since 3 exists
s.insert(4); // s == {3, 4}
set<int>::iterator sit = s.begin();
*sit = 2; // s == {2, 4}
unsigned c = s.count(3);  // c == 0
```

The first **`insert`** adds 3 to the set , **`s`**. The second attempt to insert 3 fails since that element is already present.

The set iterator, **`sit`**, can access each element of the set in sorted order. If the iterator is dereferenced to assign a new value, the effect is the same as removing the existing element, pointed to by the iterator, from the set and inserting the new value. In the example the 3 is removed and 2 is inserted. It is emphasized that the iterator cannot *insert* additional elements, only replace existing elements.

For **`set<T>`**, the **`count`** method returns either 1 or 0, if the element is in the set or not, respectively. In this example, **`s.count(3) == 0`**, after the dereferenced iterator **`sit`** assignment to 2.

An important difference between **`set<T>`** and **`vector<T>`** is that an iterator is not rendered invalid by inserting or deleting elements into the set. For example,

```
s.insert(4);// s == {3, 4}
set<int>::iterator sit = s.begin();// *s == 3
s. insert(1); //  *s == 3 , still accessing element 3
```

This property is useful when a set of unique elements need to be inserted or removed from a container by an iteration loop. The loop iterator is not invalidated by the set operations.

The **`insert_iterator`**

Any type **`T`** can form a **`set<T>`** as long as a comparison operator is defined on pairs of on the elements. In the following example the set contains elements of type **`string`**.

```
# include <string>
# include <set>
# include <iterator>
# include <algorithm>

int main(){
  string a[5]={"rat", "sat","hat", "mat","cat"};
  string b[3]={"bat", "rat", "pat"};
  set<string> sa(a, a+5), sb(b, b+3), sc;

  for(set<string>::iterator ait = sa.begin();
      ait != sa.end();++ait)
    cout << *ait << endl; //alphabetic order

  set<string>::iterator ait = sa.find("rat");
  bool found = ait != sa.end(); // found == true

  insert_iterator<set<string>> iit(sc, sc.begin());

  iit = set_intersection(sa.begin(),sa.end(),
                         sb.begin(),sb.end(), iit);

  for(set<string>::iterator it = sc.begin(); it!=sc.end();++it)
    cout << endl << *ait << endl; // sc == {"rat"}
}
```

The program output follows.

```
cat
hat
mat
rat
sat

rat
```

There are several things to note about this example. First, **`set<string>`** can be constructed from an iterator which is a regular memory pointer to an array of strings, using the following constructor.

```
set<string> sa(a, a+5);//a is the start address, a+5 is the end
```

This construction interface is very natural and one of the powerful aspects of STL.

The second aspect to notice is that the elements of the set are ordered differently than the order in the string memory array. **`set<string>`** has a default comparison predicate, which is lexicographic (alphabetic) order. Thus, the strings are maintained in alphabetic order in the set container.

The third illustration is that the **`find`** method can determine the existence of string elements in the set, just as any other type.

The next new aspect is the line[2],

```
insert_iterator<set<string> > iit(sc, sc.begin());
```

The idea here is that it is not feasible to use the iterator to the set itself, **`set<string>::iterator`**, to insert new elements. As shown above, this iterator should only be used to access set elements. What is needed is an **`OutputIterator`** that can insert new elements into an existing **`set`**. This capability is provided by the appropriately named **`insert_iterator<T>`**. In this case, the type **`T`** must correspond to a container.

In the example, the type is **`T==set<T>`**. So now it is possible to add elements to the set by assigning a value to the dereferenced iterator and then advancing it to add the next element.

Turning away from the example for a second, the **`insert_iterator<T>`** can be used to copy the elements from **`sa`** to **`sc`** as follows.

