Standard Template Library

- ADT's and Implementations
- C++ and templates revisited
- STL philosophy and goals:
  - generic algorithms
  - efficiency
- STL components:
  - containers
  - algorithms
  - iterators
- STL guided tour

ADT's and Implementations

- In the last homework, we implemented some Abstract Date Types (ADT)
- We learned that the *implementation* is not the same as the ADT
  - The ADT determines the *properties*, or allowable operations
  - A particular implementation may be efficient for some operations, and inefficient for others.
    - implementing a list using an array makes binary search $O(\log N)$, but insertion $O(N)$.
    - implementing a list using doubly-linked nodes makes binary search $O(N)$, but insertion $O(1)$.

  which is better? It depends which operation is done more often.

- We have one choice: for a given ADT, we could pick an implementation that is most efficient for the specific task.
- There is another choice: we could fix the implementation, and choose the most appropriate ADT for the specific task. E.g. if a list is implemented with doubly-linked nodes, and a vector with a (smart) array:
  - if we'll be doing a lot of binary searches, choose the vector
  - if we'll be doing a lot of insertions (and delete), choose the list

C++ and templates revisited

A reminder of templates:
Templates come in 2 flavors:

1. Template Functions
   The arguments to a function can be template arguments, allowing the compiler to stamp out the appropriate signature.

   In the following example, note that `operator?:` must be defined for type `T`.

```
#include <iostream.h>   // TemplateFunction.cc

template <class T>
T min(const T& x, const T& y) { return (x<y) ? x : y; }

int main() {
  cout << "Keep entering pairs of integers: " << ends;
  int a,b;
  while ( cin >> a >> b ) {
    cout << "min("<<a <<","<<b<<") = "<< min(a,b) << endl;
  }
}
```

2. Template Classes

   Classes can be defined with template arguments, allowing the compiler to stamp out specific classes.

```
#include <String.h>   // TemplateClass.cc
#include "TemplateClass.hh"

int main() {
  Pair<int, String> a(43, "Hello");
  Pair<float, char> b(3.14, 'x');
  Pair<Pair<int, String>, long> c(a, 1234567890);
  cout << a <<"\n"<< b <<"\n"<< c << endl;
}
```

```
#ifndef __TEMPLATECLASS_HH   // TemplateClass.hh
#define __TEMPLATECLASS_HH
#include <iostream.h>

template <class T1, class T2>
class Pair {
public:
  Pair(const T1& t1,const T2& t2) : m_first(t1), m_second(t2){}
  T1 first()  const { return m_first; }
  T2 second() const { return m_second; }
  friend ostream& operator<<(ostream&, const Pair<T1,T2>&);
private:
  T1 m_first;
  T2 m_second;
};
template <class T1, class T2>
ostream& operator<<(ostream& os, const Pair<T1,T2>& p) {
  os <<"("<<p.m_first<<", "<<p.m_second<<")"; return os; }
#endif  //  __TEMPLATECLASS_HH
```

Note:

- The class can have several comma-separated arguments
- Each class type (`classname<T, U, ...>`) generates a distinct class
- Templates can be nested – since `Pair<int, String>` is just a type.
- Operators such as `==` etc. (if used) must be defined for `class T`.

More Templates

- Templates support default parameters (g++ does, but not all compilers do). The syntax is similar to constructor default arguments.

```
template <class T, class U=Foo>
```

- A template class can have a template member function (it's in the standard, but g++ does not yet support).

Boolean Type

The C++ ANSI standard specifies a Boolean – type `bool`, that is defined in `bool.h` (it just uses `enum` to define `true` and `false`).

```
#include <bool.h>   // Boolean.cc
#include <iostream.h>

int main() {
  bool a( (17>42) );
  bool b( !a );
  cout << "a, b: " << a << ", " << b << endl;
}
```

Function Objects

- We have seen the components – but not given it a name.
- It is convenient to define a class consisting of just the overloaded function call operator, `operator()()` (and possibly some data), rather than using function pointers.
- An instance of such a class is called a *Function Object*.

```
#include <bool.h>   // FunctionObject.cc
#include <iostream.h>

template <class T>
class greater {
public:
  bool operator()(const T& x, const T& y) const { return (x>y); }
};

template <class T, class Predicate>
void print(const T& x, const T& y, const Predicate& p) {
  cout<<"x,y: "<<x<<", "<<y<<"  Predicate: "<<p(x,y)<<endl;
}

int main() {
  print(17, 42, greater<int>());
  print(3.14, 2.7, greater<double>() );
}
```

