C++ containers: usage requirements

To use C++ containers for a user-defined class, this class must contain the following methods:

- Copy constructor or move constructor or the both (obligatory)
- Destructor (obligatory)
- Assignment operator= with copying or moving or with the both (obligatory)
- Constructor without arguments (obligatory)
- operator== (not obligatory, but if not present, a lot of operations will fail)
- operator< (as above)

Vectors (1)

The vector is like array but when an element is inserted or deleted, it automatically resizes itself. A vector is defined as follows:

vector<type_of_elements> vector_name(number_of_elements, initial_value)

or

vector<type_of_elements> *pointer_name = new vector<type_of_elements> (number_of_elements, initial_value)

The initial value is optional. If it is not present the elements are initialized to zero or as objects are constructed by default (having no arguments) constructor.

Examples:

```
#include <vector> // See www.cplusplus.com/reference/vector/vector/
using namespace std;
vector<int> iVector(10); // array for 10 integers initialized to 0 as object iVector
vector<double> dVector(10, 10.0); // array for 10 doubles initialized to 10
vector<string> sVector(10); // array of 10 empty strings
vector<Date> January(31); // array of 31 dates as object January, the attributes of Date
                          // objects are set by default constructor, i.e. they are all the
                          // same
vector<Date> *pJanuary = new vector<Date>(31);
                             // dynamically allocated array of 31 dates
delete pJanuary; // not delete[]
```

Vectors (2)

There are 5 possibilities to access vector elements:

1. Overloaded *operator[]*, for example:
   cout << January[0].GetDay() << endl;
   If the index is wrong, the program will crash.
2. Method *at*, for example:
   cout << January.at(0).GetDay() << endl;
   If the index is wrong, throws the *out_of_range* exception.
3. Method *front* to access the first element, for example:
   cout << January.front().GetDay() << endl;
4. Method *back* to access the last element, for example:
   cout << January.back().GetDay() << endl;
5. Method *data* to get the pointer to the first element, for example:
   Date *pDate = January.data();
   cout << pdate->GetDay() << endl; // prints the first day
   cout << (pDate+1)->GetDay() << endl; // prints the second day

```
Date d30 = January[30]; // for d30 copy constructor from class Date is called
Date d1;
d1 = January[1];  // for d1 operator=() from class Date is called
```

Vectors (3)

Replacing an element is straightforward, for example:

```
January[0] = Date(1, 1, 2019);
January.at(1) = Date(2, 1, 2019);
```

In both cases the old date is destroyed and, using the constructor and overloaded assignment, the new element is built. Important:

```
Date d(1, 1, 2019);
January[0] = d; // January[0] and d are different objects with their own memory fields
```

For better understanding study the following examples:

```
vector<Date> week(7); // as there are no initial values, the vector is filled with dates created
                      // by constructor without arguments Date::Date()
for (int i = 0;  i < 7;  i++)
   week[i] = Date(6 + i, 1, 2020); // the members of vector are replaced, the week now
                                   // presents interval from Jan 6 until Jan 12 2020.
for (int i = 0;  i < 7;  i++)
   cout << week[i].ToString() << endl; // prints the vector members
```

Here *week* is a local variable. When it gets out of scope, destructors for all its members are called automatically. After that the destructor of *vector* is called (also automatically). More about automatical deleting of vector elements see slides *Vectors(17)* and *Vectors(18)*.

Vectors (4)

```
vector<Date> *pWeek = new vector<Date>(7); // as there are no initial values, the vector is
                                           // filled with dates created by constructor
                                           // without arguments Date::Date()
for (int i = 0;  i < 7;  i++)
    pWeek->at(i) = Date(6 + i, 1, 2020); // the members of vector are replaced, the week now
                                         // presents interval from Jan 6 until Jan 12 2020.
```

Alternative solution:

```
for (int i = 0;  i < 7;  i++)
    (*pWeek)[i] = Date(6 + i, 1, 2020);
```

```
for (int i = 0;  i < 7;  i++)
    cout << pWeek->at(i).ToString() << endl; // prints the vector members
```