```
for(set<string>::iterator ait = sa.begin(); ait != sa.end();++ait)
   *(iit++)=*ait;
```

After the loop, the content of **`sc`** is **`sc = {"rat", "sat","hat", "mat","cat"}`**. The effect is that the elements of set **`sa`** have been *inserted* in **`sc`**.

Going back to the example, a new STL algorithm is introduced,

```
iit = set_intersection(sa.begin(), sa.end(),
                       sb.begin(), sb.end(), iit);
```

which performs set intersection on two containers. The first four arguments are the iterator ranges (subsets) of the first set and the second set to be intersected. The fifth argument is an **`insert_iterator`** that will output the elements of the intersection. The return value of **`set_intersection`** is the value of the insertion iterator after the elements of the intersection have been inserted, i.e. one past the end of the range of

[2] The syntax **`<set<string> >`** leaves a space between the terminating **>>** characters. This spacing should always be done, since some compilers erroneously interpret the **>>** as the input stream operator.

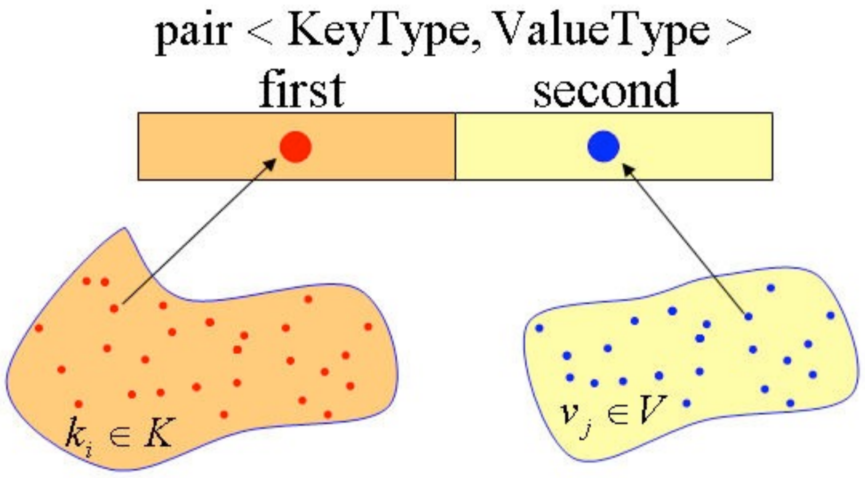
inserted elements. This return value of **`set_intersection`** can usually be ignored unless additional element insertions into the intersection set is necessary, due to further processing.

In the example, the only element in common between **`sa`** and **`sb`** is **`"rat"`** and thus

```
 sc == {"rat"}
```

The **`pair`**

In order to implement more powerful associative containers, it is necessary to have a mechanism for associating a *key* with a *value*. This task is achieved by the STL **`pair`** as shown in the figure below.



The pair is a templated **`struct`** defined as below[3].

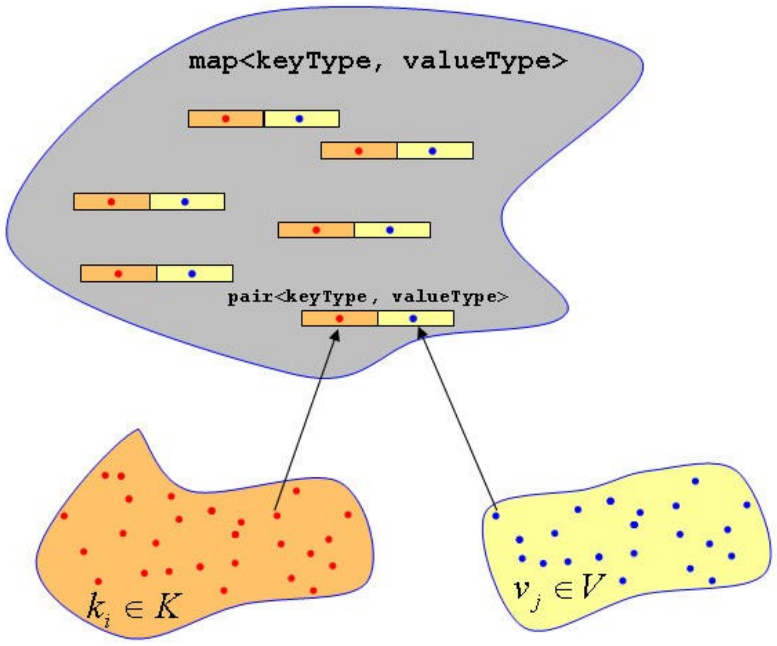
```
template<class Type1, class Type2>
struct pair {
  typedef Type1 first_type;
  typedef Type2 second_type;
  Type1 first;
  Type2 second;
  pair( );
  pair(const Type1& key, const Type2& val);
);
```

The pair has two members, **`first`** and **`second`**. Typically the **`first`** element is a key and the **`second`** element is a value.
 The application of the **`pair`** in the implementation of associative containers will now be illustrated with the STL **`map`** container.

[3] A **`struct`** is almost identical to a **`class`**. The only difference is that its members are **`public`** by default rather than **`private`** by default as for a **`class`**. Many programmers conventionally use the **`struct`** for simple data structures with few methods and reserve the **`class`** for complex data and operations.

The **map**

The **map** is a widely used associative container that supports associating quite different data types in a collection. The following figure illustrates that a **map** is a collection of pairs.



The following program shows how to construct a **map** and insert **pair** elements into the collection.