Points to note:

1. class `greater` has no data – so use the default constructor.
2. class `greater` is templatized (tho it needn't be). It assumes the existence of `operator>` for `class T`.
3. we overload `operator()`, which returns a `bool`
4. in the call to `print`, we pass a (temporary) instance of a `greater` object as an argument – using the default constructor.
5. function `print` uses the predicate like any other object.

Why use function objects?

- the function object is resolved at compile time
- the code can be inlined – improving efficiency for small functions
- the function can use member data

STL philosophy and goals

- A goal of STL is to standardize software components – Software IC's.
- But it also has to be efficient:
  - efficient in the *implementation* of an algorithm – within a few % of assembler code.
  - efficient in the *choice* of algorithm – e.g. if the best we can do is $O(N \log N)$, STL must be no worse.
- Algorithms should be *generic* – not dependent on the actual data structure. Algorithms and data structures are *orthogonal*.
- But cannot compromise efficiency for some particular data structures.

Are these goals mutually consistent?

generic algorithms

- Is it better to have a different algorithm (e.g. *sort*) for each data structure? Perhaps we could optimize the efficiency?
- Disadvantages of this approach:
  - more difficult to extend (and maintain)
  - interfaces more complicated – dependent on data structure
- STL algorithms are generic (i.e. the same for all data structures).
  - if this is applied too rigidly, we would lose efficiency
  - remember that $O(N \log N)$ means $cN \log N$ – different algorithms have a different c
  - the algorithm is chosen to be the *most* efficient – we then do differently for data structures that cannot support this

E.g. on *average*, quicksort is $O(N \log N)$, but the worst case is $O(N^2)$. Heapsort is guaranteed $O(N \log N)$, but with a different c – c(heapsort) $\simeq 2 \times c$(quicksort)
So STL provides both.

- *But* both get efficiency from using random access – which won't work with lists. So lists have their own sort (member function) – which is still $O(N \log N)$, but not as efficient as the generic sort.

efficiency

- STL's efficiency is a corollary of being generic:
  - always use the *most* efficient algorithm
  - if that is not possible with some particular data structure, then use a restricted algorithm for *that* data structure.
- Efficiency is also gained from C++ language features: templates, function objects, inlining, etc.
- Efficiency arises from the choice of STL components, and their inter-relations.

STL components:

The key components of STL are:

- Containers – the data structures, or implementations of the ADT.
- Algorithms – the operations performed on the containers – e.g. `sort`, `find`, etc.
- Iterators – the means to traverse a data structure so as to implement a generic algorithm.

There are additional components that add to the versatility:

- Function Objects – extend a relation or predicate
- Adaptors – extend a container, iterator, or function
- Allocators – extend a particular memory model

Containers

- An STL container is a C++ container template class that holds a *sequence* of items of type `T`.
- The STL container is the implementation of the ADT
- STL provides several containers – others can be based on these:
  - Vector
  - Deque
  - List
  - Set and Multiset
  - Map and Multimap

Algorithms

STL provides classes of generic algorithms for operations on containers:

- copy
- sort
- find
- fill
- partition
- insert, delete
- set operations (union, intersection)
- accumulate

Iterators

Iterators are the glue of STL that makes it possible to use generic algorithms and orthogonalize those algorithms from the data structures.

Definition: an iterator, `i`, is a generalized means of traversing a data structure.

E.g. for an array, an array index, or pointer, is an iterator.

- An iterator is also a “smart pointer”.
- Dereferencing an iterator, `*i`, is guaranteed to give the item, but in general, an iterator does not obey all pointer operations.

