Initializer lists (1)

Container initializer_list is defined as follows:

```
initializer_list<type_of_elements> list_name = { sequence_of_values };
```

Methods implemented in initializer_list are *size*, *begin* and *end*.

Examples:

```
#include <initializer_list>
        // see https://en.cppreference.com/w/cpp/utility/initializer_list.html
using namespace std;
initializer_list<int> il = { 1, 2, 3, 4, 5 };
for (initializer_list<int>::iterator it = il.begin(); it != il.end(); ++it)
     cout << *it << " ";
initializer_list<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019),
                                     Date(26, 12, 2019) };
for (auto it = christmas.begin(); it != christmas.end(); it++)
    cout << it->ToString() << endl;
```

The values specified in initializer list are constants:

```
for (auto it = christmas.begin(); it != christmas.end(); it++)
    cout << it->SetYear(2020) << endl; // error
```

Initializer lists (2)

The initializer list is very efficient for writing functions with variable number of arguments.
Example:

```
void print(initializer_list<int>);
void print(initializer_list<int> il)
{
  for (auto it = il.begin(); it != il.end(); ++it)
  {
       cout << *it << " ";
   }
   cout << endl;
}
```

Usage:

```
int main()
{
  print({ 1, 2, 3, 4, 5, 6 });
  return 0;
}
```

String as container

Generally, string in C++ is also a container but as it has a lot of specific methods for operating with characters, we may do not turn attention to this fact. The string has the same iterators as vector: *begin, end, cbegin, cend, rbegin, rend, crbegin, crend*. There is a constructor that uses iterators to another string as parameters. Example:

```
string s = "We have 125 euros"; // extract the number
int i = 0, j = 0;
for (auto it = s.begin(); it != s.end(); it++, i++) {
    if (isdigit(*it))
        break;
}
for (auto it = s.begin() + i; it != s.end(); it++, j++) {
    if (!isdigit(*it))
        break;
}
string our_money(s.begin() + i, s.begin() + i + j); // constructor with iterators
cout << our_money << endl; // prints 125
```

String iterators may be used for inserting, erasing and replacing. Example:

```
string no = "no";
s.replace(s.begin() + i, s.begin() + i + j, no.begin(), no.end());
cout << s << endl; // prints "We have no euros"
```

See more on https://en.cppreference.com/w/cpp/string/basic_string.html

Multimaps (1)

Multimap is very similar to map. The difference is that the multimap may contain several elements with the same key.

Example:

```
#include <map> // See https://en.cppreference.com/w/cpp/container/multimap.html
using namespace std;
multimap<string, Date> deadlines = {
        { "Mathematics", Date(5, 1, 2019) },
        { "Mathematics ", Date(10, 1, 2019) },
        { "Mathematics ", Date(15, 1, 2019) }
};
```

The multimap cannot support *operator[]* and *at* methods. As the inserting never fails,

```
auto return_value_name = multimap_name.insert( { key, value });
```

and

```
auto return_value_name = multimap_name.insert( make_pair(key, value));
```

return always the iterator to the inserted element.

To get the number of elements with the same key use method *count*:

```
int number_of_elements = multimap_name.count(key);
```

Example:

```
cout << deadlines.count("Mathematics") << endl;  // prints 3
```

Multimaps (2)

To get the range of elements with the same key use methods *lower_bound* and *upper_bound*:

```
multimap<string, int> students = { { "John", 5 }, { "Mary", 4 }, { "Mary", 2 },
        { "Elizabeth", 5 }, { "James", 1 }, { "Mary", 6 }, { "Walter", 2 }, { "Samuel", 5} };
auto it1 = students.lower_bound(string("Mary"));
cout << (it1->first).c_str() << ' ' << it1->second << endl; // prints Mary 4
auto it2 = students.upper_bound(string("Mary"));
cout << (it2->first).c_str() << ' ' << it2->second << endl;
// prints Samuel 5 (the first that is not Mary)
// the order inside map is Elizabeth, James, John, Mary, Mary, Mary, Samuel, Walter
```

If no one element was found, the return values of *lower_bound* and *upper_bound* are identical:

```
auto it3 = students.lower_bound(string("Timothy"));
cout << (it3->first).c_str() << endl; // prints Walter
auto it4 = students.upper_bound(string("Timothy"));
cout << (it4->first).c_str() << endl; // prints Walter
```

A bit more convenient method to get a range is to apply method *equal_range*:

```
auto range = multimap_name.equal_range(key);
```