Alternative solution:

```
for (int i = 0;  i < 7;  i++)
   cout <<  (*pWeek)[i].ToString() << endl:
```

```
delete pWeek; // destructors for all the members are called automatically
```

Vectors (5)

```
vector<Date *> week (7); // as there are no initial values, the vector contains zero pointers
for (int i = 0;  i < 7;  i++)
    week[i] = new Date(6 + i, 1, 2020); // the members of vector are replaced, the week now
                                        // presents interval from Jan 6 until Jan 12 2020.
for (int i = 0;  i < 7;  i++)
    cout << week[i]->ToString() << endl; // prints the vector members
```

Alternative solution:

```
for (int i = 0;  i < 7;  i++)
   cout <<  week.at(i)->ToString() << endl;
```

Here *week* is a local variable. When it gets out of scope, destructors of objects to which the members point are not automatically called. We have to delete the objects ourselves:

```
for (int i = 0;  i < 7;  i++)
     delete week[i];
```

Alternative solution:

```
for (int i = 0;  i < 7;  i++)
    delete week.at(i);
```

Remark: compare with slide *Vectors(3)*.

Vectors (6)

```
vector<Date *> *pWeek = new vector<Date *>(7); // as there are no initial values, the
                                               // vector containes zero pointers
```

pWeek is a pointer to vector, the members of vector are pointers to objects of class *Date*.

```
for (int i = 0;  i < 7;  i++)
    pWeek->at(i) = new Date(6 + i, 1, 2020); // the members of vector are replaced, the week
                                             // now presents interval from Jan 6 until Jan 12
                                             // 2020.
```

Alternative solution:

```
for (int i = 0;  i < 7;  i++)
    (*pWeek)[i]= new Date(6 + i, 1, 2020);
```

```
for (int i = 0;  i < 7;  i++)
    cout << pWeek->at(i) ->ToString() << endl; // prints the vector members
```

Before deleting vector *pWeek* we have to delete the objects ourselves:

```
for (int i = 0;  i < 7;  i++)
    delete pWeek->at(i);
delete pWeek; // objects to which the members point are not automatically deleted.
```

Remark: compare with slide *Vectors(4)*.

Vectors (7)

Method *size* returns the number of elements. Method *resize* increases the size (new positions are initialized to zero) or shrinks (last elements are removed). Method *empty* returns whether the vector contains elements (*false*) or not (*true*). Examples:

```
vector<Date> *pMonth = new vector<Date>(1);
cout << pMonth->size() << endl; // prints 1
pMonth->resize(10);
cout << pMonth->size() << endl; // prints 10
pMonth->resize(0);
cout << boolalpha << pMonth->empty() << endl; // prints true
```

If you define a vector not specifying the number of elements, you get also an empty vector:

```
vector<Date> *pMonth = new vector<Date>;
cout << boolalpha << pMonth->empty() << endl; // prints true
```

Vectors (8)

Vector has constructors and operator functions for copying, assigning and comparing.
Examples:

```
vector<Date> January(31);
vector<Date> February = January; // copy constructor
February.resize(28);
vector<Date> March;
March = January; // assignment overloading
```

Comparing of vectors containing objects of user-defined classes is possible if the class contains *operator*== and *operator*< methods. Turn attention that for example:

```
vector<Date> week1(7);
vector<Date> week2(7);
cout << boolalpha << (week1 == week2) << endl;
```

compiles if the operator function

```
bool Date::operator==(const Date &other)
{
  if (Day == other.Day && iMonth == other.iMonth && Year == other.Year)
        return true;
  else
        return false;
}
```

is declared as constant: `bool operator==(const Date &) const;`

Vectors (9)

```
int iArray[100];  // C-style array
for (int i = 0;  i < 100;  i++) cout << iArray[i] << endl;
```
or
```
for (int *p = &iArray[0];    p != &iArray[100];    p++)   cout << *p << endl;

vector<int> iVector(100); // C++ vector
for (int i = 0;  i < 100;  i++)     cout << iVector[i] << endl; // traditional mode
```
or
```
for (vector<int>::iterator it = iVector.begin();   it != iVector.end();  ++it)
    cout << *it << endl; // begin() returns iterator to the first element, end() to the
                         // first non-existing element.
```