- All STL containers have iterators – but the iterator algebra depends on the container. E.g. all containers support `++i`, but `list` does not support a long jump, `i+n`.
- All STL containers handle iterators in the same, consistent way. For any container `c`, of type `T`,
  - `container<T>::iterator i=c.begin()` points to the *first* item in the sequence
  - `container<T>::iterator j=c.end()` points *beyond* the *last* item in the sequence
  - `j` is said to be *reachable* from `i`, *iff* there is a finite sequence of `operator++` that makes `i==j`
  - If `j` is reachable from `i`, then `i` and `j` refer to the same container

Why does `container<T>::iterator j=c.end()` point *beyond* the last item in the sequence? (and not *to* the last item)

- To test for the end of a sequence, *only* `operator!=` is needed (and not `operator>` which is not defined for all containers).

There are other reasons which affect convenience that we will see later.

Notation

- Denoting the iterator range by `first` and `last`, the range is written:

  $$[\text{first}, \text{last})$$

  meaning that `first` is *included* in the range, but `last` is not.

- The range is *valid* if `last` is reachable from `first`. The result of an algorithm on an invalid range is undefined.

- if `first==last`, the range is *empty*, but valid.

Iterator Hierarchy

To ensure that the algorithms operate on the appropriate containers, there is an iterator hierarchy. (we do not have to remember the restrictions – the compiler saves us from ourselves.)

1. input iterators
2. output iterators
3. forward iterators
4. bidirectional iterators
5. random access iterators

An algorithm that works with one iterator will always work with a container supporting a *higher* iterator, but not vice versa.

STL guided tour

To see how it all works, let's do some examples.

Let's populate a `vector`, shuffle it, then sort it.

```
#include <vector.h>   // example01.cc
#include <algo.h>
int main() {
  vector<int> a;
  ostream_iterator<int> out(cout, " ");
  for (int i=0; i!=20; a.push_back(i++)) {}
  copy(a.begin(), a.end(), out);   cout << endl;
  random_shuffle(a.begin(), a.end());
  copy(a.begin(), a.end(), out);   cout << endl;
  sort(a.begin(), a.end());
  copy(a.begin(), a.end(), out);   cout << endl;
}
```

Points to note:

1. We need the header files `vector.h` and `algo.h`
2. We declare a `vector` of `int` with no items.
3. We use the member function `push_back()` to add items to the *back* of the vector.
4. We declare `out` to be of type `ostream_iterator<int>` – this is an ostream (output) iterator
5. The 2nd argument in the `ostream_iterator` constructor is a string to place between successive values on the output stream.
6. The `ostream_iterator` allows us to write *to* the stream, but not read *from* it.

7. The iterator only supports `operator++`. Once we have passed a value, we cannot write to that position in the stream again.

8. The `copy` function (a generic algorithm) copies items from the vector to the output stream.

9. The `random_shuffle` function (a generic algorithm) randomizes the vector.

10. The `sort` function (a generic algorithm) then sorts the vector in place.

11. Both `sort` and `random_shuffle` take iterators of type `RandomAccessIterator` as arguments, so cannot work with lists.

12. But this *will* work with other containers such as `deque`

```
#include <algo.h>   // example02.cc
#include <deque.h>

int main() {
  deque<int> a;
  ostream_iterator<int> out(cout, " ");
  for (int i=0; i!=20; a.push_back(i++)) {}
  copy(a.begin(), a.end(), out);   cout << endl;
  random_shuffle(&a[0], &a[a.size()]);
  copy(a.begin(), a.end(), out);   cout << endl;
  sort(a.begin(), a.end());
  copy(a.begin(), a.end(), out);   cout << endl;
}
```

Some of the STL implementation (e.g. `deque`) looks buggy. Is this STL or g++?

Why won't `random_shuffle` work with lists?

- `random_shuffle` works in *linear* time for a random access iterator.
- To work with lists, it would have to be $O(N^2)$.
- Since the *best* it can be is $O(N)$, STL only allows those iterators which are $O(N)$.

Is this restrictive? STL places *efficiency* above generality.