```
#include <map>// includes pair map<string, double> m;
pair<string, double> p1("bat", 14.75);
pair<string, double> p2("cat", 10.157);
pair<string, double> p3("dog", 43.5);
m.insert(p1);
m.insert(p2);
m.insert(p3);
for(map<string, double>::iterator mit = m.begin();
    mit!=m.end(); ++mit)
  cout << mit->first << ' ' << mit->second << endl;
```

The output of this program is as follows. The **first** member is the *key* and the **second** member is the *value*.

```
bat 14.75
cat 10.157
dog 43.5
```

Note that the **#include <map>** statement also includes the .h file for **<pair>**. In this example, a **string** key is being associated with a **double** value. The **map** has an **insert** method, which inserts the pairs into the collection.

A value can be accessed using the `[]` operator of the `map`. To illustrate this accessor, the following applies the `[]` operator on the `map` above.

```
double x1 = m["cat"];  // x1 == 10.157
```

This example shows the power of an associative map. The key index into the `map` can be any type that can be ordered by a sort operation. In this example, the `string` index is lexicographically ordered.

A *key* is immutable, as for the `set`, but the associated *value* can be changed. For example,

```
// change the value of pair with key "dog" to 7.21
m["dog"] = 7.21;
map<string, double>::iterator mit = m.begin();
mit++; mit++; // advance to the third pair
cout << mit->first << ' ' << mit->second << endl;
```

The output of the program is as follows.

```
dog 7.21
```

The `size()` method applies to `map<T1, T2>` as well. In the example,