Here *range* is a pair in which member *first* is the iterator pointing to the lower bound and member *second* is the iterator pointing to the upper bound. Example:

```
auto range = students.equal_range(string("Mary"));
cout << (range.first)->first.c_str() << ' ' << (range.first)->second << endl; // prints Mary 4
cout << (range.second)->first.c_str() << ' ' << (range.second)->second << endl; //prints Samuel 5
```

Sets (1)

Set is also very similar to map. The difference is that the elements are not key / value pairs but the element itself is a unique key. In memory the sets are implemented as balanced binary trees. A set is defined as follows:

```
set<type_of_elements> set_name = { sequence_of_initial_values };
```

or

```
set<type_of_element> *pointer_name  =  new set<type_of_elements>
                                        { sequence_of_initial_values };
```

The initial values are optional. If they are not present, empty set is created.

Example:

```
#include <set> // See https://en.cppreference.com/w/cpp/container/set.html
using namespace std;
set<string> subjects = {  "Mathematics", "Physics", "Chemistry", "Programming in C++",
"Programming in Java" };
```

The set cannot support *operator[]* and *at* methods. As in maps

```
auto return_value_name = set_name.insert(new_element);
```

the return value is a pair in which member *first* is an iterator referring to the new element or, if the inserting failed, to already existing element having the same key. Member *second* is of type *bool*. If it is *false*, the operation failed. Example:

```
auto ret = subjects.insert("Software security");
cout << boolalpha << ret.second << endl; // prints true
cout << ret.first->c_str() << endl; // prints "Software security"
```

Sets (2)

Traveling through the set using iterators is as with the other containers:

```
for (auto it = subjects.begin(); it != subjects.end(); ++it)
   cout << *it << endl;
```

or

```
for (auto& x : subjects)
    cout << x.c_str() << endl;
```

The results are printed in sorted order.

The elements of a set are constants:

```
for (auto it = subjects.begin(); it != subjects.end(); ++it)
{
  if (*it == "Programming in Java")
    *it = string("Programming in C#"); // error, cannot change the set elements
 }
for (auto& x : subjects)
{
  if (x == "Programming in Java")
     x = "Programming in C#"; // error, cannot change the set elements
 }
```

Methods like *find*, *erase*, *lower_bound*, *upper_bound*, etc. are as in maps except that instead of key we have to use the element itself.

Multisets

The difference between *set* and *multiset* is that a value in *multiset* may occur several times.

Example:

```
#include <set> // See https://en.cppreference.com/w/cpp/container/multiset.html
using namespace std;
multiset<Date> deadlines = {  Date(5, 1, 2019),  Date(10, 1, 2019) }, Date(15, 1, 2019),
                              Date(5, 1, 2019), Date(5, 1, 2019), Date(15, 1, 2019) };
```

As the inserting never fails,

```
auto return_value_name = multiset_name.insert(new_element);
```

returns only the iterator to the inserted element.

As multimap, multiset also supports method *count.* To get the range of elements with the same key use methods *lower_bound* and *upper_bound.*

Hashing

Let us have an empty array (hash table) with length *m* and a function *h(k)* (hash function) so that:

- the arguments of *h(k)* are the keys of our objects
- the value calculated by *h(k)* is an integer of range *0…m-1*

If the keys are integers, the simplest hash function is:

```
int hash(int k, int m)
{
    return k % m; // remainder of division operation
}
```

The return value of hash function gives us the index of our object in hash table. If we need to insert an object, we put it into the calculated location. If we need to find an object, we calculate its location and check, is it there or not. Of course, the objects in table are unordered (not sorted).

The drawback of hashing is that the index calculated with hash function may be not unique, i.e. the location to which we want to store our object may be already occupied by another object. For example, if $m = 100$ then objects with keys 200, 300, .. claim position with index 0. This situation is called as collision.

There are many approaches to select a proper hash function and the length of table and to handle collisions.

Unordered maps (1)

An *unordered_map* stores key-value pairs in a hash table. Insertion, removing and access are based on keys. The keys must be unique.

An *unordered_map* is defined as follows:

```
unordered_map<type_of_key, type_of_value> unordered_map_name = { pairs_of_initial_values };
```

or

```
unordered_map<type_of_key, type_of_value> *pointer_name  = new unordered_map<type_of_key,
type_of_value> { pairs_of_initial_values }
```

The initial values are optional. If they are not present, empty map is created.

