Associative Containers

An associative container is a set (or a multiset) of pairs, which is arranged in such a way that it can do quick lookup on the first element. The first element is usually called key, and the second element is usually called value. There exist two main types of associative container:

1. Binary trees. The tree is ordered, and the left and right subtree have approximately the same size, so that look up can take place in $O(\log(n))$ time. Insertion and deletion can be done in $O(\log(n))$ time amortized.
2. Hash maps. The keys are mapped to natural numbers in pseudo random fashion. The function that does this is called hash function. The pairs are stored in an array indexed by the hash values. If the hash function is random enough, different keys are put on the same place very rarely. Lookup can be done in time $O(1)$. Insertion and deletion can be done in time $O(1)$ amortized.

Pairs

Pairs are constructed by the constructor `std::pair<X,Y>` :

```
std::pair<int,double> p( 1, 2 );
std::pair<int,std::string> q = { 100, "sto" };
```

If both X, Y have default constructors, they can be default constructed:

```
std::pair<int, std::string> pp;
```

If X, Y have assignment operators, then pairs can be assigned:

```
std::pair<int,std::string> q = { 100, "sto" };
q = { 1000, "tysiace" };
```

Pairs

The definition of `std::pair` is

```
template< class X, class Y > struct std::pair
{
   typedef X first_type;
   typedef Y second_type;

   X first;     // Fields are public.
   Y second;
```

Pairs

```
   pair( ) :
      first( X( )),
      second( Y( ))
   { }

   pair( const X& x, const Y& y ) :
      first( x ),
      second( y )
   { }
}
```

Pair has default constructor, copy constructor, moving constructor, assignment, moving assignment, whenever both of its members have it.

std::map and std::unordered_map

```
std::map< std::string, unsigned int > english =
   { { "one", 1 }, { "two", 2 }, { "three", 3 },
     { "four", 4 }, { "five", 5 } };

std::unordered_map< std::string, unsigned int > polish =
   { { "jeden", 1 }, { "dwa", 2 }, { "trzy", 3 },
     { "cztery", 4 }, { "piec", 5 } };

for( const auto& p : english )
   std::cout << p. first << " => " << p. second << "\n";

// Also works for Polish.
```

Inserting an Element

```
auto p =
english. insert(
   std::pair< std::string, unsigned int > ( "six", 6 ));

   // Alternatively, nicer:

   auto p = english. insert( { "six", 6 } );
      // Compiler will find that { ... } must be
      // std::pair< std::string, unsigned int >

   // { "six", 6 } is only inserted if there was no entry
   // for "six". Otherwise, existing value is not changed.
```

```
if( p. second )
   std::cout << "inserted\n";
else
   std::cout << "already existed, value not changed";

if( p.first -> first != "six" )
   std::cout << "something impossible happened" );

std::cout << ( p.first -> second ) << "\n";
   // 6 when insertion happened.

( p. first -> second ) = 6;
   // Now are are sure it's 6.
```

The type of `p` is

```
std::pair< std::pair< const std::string,
                      unsigned int >>
```

Looking for an Element

```
std::string s = "fuenf";

std::unordered_map< const std::string, unsigned int > p =
               polish. find( s );

   // Returns valid iterator (containing a pair)
   // if element exists, otherwise .end( ).

   if( p != polish. end( ))
      std::cout << ( p -> first ) << " = "
               << ( p -> second ) << "\n";
   else
      std::cout << s << " not found";
```

[ ] and `at( )`

```
english [ "five" ] = 5;
english [ "six" ] = 6;

polish [ "piec" ] = 5;
polish [ "szesc" ] = 6;
   // Easy to use, but there are some subtleties.
   // If you want presence of the element to be checked,
   // use:

english. at( "fuenf" ) = 5;
polish. at( "cinq" ) = 5;
   // Throw out_of_range exception. It is guaranteed
   // that at( ) does not change the container.
```

[ ] and default constructors

In contrast to `at( )`, and also in contrast to the behaviour of [ ] on `std::vector`, the method [ ] is total.

If no value exists for the given key, it will use the default constructor of the value type to construct one.

1. The value type Y must have a default constructor. Otherwise, the code will not compile.

2. The associative container cannot be **const**. Otherwise, the code will not compile.

3. You find it acceptable that the associative container will change, or you are sure that key X is already in the container.

4. You find it acceptable that Y is first initialized, and then overwritten, or you are certain that this will not happen because X is already present in the container.

When to use [ ].

In cases where being present with the default value means the same as not being present.

For example if you want to count how often string appear in a text, you could use
`std::map< std::string, unsigned int > counter`. Strings that don't occur in `counter` can be assumed to appear in counter with value 0. There is no distinction between that.

```
while( ...... )
{
   std::string s = ... (string to be counted)
   counter [ s ] ++ ;
}
```

Map

`std::map<X,Y>` is defined by a binary search tree. The compiler needs to know how sort the X. If you define `std::map<X,Y>`, the compiler needs to now which order to use for sorting the tree.

The default is `std::less<X>`, which is defined as `operator <` for most standard types.

`std::map` assumes that two elements `x1,x2` are equal if both `x1<x2` and `x2<x1` return `false`.

Providing the Order

If type X has no defined order, or you want to use a different order, you can provide an order as third argument.

The order can is passed as a type. The type must have a default constructor, and any inhabitant of the type must be applicable to two elements of X through an application operator:

```
struct compare
{
   bool operator( const X& x1, const x& x2 ) const;
};
```

Don't define `operator <` on type `X`, when there is no natural meaning.

Ordered Map

In general, `std::map` is less efficient than `std::unordered_map`.

Use it only when the order matters, as in

```
auto p1 = english. find( "one" );
auto p2 = english. find( "two" );
for( auto p = p1; p < p2; ++ p )
{
   .. will be in alphabetical order.
}
```

If you don't want to use this, then use `unordered_map`.

Hash Map

Hashmaps were (finally) added in $C^{++}11$. As far as I know, they are implementated as follows:

```
template< class Key, class Value, class Hash = hash<Key>,
          class Pred = equal_to<Key> >
struct std::unordered_map
{
   std::list< std::pair< Key, Value >> contents;

   std::vector< std::list< std::pair< Key, Value >>
            :: iterator > table;
   // If the hash value of k equals i, we look for
   // k in the segment table[i] .. table[i+1] of contents
   // (table[i] .. contents. end( )
   //    if i+1 == table. size( )).
};
```

Hash Map

The following two things are needed:

1. A hash function.
2. Some way of determining when two keys are equal.

Hash Function

The hash function is passed as type. The type must have a default constructor. Inhabitants of the type must have an application operator:

```
struct hashtype
{
   unsigned int operator( ) ( const Key& k ) const
};
```

The `unordered_list` will construct a default object, and call it to compute hash values.

Hash Function

If you don't specify the hash function, `hashmap` will use default `std::hash<Key>`, which exists for most built-in types.

For self-defined classes, you need to provide a hash function type.

Equality Predicate

The equality predicate is passed as a type. The type must have a default constructor.

Inhabitants of the type must have an application operator:

```
struct equals // Or some other name.
{
   bool operator( ) ( const Key& k1, const Key& k2 ) const
};
```

If you don't provide an equality predicate, then `equal_to<Key>` will be used, which is defined as `==` on most types.

You must define your own equals predicate for your own types, or if you don't want `==`.

Summary

Use the STL! It is well-designed, has a nice user interface, and is close to optimal in terms of efficiency.

Never put pointers in a container!