As well as an output iterator, there is (not surprisingly) an input iterator. It has 2 constructors:

- `istream_iterator(istream& in)` – constructs an `istream_iterator` object that reads values from the input stream `in`.
- `istream_iterator()` – constructs the end of stream iterator value.

```
#include <vector.h>   // example03.cc
#include <algo.h>

int main() {
  vector<float> a;
  istream_iterator<float> eos;
  ostream_iterator<float> out(cout, " ");
  cout << "Enter some floats, ^D to end" << endl;
  for ( istream_iterator<float> in(cin); in!=eos; ++in ) {
    a.push_back(*in);
  }
  copy(a.begin(), a.end(), out);   cout << endl;
  sort(a.begin(), a.end());
  copy(a.begin(), a.end(), out);   cout << endl;
}
```

Points to note:

- The `istream_iterator` allows us to read *from* the stream, but not write *to* it.
- The iterator only supports `operator++`. Once we have passed a value, we cannot read from that position in the stream again.
- The end of stream iterator, `eos`, allows us to read to the end of stream.
- Using `a.push_back(*in)`, we add to the end of the vector.
- If `a` is empty, `a.end()` and `a.begin()` both point to the beginning of the vector.

more on iterators

Now we've seen how containers, algorithms, and iterators work together, we can categorize the iterators:

1. **Input Iterators.**
   We can see the requirements for input iterators by coding the `find` algorithm:

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

This code implies that the following operations are required:

- `operator!=` to test termination (note that this is less restrictive than other tests)
- `operator++` for prefix incrementing
- `operator*` for iterator dereferencing

For `find` to work *efficiently*, each of these operations must work in constant time.

In addition, `class InputIterator` requires:

- `operator++` (postfix) – implemented in terms of prefix
- `operator==`

These requirements are also met by built-in pointer types, but built-in pointer types also have *additional* properties. Therefore built-in pointer types can serve as input iterators.

```
#include <list.h>   // example04.cc
#include <algo.h>
#include <assert.h>

int main() {
  const int a[]={0,4,6,7,4,2,3,89,12,34};
  const int kArraySize(sizeof(a)/sizeof(int));
  const int* pa=find(a, a+kArraySize, 89);
  assert( *pa==89 && *(pa+1)==12 );
  list<int> list1(a, a+kArraySize);
  list<int>::iterator i=find(list1.begin(), list1.end(), 89);
  assert( *i==89 && *(++i)==12 );
}
```

Points to note:

- the value of `a` or `&a[0]` clearly points to the beginning of the array.
- the value of `a+10` or `&a[10]` points beyond the last item.
  - `last` is never dereferenced, so that's not a problem
  - `a+10` is clearly reachable from `a` by repeated application of `operator++`

2. **Output Iterators.**

As well as the “obvious” difference between input and output iterators, there are also subtle ones:

- for class `InputIterator`, we can use `foo=*in` (as we did in an earlier example)
- for class `OutputIterator`, we can use `*out=...` but cannot dereference `out`
- since there is no equivalent of `eos`, we do not need `==` or `!=`

```
#include <iterator.h>   // example05.cc

int main() {
  ostream_iterator<int> out(cout, "\n");
  *out = 37;
}
```

More generally, we can code the `copy` algorithm to illustrate the iterator requirements:

```
template <class InputIterator, class OutputIterator>   // copy.cc
OutputIterator copy(InputIterator first, InputIterator last,
    OutputIterator result)
{
  while (first != last) { *result++ = *first++; }
  return result;
}
```

---

which only needs `++` (postfix and prefix)

3. **Forward Iterators.**

Forward Iterators have all the properties of Input and Output Iterators *plus*:

- they can be used in *multipass* algorithms

Let's look at the `replace` algorithm:

```
template <class ForwardIterator, class T>   // replace.cc
void replace(ForwardIterator first, ForwardIterator last,
     const T& old_value, const T& new_value) {
  while (first != last) {
    if (*first == old_value) *first = new_value;
    ++first;
  }
}
```

```
#include <algo.h>   // example06.cc
#include <vector.h>

int main() {
  const int a[]={0,4,6,7,4,2,3,89,12,34};
  vector<int> b(&a[0], &a[(sizeof(a)/sizeof(int))]);
  ostream_iterator<int> out(cout, " ");
  copy(b.begin(), b.end(), out); cout<<endl;
  replace(b.begin(), b.end(), 4, 23);
  copy(b.begin(), b.end(), out); cout<<endl;
}
```

4. $\boxed{\textbf{Bidirectional Iterators.}}$

Surprise, surprise! Bidirectional Iterators support all the properties of Forward Iterators *plus*:

- they must have the `--` operator (prefix and postfix)

so a sequence can be traversed in the reverse direction

```
#include <algo.h>   // example07.cc
#include <list.h>

int main() {
  const int a[]={0,4,6,7,4,2,3,89,12,34};
  list<int> b(&a[0], &a[(sizeof(a)/sizeof(int))]);
  ostream_iterator<int> out(cout, " ");
  copy(b.begin(), b.end(), out); cout<<endl;
  reverse(b.begin(), b.end());
  copy(b.begin(), b.end(), out); cout<<endl;
}
```

5. $\boxed{\textbf{Random Access Iterators.}}$

Finally, Random Access Iterators ensure that any position in a sequence can be reached from any other position in *constant time*.