```
unsigned n  = m.size(); // n ==3, the number of pairs in the map
```

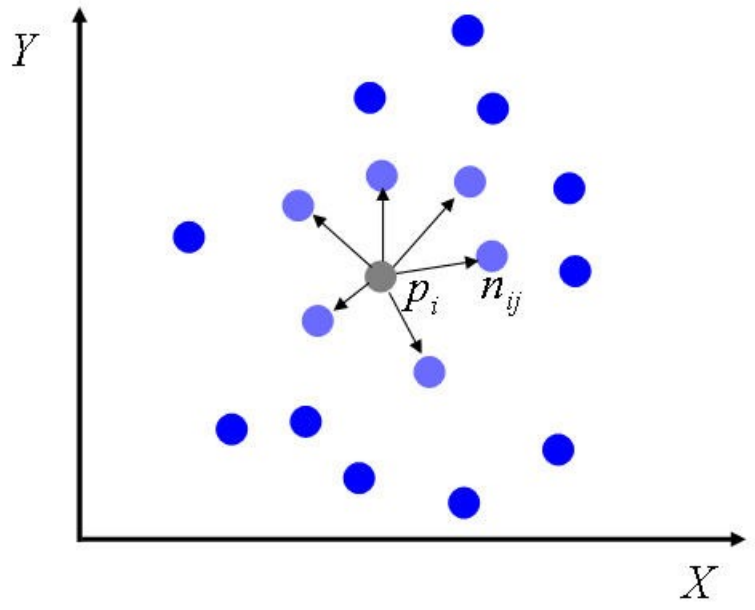
Using the `map` to maintain a neighborhood

In order to illustrate the power of the `map` in a more realistic application setting, an implementation of a `point` neighborhood will be developed. A set of 2-d points is shown below.

In many physical problems it is essential to know the set of nearby points in a point set. For example if the points are repelled from each other by force inversely proportional to the square of the distance, e.g. electric force, then it is necessary to know the set of closest points to determine an approximation to the total force on a given point, in order to determine its subsequent motion.

It is therefore essential to devise a data structure for point set and its neighborhoods that supports efficient retrieval of the points in an order related to their proximity.

The STL library provides an effective set of tools for accomplishing this task, including the `map`.



The figure above shows a point p_i and its set of neighboring points, n_{ij}.

To study the issues of computing and maintaining the point neighborhoods, the following implementation of the point class will be used.

```
class point {
  public:
    point(): x_(0), y_(0){}
    point(double x, double y): x_(x), y_(y){}
    double x() const {return x_;}
    double y() const {return y_;}
    bool operator < (point const& p) const {return x_<p.x();}
  private:
    double x_;
    double y_;
};
```

This implementation is similar to previous designs except for the **<** operator.
Since a point an element of the 2-d plane, there is no natural ordering predicate, **<**.
In this example, the ordering relation between two points is based on their **x** coordinates.

Consider the following code fragment where points are inserted into a **set**.

```
#include <vector>
#include <set>
#include <iostream>
//A set of 14 points taken from the figure above

vector<point> p0; //the point set as a vector
p0.push_back(point(0.5,1.4));
p0.push_back(point(1.0,0.6));
p0.push_back(point(1.1,1.1));
p0.push_back(point(1.2,1.6));
p0.push_back(point(1.4,1.2));
p0.push_back(point(1.4,1.7));
p0.push_back(point(1.3,2.1));
p0.push_back(point(1.6,0.75));
p0.push_back(point(1.8,1.3));
p0.push_back(point(1.75,1.6));
p0.push_back(point(2.0,2.0));
p0.push_back(point(1.8,2.3));
p0.push_back(point(2.3,1.3));
p0.push_back(point(2.3,1.6));

// insert in a set
set<point> sp(p0.begin(), p0.end());
unsigned i = 0;
for(set<point>::iterator sit = sp.begin();
    sit != sp.end(); ++sit, ++i)
  cout << "p[" << i << "]=(" << sit->x() << ' '
       << sit->y() << ')' << endl;
```

```
p[0]=(0.5 1.4)
p[1]=(1 0.6)
p[2]=(1.1 1.1)
p[3]=(1.2 1.6)
p[4]=(1.3 2.1)
p[5]=(1.4 1.2)
p[6]=(1.6 0.75)
p[7]=(1.75 1.6)
p[8]=(1.8 1.3)
p[9]=(2 2)
p[10]=(2.3 1.3)
```

There are three aspects to notice about this result:

1. the points in **`set<point> sp`** are indeed ordered by the **`x`** coordinate value.

2. This order comes about due to the default behavior of the **`set`** declaration. A second, default template argument on **`set`**, is the templated order predicate, **`less<T>`**. This predicate in turn applies the **`<`** operator, implemented for type **`T`**. In other words,

   ```
   set<point, less<point> > sp(p0.begin(), p0.end());
   ```

   is an equivalent constructor[4].

3. Three points are missing from the set! They are missing because there was already a point in the set with the same **`x`** coordinate. For example, the insert of (1.4, 1.7) failed because (1.4, 1.2) was already in the set. That is, the possibility of duplicate keys in a set is nonsensical.

[4] Note that there is also defined a **`greater<T>`** predicate function that applies the **`>`** operator on the type **`T`**, i.e., **`set<point, greater<point> > sp(p0.begin(), p0.end());`**

In order for an element, *k*, to be inserted in a set it must be true that:

1. the set is empty;
2. or there is an element *e* in the set such that *e*<*k* || *k*<*e*.

This problem of these missing elements can be fixed by changing the order predicate of the point class to the following.