Examples:

```
#include <unordered_map> // See https://en.cppreference.com/w/cpp/container/unordered_map.html
using namespace std;
unordered_map<string, Date> deadlines = {
            { "Mathematics", Date(5, 1, 2019) },
            { "Chemistry", Date(10, 1, 2019) },
            { "Physics", Date(15, 1, 2019) }
};
unordered_map<string, Date> *pDeadlines = new unordered_map<string, Date>;
delete pDeadlines;
```

Unordered maps (2)

Default hash function is provided for keys of C++ standard types like integers and strings. If the key is an user-defined object, the *unordered_map* is defined as:

```
unordered_map<type_of_key, type_of_value, hash_function_class>
                                    unordered_map_name = { pairs_of_initial_values };
```

Hash function class must contain constant method *operator()*. The argument must be a constant reference to key object and return value must be of type *size_t*.

Example:

```
class DateHash
{
 public:
     size_t operator() (const Date &d) const {
              return (d.GetDay() + d.GetMonth() + d.GetYear()) % 101;
     }
};
unordered_map<Date, string, DateHash> deadlines = {
        { Date(5, 1, 2019), "Mathematics" },
        { Date(10, 1, 2019), "Physics" },
        { Date(15, 1, 2019), "Chemistry" }
};
```

Unordered maps (3)

Most of methods of *unordered_map* are similar to the corresponding methods of *map*. Examples:

```
auto ret = deadlines.insert(make_pair(Date(18, 1, 2019), "Programming in Java"));
```

Remember that the return value is a pair in which member *first* is a map iterator referring to the new element or, if the element with the specified key was present and therefore the inserting failed, to already existing element having the same key. Member *second* is of type *bool*. If it is *false*, the operation has failed. The map iterator in return value itself has members *first* presenting the key and *second* presenting the value.

```
if (ret.second)
   cout << (ret.first->first).ToString() << ' ' << (ret.first->second).c_str() << endl;
for (auto it = deadlines.begin(); it != deadlines.end(); ++it)
   cout << (it->first).ToString() << ' ' << (it->second).c_str() << endl;
deadlines[Date(12, 1, 2019)] = "Programming in C++";
for (auto& x : deadlines)
   cout << x.first.ToString() << ' ' << x.second.c_str() << endl;
auto it = deadlines.find(Date(5, 1, 2019));
cout << (it->first).ToString() << ' ' << (it->second).c_str() << endl;
deadlines.erase(Date(15, 1, 2019));
```

unordered_map does not support iterating backwards (*rbegin*, *rend*, *crbegin*, *crend*) and methods *lower_bound* and *upper_bound*.

Unordered maps (4)

unordered_map has several methods for analyzing the situation in built-in hash table. The table elements are called buckets. Normally a bucket contains one object, but due to collisions there may be several.

1. int bucket_number = bucket_count(); returns the number of buckets in hash table.
2. void rehash(number_of_buckets); sets the new number of buckets accompanied with the reorganization of table. Ignored if the argument is less than the current number of buckets.
3. int max_possible_bucket_number = max_bucket_count(); returns the number of buckets that the hash table is possible to contain.
4. int bucket_index = bucket(key); returns the index of bucket containing object with the specified key.
5. int number_of_elements_in_bucket = bucket_size(bucket_index); returns the number of objects in the specified bucket (mostly 1).
6. float load_factor = load_factor(); the load factor is the ratio between the number of objects in the container and the number of buckets. If the load factor is 1, there are no empty buckets and the collisions are inevitable.
7. void max_load_factor(max_value); sets the upper limit for load factor. After each inserting the load factor is automatically recalculated and compared with the upper limit. If the limit is exceeded, the number of buckets is automatically increased and the table reorganized.

Unordered multimaps

unordered_multimap is very similar to *unordered_map* and *multimap* It may contain several elements with the same key.

Example:

```
#include <map> // See https://en.cppreference.com/w/cpp/container/unordered_multimap.html
using namespace std;
unordered_multimap<string, Date> deadlines = {
        { "Mathematics", Date(5, 1, 2019) },
        { "Mathematics ", Date(10, 1, 2019) },
        { "Mathematics ", Date(15, 1, 2019) }
};
```

Unordered sets

unordered_set is very similar to *unordered_map* and *set*. The element itself is a unique key.
Example:

```
#include <set> // See https://en.cppreference.com/w/cpp/container/unordered_set.html
using namespace std;
unordered_set<string> subjects = {  "Mathematics", "Physics", "Chemistry", "Programming in C++", "Programming in Java" };
```

Unordered multisets

unordered_multiset is very similar to *unordered_set*. A value in *unordered_multiset* may occur several times.

