Maps
The map class supports an associative container in which unique keys are mapped with values. In essence, a key is simply a name that you give to a value. Once a value has been stored, you can retrieve it by using its key. Thus, in its most general sense, a map is a list of key/value pairs. The power of a map is that you can look up a value given its key. For example, you could define a map that uses a person's name as its key and stores that person's telephone number as its value. Associative containers are becoming more popular in programming. As mentioned, a map can hold only unique keys. Duplicate keys are not allowed.

To create a map that allows nonunique keys, use multimap. The map container has the following template specification:
template <class Key, class T, class Comp = less<Key>,
class Allocator = allocator<pair<const key, T> > > class map
Here, Key is the data type of the keys, T is the data type of the values being stored (mapped), and Comp is a function that compares two keys. This defaults to the standard less() utility function object. Allocator is the allocator (which defaults to allocator).
A map has the following constructors:
explicit map(const Comp &cmpfn = Comp(),
const Allocator &a = Allocator() );
map(map<Key, T, Comp, Allocator> &ob);
template <class InIter> map(InIter start, InIter end,
const Comp &cmpfn = Comp(), const Allocator &a = Allocator() );
The first form constructs an empty map. The second form constructs a map that contains the same elements as ob. The third form constructs a map that contains the elements in the range specified by the iterators start and end. The function specified by cmpfn, if present, determines the ordering of the map. In general, any object used as a key should define a default constructor
and overload the `<` operator and any other necessary comparison operators. The specific requirements vary from compiler to compiler.

The following comparison operators are defined for `map`.

`==, <, <=, !=, >, >=`

`key_type` is the type of the key, and `value_type` represents `pair<Key, T>`.

Key/value pairs are stored in a map as objects of type `pair`, which has this template specification.

```
#template <class Ktype, class Vtype> struct pair {
#    typedef Ktype first_type; // type of key
#    typedef Vtype second_type; // type of value
#    Ktype first; // contains the key
#    Vtype second; // contains the value
#    // constructors
#    pair();
#    pair(const Ktype &k, const Vtype &v);
#    template<class A, class B> pair(const A, B &ob);
#};
```

As the comments suggest, the value in `first` contains the key and the value in `second` contains the value associated with that key. You can construct a pair using either one of `pair`'s constructors or by using `make_pair()`, which constructs a `pair` object based upon the types of the data used as parameters. `make_pair()` is a generic function that has this prototype.

```
#template <class Ktype, class Vtype>
pair<Ktype, Vtype>
make_pair(const Ktype &k, const Vtype &v);
```

As you can see, it returns a pair object consisting of values of the types specified by `Ktype` and `Vtype`. The advantage of `make_pair()` is that the types of the objects being stored are determined automatically by the compiler rather than being explicitly specified by you.

The following program illustrates the basics of using a map. It stores key/value pairs that show the mapping between the uppercase letters and their ASCII character codes. Thus, the key is a character and the value is an integer. The key/value pairs stored are
and so on. Once the pairs have been stored, you are prompted for a key (i.e., a letter between A and Z), and the ASCII code for that letter is displayed.

// A simple map demonstration.
#include <iostream>
#include <map>
using namespace std;
int main()
{
    map<char, int> m;
    int i;
    // put pairs into map for(i=0; i<26;
    i++) { m.insert(pair<char, int>('A'+i,
        65+i));
    }
    char ch;
    cout << "Enter key: ";
    cin >> ch;
    map<char, int>::iterator p;
    // find value given key
    p = m.find(ch);
    if(p != m.end())
        cout << "Its ASCII value is " << p->second;
    else
        cout << "Key not in map.\n";
    return 0;
}

Notice the use of the pair template class to construct the key/value pairs. The data types specified by pair must match those of the map into which the pairs are being inserted. Once the map has been initialized with keys and values, you can search for a value given its key by using the find( ) function. find( ) returns an iterator to the matching element or to the end of the map if the key is not found. When a match is found, the value associated with the key is contained in the second member of pair.
In the preceding example, key/value pairs were constructed explicitly, using \texttt{pair<char, int>}. While there is nothing wrong with this approach, it is often easier to use \texttt{make_pair()}, which constructs a pair object based upon the types of the data used as parameters. For example, assuming the previous program, this line of code will also insert key/value pairs into \texttt{m}.

\texttt{m.insert(make_pair((char)('A'+i), 65+i));} Here, the cast to \texttt{char} is needed to override the automatic conversion to \texttt{int} when \texttt{i} is added to 'A.' Otherwise, the type determination is automatic.