An iterator is any object that, pointing to some element in an array or other range of elements, has the ability to iterate through the elements of that range using a set of operators (at least, the (++) increment and (*) dereference). In C-style array the simplest iterator is the pointer. For C++ vectors and other containers the iterators are objects of certain classes. There are several categories of iterators, but almost all of them have copy constructor and operator functions for ++, * , ->, =, == and *!*=. Thus, the iterator objects and ordinary pointers have the same set of functionalities. Otherwise, the iterator is a smart pointer.

Vectors (10)

Example:

```
vector<Date> Jan(31);
int i = 1;
for (vector<Date>::iterator it = Jan.begin(); it != Jan.end(); it++)
{ // or simply for (auto it = Jan.begin(); it != Jan.end(); it++)
    it->SetDay(i++); // or (*it).SetDay(i++);
    it->SetMonth(1);
    it->SetYear(2019);
}
```

As we have vectors, we may also write the same in traditional way:

```
vector<Date> Jan(31);
for (int i = 0;  i < 31;  i++)
{
    Jan.at(i).SetDay(i + 1);
    Jan.at(i).SetMonth(1);
    Jan.at(i).SetYear(2019);
}
```

However, there are containers in which the iterators are the only mode to access the container elements. As for vectors, inserting and removing of elements also need iterators.

Vectors (11)

const_iterator does not allow to change the elements to which it points. Example:

```
for (vector<Date>::const_iterator it = Jan.cbegin(); it != Jan.cend(); it++)
    cout << it->ToString() << endl;
```

where

```
char *Date::ToString() const
{ // remember how to introduce changing of attributes into constant member functions
     (const_cast<Date *>(this))->pText = new char[12]; // pText is the member of Date
     sprintf_s(pText, 12, "%02d %s %d", Day, Month, Year);
     return pText;
}
```

Methods of vector to get the needed iterators are:

1. *begin* and *cbegin*: return the *iterator* or *const_iterator* to the first element of vector.
2. *rbegin* and *crbegin*: return the *reverse_iterator* or *const_reverse_iterator* to the last element of vector.
3. *end* and *cend*: returns the *iterator* or *const_iterator* pointing to the theoretical element that follows the last element in the vector.
4. *rend* and *crend*: returns the *reverse_iterator* or *const_reverse_iterator* pointing to the theoretical element preceding the first element in the vector.

```
for (vector<Date>::const_reverse_iterator it = Jan.crbegin(); it != Jan.crend(); it++)
    cout << it->ToString() << endl; // decrementing of iterators is not supported
```

Vectors (12)

To add new elements into vector use method *insert*:

1. vector_name.insert(iterator_to_position, value_to_insert);
2. vector_name.insert(iterator_to_position, number_of_elements_to_insert, value_to_insert);
3. vector_name.insert(iterator_to_position, iterator_to_first_element_to_insert, iterator_to_first_element_not_to_insert);

Examples:

```
vector<int> vec(5, 0); // have 0, 0, 0, 0, 0
vec.insert(vec.begin() + 2, 1); // insert 1 to position 2
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // prints 0, 0, 1, 0, 0, 0
vec.insert(vec.begin() + 2, 3, 2); // insert three times 2 from position 2
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // prints 0, 0, 2, 2, 2, 1, 0, 0, 0
vec.insert(vec.begin() + 6, vec.begin() + 2, vec.begin() + 4);
                // takes elements from positions 2 and 3 (not 4!), inserts from position 6
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // prints 0, 0, 2, 2, 2, 1, 2, 2, 0, 0, 0
```

There is also one possibility to add an element without using iterators:

```
vector_name.push_back(value_to_append);
```