```
bool operator < (point const& p) const
  {if(x_!=p.x())return x_<p.x(); else return y_<p.y();}
```

Points are now ordered first by **x**, and in case of a tie then by **y**. The result of the insertion into **set<point> sp** is shown below.

```
p[0]=(0.5 1.4)
p[1]=(1 0.6)
p[2]=(1.1 1.1)
p[3]=(1.2 1.6)
p[4]=(1.3 2.1)
p[5]=(1.4 1.2)
p[6]=(1.4 1.7)
p[7]=(1.6 0.75)
p[8]=(1.75 1.6)
p[9]=(1.8 1.3)
p[10]=(1.8 2.3)
p[11]=(2 2)
p[12]=(2.3 1.3)
p[13]=(2.3 1.6)
```

All the points were successfully inserted into the set, which now ordered first by **x** and then by **y**.

Customizing the ordering predicate

There is still a fundamental problem with the ordering predicate. What is desired is an ordering that applies the relative distance from a given point to its neighbors as the ordering relation. The **<** predicate on the point class above is based on absolute point coordinates, not relative distance to a given point.

A first thought is to create a customized order predicate such as,

```
bool d_less(point const& pa, point const& pb);
```

Unfortunately, there is no way to input a focus point, **pi**, from which the distances of **pa** and **pb** are compared by their distance from **pi**. The predicate function signature requires exactly two **point** arguments, and a function with more arguments will not be accepted by the compiler.

The designers of STL anticipated this difficulty and allow the definition of a class with **`operator ()`** to be overloaded as an ordering predicate. Such a class is called a *function object*, or a *functor*.

To illustrate this approach, a global function that computes the distance between two points is required and is implemented as follows.