Example:

```
#include <set> // See https://en.cppreference.com/w/cpp/container/unordered_multiset.html
using namespace std;
unordered_multiset<Date, DateHash> deadlines = {  Date(5, 1, 2019),  Date(10, 1, 2019) },
Date(15, 1, 2019), Date(5, 1, 2019), Date(5, 1, 2019), Date(15, 1, 2019) };
```

Allocators

The template presenting vectors is not *template<typename T> class vector {....};* The correct expression is:

template<typename T, typename Allocator = allocator<T> > class vector { ………….. };

Templates for lists, maps, etc. are similar. Allocator class is responsible for memory management. The default allocator (STL standard) is presented by template:

template<typename T> class allocator { ………………. };

Typically, the default allocator that uses *new* and *delete* operators is good enough and as it is the default value of template second parameter, we may simply not think about allocators. Sometimes, however, due to the problems with performance (games, for example) and / or fragmentation or when we want to use the specific capabilities of operating system, we may have to develop our own custom allocator.

Bitsets(1)

In C, we have bitwise operations AND &, OR |, exclusive OR ^, negation ~, shifting left << and shifting right >>. We may also handle separate bits using bit fields.

Bitset in C++ is a container storing a fixed number of bits:

```
std::bitset<dimension> bitset _name("initial values");
```

for example:

```
#include <bitset> // see more in https://en.cppreference.com/w/cpp/utility/bitset.html
bitset<5> bits1("10101");
```

To see the values use *cout*:

```
cout << bits1 << endl; // prints 10101
```

Initial values in definition are optional. If they are not present, all the bits are set to zero:

```
bitset<5> bits2;
cout << bits2 << endl; // prints 00000
```

The initial value may be presented also by a 32-bit unsigned integer:

```
bitset<8> bits3(0xF1);
cout << bits3 << endl; // prints 11110001
```

A bitset may be converted into *string*, *unsigned long* or *usigned long long*. Examples:

```
string s = bits1.to_string();
unsigned long lu = bits1.to_ulong();
unsigned long long llu = bits1.to_ullong();
cout << "0x" << hex << llu << ' ' << dec << llu << endl; // prints 0x15 21
```

Bitsets (2)

To access a bit from set you may use unsecure *operator[]*. Examples:

```
bitset<5> bits1("10101");
bits1[1] = 1;
cout << bits1 << endl;  // Order positions are counted from the rightmost bit, which is order
                        // position 0, the result is 10111 and not 11101
cout << boolalpha << bits1[1] << endl; // prints true
bits1[10] = 1; // wrong index, the program crashes
```

Secure access methods are *test* (returns the value of specified bit), *set* (sets new value for the specified bit) and flip (converts the specified bit from 1 to 0 or vice versa):

```
try {
	cout << boolalpha << bits1.test(1) << endl; // prints true
	bits1.set(3, 1);
	bits1.set(4, 0);
	bits1.flip(0);
	cout << bits1 << endl; // prints 01110
	cout << boolalpha << bits1.test(10) << endl; // throws exception
}
catch (out_of_range &e) {
	cout << e.what() << endl; // prints "invalid bitset position"
}
```

Bitsets (3)

Method *set()* without arguments sets all ther bits to 1 and *reset()* all the bits to zero. Method *flip()* without arguments converts all the bits in set:

```
bitset<6> bits4("101010");
bits4.flip();
cout << bits4 << endl; // prints 010101
bits4.set();
cout << bits4 << endl; // prints 111111
bits4.reset();
cout << bits4 << endl; // prints 000000
```

Assignment, comparing (only == and *!*=) and bitwise operations between bitsets are supported but only if the dimensions match. Examples:

```
bitset<6> bits5("101010"), bits6("010101");
bitset<6> bits7 = bits5 & bits6;
cout << bits7 << endl; // prints 000000
bitset<6> bits8 = bits5 | bits6;
cout << bits8 << endl; // prints 111111
bitset<6> bits9 = bits5 ^ bits6;
cout << bits8 << endl; // prints 111111
bitset<6> bits10 = bits5 << 2;
bitset<6> bits11 = bits6 >> 2;
cout << bits10 << ' ' << bits11 << endl; // prints 101000 000101
```

Bitsets (4)

Method *all()* returns *true* if all the bits in set are 1. Method *none()* returns *true* if all the bits are zero. Method *any()* returns true if there is at least one bit with value 1. Examples:

```
bitset<6> bits12("111111"), bits13("000000"), bits14("001000");
cout << boolalpha << bits12.all()  << ' ' << bits13.none() << ' ' << bits14.any() << endl;
                                    // prints true true true
cout << boolalpha << bits14.all() << ' '<< bits14.none() << endl; // prints false false
```

Method size() returns the dimension of bitset. Method count() returns the number of bits with value 1:

```
cout << bits14.size() << ' ' << bits14.count() << endl; // returns 6 1
```