Example:

```
vec.push_back(3);
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // prints 0, 0, 2, 2, 2, 1, 2, 2, 0, 0, 0, 3
```

Vectors (13)

If the vector contains objects, it may be useful instead of *insert* use method *emplace*:
vector_name.emplace(iterator_to_position, value_to_insert);

Examples:
vector<Date> vec(5);
vec.insert(vec.begin() + 2, Date(15, 12, 2018)); // insert to position 2
The operation has 3 steps:
1. Create an anonymus object
2. Copy or move the anonymus object into vector
3. Destroy the anonymus object

vec.emplace(vec.begin() + 2, Date(15, 12, 2018)); // insert to position 2
Just one step – call the constructor and create the object inside vector

Similarly, it may be useful instead of *push_back* use method *emplace_back*:
vector_name.emplace_back(value_to_append);

Tests, however, show that the compiler implemented in Visual Studio handles the inserting and emplacing methods in identical way.

To increase performance set the supposed maximal length of vector beforehand:
vector_name.reserve(supposed_number_of_elements);

Vectors (14)

To completely reset the vector use method *assign*:

1. vector_name.assign(new_number_of_elements, initial_value_for_elements);
2. vector_name.assign(pointer_to_the_first_element_in_C-style_array, pointer_to_the_element_in_C-style_array_following_the_last_element);
3. vector_name.assign(iterator_to_the_first element, iterator_to_the_element_following_the_last_ element);

Examples:

```
vector<int> vec1(5, 0); // have 0, 0, 0, 0, 0
vec1.assign(3, 4);
for (auto it = vec1.begin(); it != vec1.end(); cout << *(it++)); // get 4, 4, 4
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vec1.assign(arr + 2, arr + 7);
for (auto it = vec1.begin(); it != vec.1end(); cout << *(it++)); // get 3, 4, 5, 6, 7
vec1.assign(vec1.begin() + 2, vec1.begin() + 4);
for (auto it = vec1.begin(); it != vec1.end(); cout << *(it++)); // get 5, 6
vector<int> vec2; // empty
vec2.assign(arr, arr + 9);
for (auto it = vec2.begin(); it != vec2.end(); cout << *(it++)); // get 1, 2, 3, 4, 5, 6, 7, 8, 9
vec1.assign(vec2.begin() + 2, vec2.begin() + 5);
for (auto it = vec1.begin(); it != vec1.end(); cout << *(it++)); // get 3, 4, 5
```

Vectors (15)

There is an additional possibility for initializing a vector:

```
vector<type_of_elements> vector_name = { sequence_of_initial_values };
```

Example:

```
vector<int> vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

This is the same as

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vec.assign(arr, arr + 10);
```

Examples:

```
vector<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
```

or

```
vector<Date> *pChristmas = new vector<Date>{ Date(24, 12, 2019), Date(25, 12, 2019),
Date(26, 12, 2019) };
```

or

```
vector<Date *> *pChristmas = new vector<Date *>{ new Date(24, 12, 2019), new Date(25,
12, 2019), new Date(26, 12, 2019) };
```

Vectors (16)

To remove from the vector use method *erase*:

1. vector_name.erase(iterator_to_position);
2. vector_name.erase(iterator_to_first_element_to_remove,
   iterator_to_first_element_not_to_remove);

Examples:

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vec.assign(arr, arr + 10);
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // get 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
vec.erase(vec.begin() + 3);
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // get 1, 2, 3, 5, 6, 7, 8, 9, 10
vec.erase(vec.begin() + 3, vec.begin() + 6);
for (auto it = vec.begin(); it != vec.end(); cout << *(it++)); // get 1, 2, 3, 8, 9, 10
```

To remove all the data stored in vector use method *clear*:

```
vector_name.clear();
```

To remove the last element use method *pop_back*:

```
vector_name.pop_back();
```

Vectors (17)

```
vector<Date> week1(7);
week1.erase(week1.begin() + 3); // destructor for week[3] is also called
week1.clear(); // destructors for all the members are called

vector<Date> *pWeek1(7);
pWeek1->erase(pWeek1->begin() + 3); // destructor for week[3] is also called
pWeek1->clear(); // destructors for all the members are called

vector<Date *> week2 (7); // contains zero pointers
for (int i = 0;  i < 7;  i++)
    week2[i] = new Date(6 + i, 1, 2020);
week2.erase(week2.begin() + 3); // error, destructor for week[3] is not called
delete *(week2.begin() + 3); // alternative: delete week2[3]
```

After that erase.