```
#include <math.h>
double d(point const& pa, point const& pb) {
   double dsq =
     (pa.x()- pb.x())*(pa.x()- pb.x()) +
     (pa.y()- pb.y())*(pa.y()- pb.y());
   return sqrt(dsq);
}
```

Next, a function object, **`p_less`**, is implemented that holds the point that is to be used as the reference (focus) for relative distance calculations.

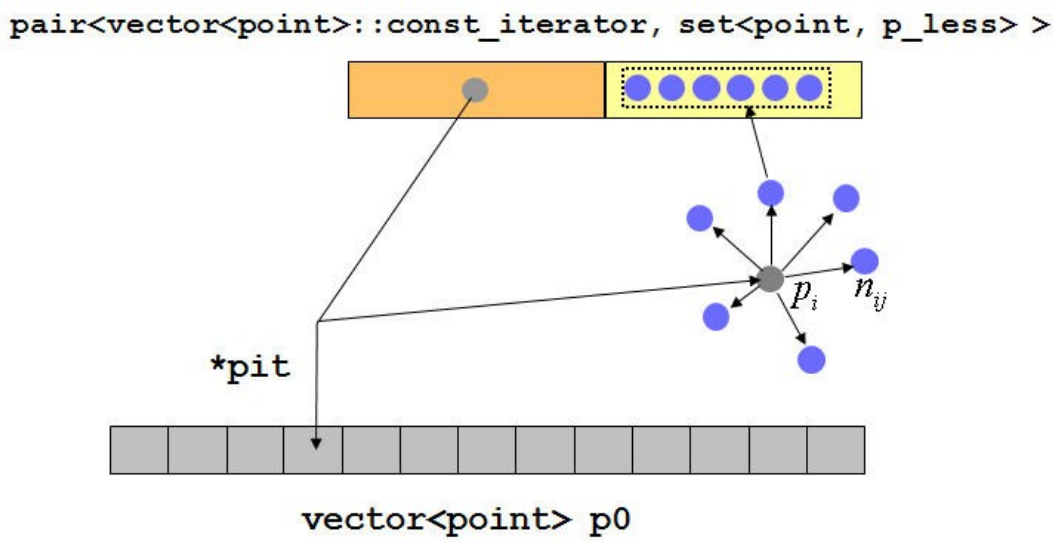
```
class p_less {
 public:
  p_less(point const& p): focus_(p){} // constructor with focus
  p_less(): focus_(point()){} // default constructor,focus_==(0,0)
  // the predicate function
  bool operator()(point const& pa, point const& pb) const {
   return d(pa, focus_) < d(pb, focus_);
  }
 private:
  point focus_;
};
```

There are a number of aspects of this implementation that require explanation.

The key method on **`p_less`** is the **`()`** operator that will be called by STL algorithms and containers to provide an ordering relation on elements.

In this example the class **`p_less`** maintains the member **`focus_`** that is the point relative to which distances are to be ordered. The order relation is based on the relative distance of two points to the *focus* point. This function object can be used as an order predicate in STL algorithms and associative containers.

The neighborhoods of each point can be implemented as shown in the figure below.



The idea is to store the point set as a **`vector<point>`** and use an iterator to this vector as the key for accessing the point neighborhood. If the vector is modified, then the iterator could become invalid, so this approach only works if the elements of the point vector are unchanged during operation.
In other implementations, the key could be a memory pointer to points stored on the heap, or even to a **`point`** type itself.

This code fragment shows the creation of a neighborhood for each point in the **`vector<point>`** , **`p0`**.

```
vector<point> ps=p0; // make a copy for inserting neighborhoods
// the map between a point and its neighborhood
map<vector<point>::const_iterator, set<point, p_less> > mp;
// iterate through the point set to construct the neighborhoods
for(vector<point>::const_iterator pit=p0.begin();
    pit!= p0.end(); ++pit) {
  // the p_less function class instance pl(*pit) order relation
  p_less pl(*pit);
  // The neighborhood with p_less(*pit) order
  set<point, p_less> nbrhd(pl);
  //extract the neighborhood of pit
  double t = 1.2; // the threshold on distance
  vector<point>::iterator nit = ps.begin();
  for(; nit!=ps.end()&&d(*nit, *pit)<t ; ++nit)
    if(!( (*nit) == (*pit) )) // skip focus point
      nbrhd.insert(*nit); // will automatically sort on insertion
  // insert the association between pit and its neighborhood into mp
  pair<vector<point>::const_iterator, set<point,p_less>> pr(pit,nbrhd);
  mp.insert(pr);
}
```

The point neighborhoods are maintained by the map, **`mp`**, which holds a collection of pairs shown in the figure above.

