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 `size_t` 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, `pairs` 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< typename X, typename Y >
struct std::pair
{
    typename first_type = X;
    typename second_type = Y;

    X first;
    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

```cpp
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.
```

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

if( p.first -> first != "six" )
   std::cout << "the impossible just 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::map< const std::string,
                            unsigned int > :: iterator, bool >
```

If you apply * on the first element, you get

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

This means that `p.first -> first` is the key, and
`p.first -> second` is the value (either the inserted value, or the
previous value).

Emplace

Since C^{++}-11, elements can be inserted with `emplace` as follows:

```
auto p = english. emplace( "six", 6 );
```

It reduces copying, in case that it is important.

Looking for an Element

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

std::unordered_map< const std::string,
                    unsigned int > :: iterator
   p = polish. find( s );    // Use 'auto' in real code.

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

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

11

Checking if Element is there

If you don't plan to insert the element (only check if it is there), you can use:

```
std::string s = "fuenf";
if( polish. find(s) != polish. end( ))
    std::cout << "element " << s << " is there\n";
```

Deleting an Element

Deleting can be done in two ways:

```
size_t s = english. erase( "three" );
    // Returns number of deleted elements,
    // 0 or 1.


p = english. find( "four" );
if( p != english. end( ))
    english. erase(p);
        // This version of erase has no
        // return value.
```

[ ] and default constructors

There are some difficulties in using [ ], deriving from the fact that [ ] 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 [ ].

$\Rightarrow$ 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 ];
   }
```

[ ] and `at( )`

`at` is similar to [ ], but is not total, and does not create entries:

```
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.
```

Map

`std::map<X,Y>` is defined by a binary search tree (usually red-black tree). The tree needs to know how sort the X.

For this, it uses a third parameter C. The default value is `C = 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 <, or you want to use a different order, you can provide an order as third argument.

The order 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 comparator
{
    bool operator( ) ( const X& x1, const x& x2 ) const;
};
```

Don't define `operator` < on type `X`, when there is no natural meaning. Once you have defined it, it is everywhere.

Ordered Map

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

Use it only when the order matters, as in

```
for( auto p = mp. begin( ); p != mp. end( ); ++ p )
{
    .. will be in alphabetical order.
}
```

If you don't care about order, use `unordered_map`.

Hash Map

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

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

   std::vector< std::list< std::pair< Key, Value >>
            :: iterator > table;

   // If hash(s) % table.size( ) = i,  we look for s
   // in the segment table[i] .. table[i+1] of contents.
};
```

Hash Map

The following two things are needed:

1. A hash function.

2. Some way of determining when two keys are equal.

Hash functions should be such that $s_1 \neq s_2, \quad h(s_1) = h(s_2)$ happens very rarely.

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
{
   size_t operator( ) ( const Key& k ) const
};
```


The `unordered_map` 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 a class that you defined by yourself, you need to define a `struct my_hashfunction{  };`

Equality Predicate

The equality predicate is also passed as type. The type must also 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 some `struct my_equals` if you don't want `==`.

Why Equals?

Some students seem to believe that one can implement

```
struct equals
{
   hash h;

   bool operator( ) ( const Key& k1, const Key& k2 ) const
   {
      return h( k1 ) == h( k2 );
   }
};
```

I have seen such code in exercises (I saw it again in 2017).

It is wrong, because equal hash values (for different objects) should be improbable, but still are possible. This code may pass testing if the hash function is good.

Function Objects

We have now seen that the standard way of defining functions in C^{++} is by defining a type T that has a default constructor. The elements of type T must have an application operator which is the function that we are defining.

Anonymous Types, or Lambdas

There are other ways of defining functions in C^{++}, namely by lambdas or by function pointers. Both can be used with containers since $C^{++} - 11$, but the result is ugly, and I don't recommend it.

Derived Containers

There are a few more containers in STL:

```
multimap< >
    // As map, allows keys to occur more than once.
    // Use lower_bound( ), upper_bound( ) for finding
    // all elements.
unordered_multimap< >
    // As unordered_map, allows keys to occur more
    // than once. Use equal_range( ).
set< >
    // As map, but has no value type.
unordered_set< >
    // As unordered_map, but has no value type.
```

The interfaces are similar, look at `http://www.cplusplus.com/`

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!

There is a problem between [ ] and default constructors, and also between [ ] and **const**ness. Use **at**, or **find**.

Make sure that the types of containers are defined only at one point, so that you can easily change the ordering, or the hash function. Use **using, auto** and **decltype** to avoid mentioning the type of a container more than once.