- Random Access Iterators must support a long jump: `a.begin()+n`

E.g. binary search works in $O(\log N)$ time on an *ordered* sequence *iff* the sequence supports Random Access Iterators.

```
#include <algo.h>   // example08.cc
#include <vector.h>
#include <assert.h>

int main() {
  const int a[]={0,4,6,7,4,2,3,89,12,34};
  vector<int> b(&a[0], &a[(sizeof(a)/sizeof(int))]);
  sort(b.begin(), b.end());
  ostream_iterator<int> out(cout, " ");
  copy(b.begin(), b.end(), out); cout<<endl;
  assert( binary_search(b.begin(), b.end(), 89) );
}
```

constant iterators

Finally, finally: all iterators also come in a *constant* version for traversing a constant container.

Note:

- a `const_iterator i` *can* be changed
- but `*i` *cannot* be changed

```
#include <algo.h>   // example09.cc
#include <vector.h>
#include <assert.h>

int main() {
  const int a[]={0,4,6,7,4,2,3,89,12,34};
  const vector<int> b(&a[0], &a[(sizeof(a)/sizeof(int))]);
  vector<int>::const_iterator i = b.begin()+3;
  assert( *i==7 && *++i==4 );
}
```

Algorithms

- This is *not* a comprehensive tour thru the STL algorithms
- Look in `algo.h` for the complete story
- We've already met some of the STL algorithms
- Don't be fooled: STL algorithms (together with function objects) are *very* comprehensive

Let's first remind ourselves of the `sort` algorithm:

```
#include <vector.h>   // example10.cc
#include <algo.h>

int main() {
  vector<int> a;
  ostream_iterator<int> out(cout, " ");
  for (int i=0; i!=20; a.push_back(i++)) {}
  random_shuffle(a.begin(), a.end());
  sort(a.begin(), a.end());  // ascending sort
  copy(a.begin(), a.end(), out);
  cout << endl;
}
```

Suppose we want to sort in *decreasing* order?

- We could first sort, and then `reverse_copy` – but that is inefficient.
- STL could give a `descending_sort` algorithm – but that's not very flexible.
- Instead, we use function objects: `sort` can take a 3rd argument – the function object.

We could write our own function objects – but STL already provides a family of (template) classes.

- An ascending sort uses the `<` operator
- A descending sort must use the `>` operator – with the `greater<T>()` function object.

```
#include <vector.h>   // example11.cc
#include <algo.h>

int main() {
  vector<int> a;
  ostream_iterator<int> out(cout, " ");
  for (int i=0; i!=20; a.push_back(i++)) {}
  random_shuffle(a.begin(), a.end());
  sort(a.begin(), a.end(), greater<int>());  // descending sort
  copy(a.begin(), a.end(), out);
  cout << endl;
}
```

In this case, the signature for `sort` was different with a function object, so there is no ambiguity.

To use a function object with `find` (to find a value based on a predicate), the usual signature is:

```
InputIterator find(InputIterator first,
                   InputIterator last, const T& value)
```

and since a predicate function object is just a class, this signature would *not* be unique.