```
map<vector<point>::const_iterator, set<point, p_less> > mp;
```

The **`first`** element of the map pair is a **`const_iterator`** to the point set stored in **`vector<point> p0`**. Thus, the map provides an association between a given point and its neighborhood. The **`second`** element of the map pair is a **`set<point, p_less>`** which maintains the point neighborhood in sorted order relative to the focus point. The focus point is the **focus_** member of the particular instance of **p_less** used to construct the neighborhood.

A copy, **ps**, of **vector<point> p0** is made to avoid invalidating the loop iterator ***pit** to **vector<point> p0**. **ps** is used to access points for insertion into the neighborhood within the loop.

The point neighborhood is extracted by filling **set<point, p_less> nbrhd** with neighbors of the focus point. Points are automatically inserted in sorted order by the **p_less** functor.

Points are not allowed into the neighborhood if the exceed a given distance threshold, **t**, from the focus point.

Using the map

The following code uses the map just constructed to display the neighborhood.

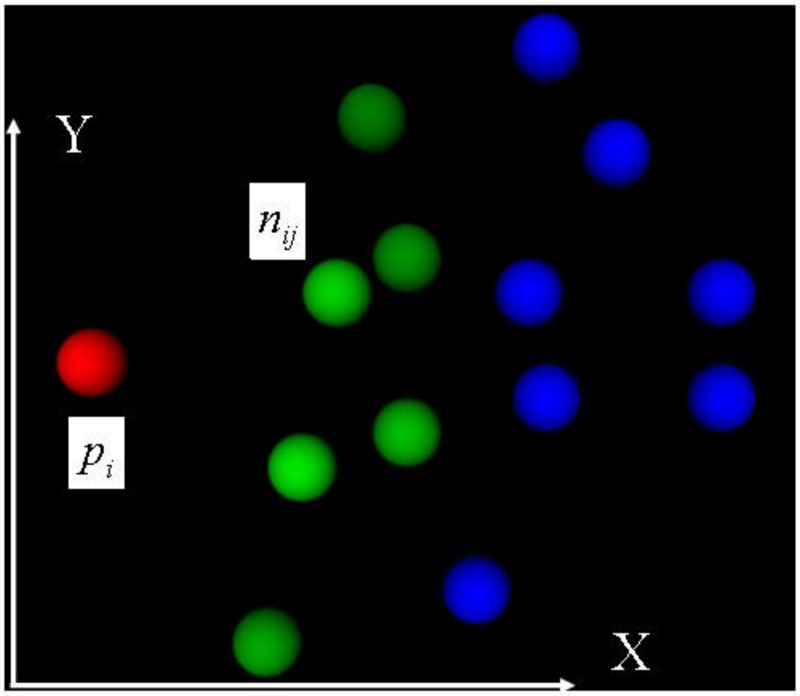
```
#include "vrmlFunctions.h" // from previous assignment
ofstream os("nbh.wrl"); //the vrml file
write_vrml_header(os);
vector<point>::iterator it = p0.begin();
sphere s0(it->x(), it->y(), 0, 0.1);
write_vrml_sphere(os, s0, 1, 0, 0, 0); // red focus sphere
set<point, p_less> nb = mp[it]; // note the [] key access to the map
// Add members of nb to the display
float sm = 0.5f/nb.size(), sj = 1.0f; // brighness of neighbors
for(set<point, p_less>::iterator nit = nb.begin(); nit!= nb.end(); ++nit) {
  sphere si(nit->x(), nit->y(), 0, 0.1);
  write_vrml_sphere(os, si, 0, sj-=sm , 0, 0);//green
}
// Add members of p0 not in nb to the display
for(vector<point>::iterator vit = p0.begin(); vit!= p0.end(); ++vit) {
  if(*vit == *it)
    continue;
  set<point, p_less>::iterator fit = nb.find(*vit);
  if(fit != nb.end())
    continue;
  sphere si(vit->x(), vit->y(), 0, 0.1);
  write_vrml_sphere(os, si, 0, 0 , 1, 0); // blue
}
```

This code accesses the map to find the neighborhood of the first point in **p0**. The access has a similar syntax to accessing an array, i.e.,

```
set<point, p_less> nb = mp[it];
```

It is emphasized that, while the syntax is similar to accessing a vector element, the **[key]** operator is acting on the keys of the map, not on an integer vector index. If the key, **it**, is not in the map, a crash will occur. This result is similar to accessing a vector with an invalid index.

The resulting display of the output vrml file, `nbh.wrl`, is shown below.



The red sphere is the *focus* point for with the neighborhood is being displayed. The green spheres are the points in the neighborhood of the red sphere, i.e., n_{ij}. The brightness of the green indicates the order of the point in the set, brightest first. Note that, as expected, the set order is the same as distance order from p_i. The blue spheres are in the point set **`p0`** but not neighbors of p_i