So STL provides the `find_if` function:

```
InputIterator find_if(InputIterator first,
                      InputIterator last, Predicate pred)
```

```
#include <algo.h>   // example12.cc

template <class T>
class myGreater {
public:
  bool operator() (const T& x) const { return (int)x > 48; }
};

int main() {
  int a[]= {12,31,45,17,21,67,8,96,13};
  int len= sizeof(a)/sizeof(int);
  cout << *find_if(a, a+len, myGreater<int>()) << endl;
}
```

There is one algorithm that does an internal traversal of a container – without requiring an external iterator:

```
Function for_each(InputIterator first,
                  InputIterator last, Function f)
```

this applies the function object `f` to *each* element of the sequence.

```
#include <algo.h>   // example13.cc
#include <list.h>
#include <math.h>

class printSqrt {
public:
  void operator() (double x) const { cout << sqrt(x) << endl; }
};

int main() {
  list<double> a;
  for (int i=0; i!=10; a.push_back(++i)) {}
  for_each(a.begin(), a.end(), printSqrt());
}
```

Another use of a predicate is to partition a sequence: all elements of the sequence satisfying the predicate are placed before those that do not.

- `partition` does *not* guarantee to preserve the order of each subset
- `stable_partition` *does* guarantee to preserve the order of each subset

```
#include <algo.h>   // example14.cc
#include <vector.h>

class myPredicate {
public:
  bool operator() (double x) const { return x>2.0; }
};

int main() {
  const int kArraySize=10;
  vector<float> a(kArraySize);
  for (int i=0; i!=kArraySize; i++) { a[i]=(i+0.5)/3.14; }
  partition(a.begin(), a.end(), myPredicate());
  ostream_iterator<float> out(cout, "\n");
  copy(a.begin(), a.end(), out);
}
```

```
#include <algo.h>   // example15.cc
#include <vector.h>

class myPredicate {
public:
  bool operator() (double x) const { return x>2.0; }
};

int main() {
  const int kArraySize=10;
  vector<float> a(kArraySize);
  for (int i=0; i!=kArraySize; i++) { a[i]=(i+0.5)/3.14; }
  stable_partition(a.begin(), a.end(), myPredicate());
  ostream_iterator<float> out(cout, "\n");
  copy(a.begin(), a.end(), out);
}
```

The `remove` function is worth noting:

- it removes an element, changing the value of the `last` iterator
- but it does *not* change the size of the container
- so if M elements are removed, *at least* M can be added before increasing the size of the container
- the return value is the iterator for the *new* end position

Not surprisingly, there is also a `remove_if` algorithm

```
#include <algo.h>   // example16.cc
#include <vector.h>
#include <assert.h>

int main() {
  vector<long> a;
  const int N(12);
  for (int i=0; i!=N; a.push_back(i++)) {}
  vector<long>::iterator new_end=remove(a.begin(), a.end(), 4);
  assert(N==a.size());
  *new_end = 17;
  assert(*(a.begin()+N-1)==17 && N==a.size());
}
```

Containers

We have already done several examples with:

array: built-in container – "standard" pointers, no member functions, no dynamic expansion, no bounds-checking, etc.

vector: "smart" array – STL member functions, dynamic expansion, bounds-checking, `push_back` in $O(1)$ time.

deque: *almost* identical to vector, but *both* `push_back` *and* `push_front` in $O(1)$ time.

list: insert and delete in $O(1)$ time, but find in $O(N)$ time

In addition, STL provides `set (multiset)` and `map (multimap)`.

Set

set and map (together with multiset and multimap) differ (somewhat) from the previous containers:

- array, vector, deque, and list are *Sequence Containers* – that is, the container is a sequence of elements of type `T`
- set and map are *Sorted Associative Containers* – that is, the container is a sorted sequence of *keys* used to access the elements of type `T`.

set and multiset (and map and multimap) differ:

- set (map) has only one element for a given key
- multiset (multimap) can have multiple elements for a given key

Definitions:

set:

In a set, the data items are just the keys themselves.

For a multiset, a key can be repeated

map:

In a map, the data items are pairs of (key, data). `pair` is an STL-defined class.

For a multimap, duplicate keys are allowed.

And now for an example:

```
#include <algo.h>   // example17a.cc
#include <set.h>
#include <String.h>

int main() {
  String s("that government of the people, by the people, "
   "for the people shall not perish from the earth.");
  cout << s << endl;
  set<char, less<char> > s1;
  for (const char* p=s.chars(); p!=s.chars()+s.length();
       s1.insert(*p++)) {}
  ostream_iterator<char> out(cout);
  copy(s1.begin(), s1.end(), out); cout << endl;
}
```

```
 ,.abefghilmnoprstvy
```

```
#include <algo.h>    // example17b.cc
#include <multiset.h>
#include <String.h>

int main() {
  String s("that government of the people, by the people, "
   "for the people shall not perish from the earth.");
  cout << s << endl;
  multiset<char, less<char> > s1;
  for (const char* p=s.chars(); p!=s.chars()+s.length();
       s1.insert(*p++)) {}
  ostream_iterator<char> out(cout);
  copy(s1.begin(), s1.end(), out); cout << endl;
}
```

```
                ,,.aaabeeeeeeeeeeeeeefffghhhhhhhh
illllllmmnnnoooooooopppppprrrrrsstttttttttvy
```

Points to Note:

- Surprise, surprise! We can use the same old methods and algorithms.
- the template argument `less<char>` is required. In this case, the STL function object. `less<char>` does a lexicographical compare.
- In general, we would *either* supply a compare function object, *or* an `operator<` for type `T`.
- For the multiset, the data item is the key, so the keys (data) are simply duplicated.

Let's do 2 further examples:

- using a different compare function
- using the `erase(key)` and `find` methods

```
#include <algo.h>   // example18a.cc
#include <set.h>
#include <String.h>

int main() {
  String s("that government of the people, by the people, "
   "for the people shall not perish from the earth.");
  cout << s << endl;
  set<char, greater<char> > s1;
  for (const char* p=s.chars(); p!=s.chars()+s.length();
       s1.insert(*p++)) {}
  ostream_iterator<char> out(cout);
  copy(s1.begin(), s1.end(), out); cout << endl;
}
```

```
 yvtsrponmlihgfeba.,
#include <algo.h>   // example18b.cc
#include <multiset.h>
#include <String.h>

int main() {
  String s("that government of the people, by the people, "
   "for the people shall not perish from the earth.");
  cout << s << endl;
  multiset<char, less<char> > s1;
  for (const char* p=s.chars(); p!=s.chars()+s.length();
       s1.insert(*p++)) {}
  s1.erase('e'),  s1.erase('h'),  s1.erase(' ');
  s1.erase(s1.find('f'));
  ostream_iterator<char> out(cout);
  copy(s1.begin(), s1.end(), out); cout << endl;
}
```

```
,,.aaabffgillllllmmnnnoooooooopppppprrrrrssttttttttttvy
```

Map

- Finally, maps (probably more useful than sets) allow a data item to be referenced by a key.
- E.g. a telephone directory is a `map` with key=name, and data=number.
- maps use the STL `pair` class – i.e. (`key, T`) is a `pair`.
- I will leave most of the details to the student.

```
#include <algo.h>   // example19.cc
#include <map.h>
#include <String.h>
#include <iomanip.h>
#include <assert.h>

ostream& operator<<(ostream& os,const pair<const String,long>& p) {
  os << setw(24) << setiosflags(ios::left)<< p.first << p.second;
  return os;
}
int main() {
  map<String, long, less<String> > m;
  m["Ricciardi, Aleta"] = 3642754;
  m["Ogg, Michael"] = 8461432;
  m["Song, John"] = 4865385;
  ostream_iterator<pair<const String, long> > out(cout, "\n");
  copy(m.begin(), m.end(), out);
  assert( (*m.find("Ogg, Michael")).second==8461432 );
}
```

which produces the output:

```
Ogg, Michael              8461432
Ricciardi, Aleta          3642754
Song, John                4865385
```

If instead we had used `multimap` we could have multiple listings for each key (name).

- I will leave as an exercise making a database of event properties, where each event is labelled by an event number (e.g. `pair<int,int>`), and `class Event` is an object of the properties.
- The *standard* map is based on an assorted associative container, so locating an element (`find`) is $O(\log N)$.
- There are extensions (which might become part of the standard) to use a *hash table* – so `find` would be $O(1)$ *most* of the time, but $O(N)$ in the worst case.

Adaptors

and *finally*: to make a container do something different (e.g. to make a vector behave as a stack), we use container adaptors. The idea:

- The new container (e.g. `stack`) has a *private* instance of the old container (e.g. `vector`)
- Therefore none of the data, nor the methods of `vector` are accessible to `stack`
- So: provide new *public* methods (e.g. `push, pop`) defined in terms of the old methods
- Presto-magico! We have a new container.

This can also be done for iterators and function objects.