```
week2.clear(); // error, destructors for members are not called
for (auto it = week2.begin();  it != week2.end();  it++)
    delete *it;
week2.clear(); // now correct
```

Alternative solution

```
 for (int i = 0;  i < 6;  i++)
    delete week2[i];
```

Remark: compare with slides *Vectors(3)…Vectors(6)*

Vectors (18)

```
vector<Date *> *pWeek2 = new vector<Date *>(7);.
for (int i = 0;  i < 7;  i++)
    pWeek2->at(i) = new Date(6 + i, 1, 2020);
pWeek2->erase(pWeek2->begin() + 3); // error, destructor for week[3] is not called
delete *(pWeek2->begin() + 3);
```

Alternatives:

```
delete pWeek2->at(3);
```

or

```
delete (*pWeek2)[3];
pWeek2->clear(); // errors, destructors for members are not called
for (auto it = pWeek2-> begin();  it != pWeek2-> end();  it++)
    delete *it;
```

Alternative solutions

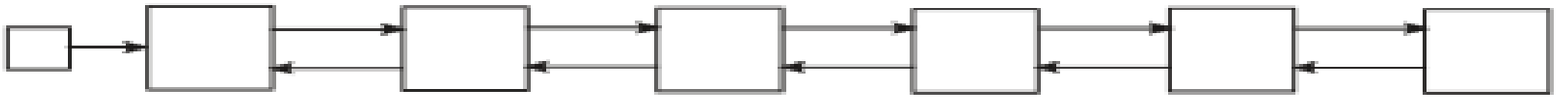
```
 for (int i = 0;  i < 6;  i++)
    delete pWeek2->at(i);
```

or

```
 for (int i = 0;  i < 6;  i++)
    delete (*pWeek2)[i];
pWeek2-> clear();
```

Lists (1)

Container *list* implements data structure called as doubly linked list:



A *list* is defined as follows:

list<type_of_elements> list_name(number_of_elements, initial_value)

or

list<type_of_elements> *pointer_name = new list<type_of_elements>
(number_of_elements, initial_value)

or

list<type_of_elements> list_name = { sequence_of_initial_values };

The initial value is optional. If it is not present the elements are initialized to zero or as the objects are constructed by default (having no arguments) constructor.

Examples:

```
#include <list> // See www.cplusplus.com/reference/list/list/
using namespace std;
list<Date> January(31, Date(1, 1, 2019));
list<Date *> *pJanuary = new list<Date *>(31, nullptr);
delete pJanuary; // not delete[]
list<int> list_int = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; // initializing with sequence
```

Lists (2)

There are no indexes in lists. To access list elements you may use methods *front* and *back* (identical to the corresponding methods of *vector*) or apply the iterators. The list iterators support increment and decrement, but not addition and subtraction of an integer. Example:

```
list<Date> Jan(31);
int i = 1;
for (list<Date>::iterator it = Jan.begin(); it != Jan.end(); it++)
{
    it->SetDay(i++); // or (*it).SetDay(i++);
    it->SetMonth(1);
    it->SetYear(2019);
}
```

To access an inner element of list we have to travel from element to element until the needed one. Example:

```
Date Jan_3;
for (auto it = Jan.begin(); it != Jan.end(); it++)
{
  if (it->GetDay() == 3)
  {
      Jan_3 = *it;
      break;
  }
}
```

Lists (3)

Methods of list operating as the corresponding methods of vector:

- *copy constructor, operator=*
- *size, resize, empty*
- *push_back, pop_back, emplace_back*
- *begin, cbegin, rbegin, crbegin*
- *end, cend, rend, crend*
- *insert, emplace, assign* (without retrieving values from C-style array)
- *erase, clear*

It is also possible to add elements to the beginning of list (methods *push_front, emplace_front)* and remove the first element (method *pop_front*).

Example:

```
list<Date> deadlines(5);
int i = 0;
for (auto it = deadlines.begin(); it != deadlines.end(); ++it, i++) {
   // as deadlines.begin() + 3 is not allowed, we have to travel to the point of insertion
   //  stepping from element to element
  if (i == 3)  {
       deadlines.insert(it, Date(2, 3, 2019));
       break;
  }
}
```

Lists (4)

Method *splice* is for transferring elements from one list into another:

1. list_name.splice(iterator_to_position, another_list_to_insert_completely);
2. list_name. splice(iterator_to_position, another_list,
   iterator_to_the_element_to_insert);
3. list_name. splice(iterator_to_position, another_list,
   iterator_to_first_element_to_insert, iterator_to_first_element_not_to_insert);

The spliced elements are removed from their original list.

Example:

```
list<int> list1, list2;
list1.assign(5, 1); // get 1, 1, 1, 1, 1
list2.assign(2, 2); // get 2, 2
int i = 0;
for (auto it = list1.begin(); it != list1.end(); ++it, i++)
{
    if (i == 2)
    {
        list1.splice(it, list2); // insert list2 into list1
        break; // get 1, 1, 2, 2, 1, 1, 1
    }
}
cout << boolalpha << list2.empty(); // true
```

Lists (5)

Example:

```
list<int> list3 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }, list4 = { 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 };
int i = 0, j = 0;
for (auto it3 = list3.begin(); it3 != list3.end(); ++it3, i++)
{ // we want to insert the underlined members from list4 into list 3 from position 2
  if (i == 2)
 { // it3 points now to position 2 in list 3
        list<int>::iterator it_start, it_end;
        for (auto it4 = list4.begin(); it4 != list4.end(); ++it4, j++)
       {
            if (j == 5) // it4 points now to position 5 in list4
                it_start = it4;
            else if (j == 8) // it4 points now to position 8 list4
           {
               it_end = it4;
               break;
           }
       }
       list3.splice(it3, list4, it_start, it_end); // list3 is 1, 2, 60, 70, 80, 3, 4, 5, 6, 7, 8, 9, 10
       break; // list4 is 10, 20, 30, 40, 50, 90, 100
      }
}
```

Lists (6)

Other specific methods of container *list*:

1. Sorts the list (possible only if in member objects *operator*< and *operator*== methods are implemented):
   void list_name.sort();
   Example:
   list<string> list1 = { "John", "James", "Mary", "Elizabeth" };
   list1.sort(); // get Elizabeth, James, John, Mary
2. Merges two lists, they both must be sorted. The result is also sorted:
   void list_name.merge(another_list);
   The merged list loses all its members.
   Example continues:
   list<string> list2 = {"Benjamin", "John", "Timothy", "Walter" };
   list1.merge(list2); // get Benjamin, Elizabeth, James, John, John, Mary, Timothy, Walter
   cout << boolalpha << list2.empty() << endl; // prints true
3. Removes duplicate elements:
   void list_name.unique();
   Example continues:
   list1.unique(); // get Benjamin, Elizabeth, James, John, Mary, Timothy, Walter

Lists (7)

4. Removes the specified element:
   void list_name.remove(element_to_remove);
   Does not throw exceptions and does not destroy the removed member.
   Example continues:
   list1.remove("John"); // get Benjamin, Elizabeth, James, Mary, Timothy, Walter
   Method *erase()* removes by iterator, method *remove()* by element.
5. Removes elements that satisfy the specified condition:
   void list_name.remove_if(predicate);
   The predicate may be a pointer to function, functor or lambda expression. If the predicate for an element returns *true*, this element will be removed. Does not throw exceptions and does not destroy the removed member.
   Example continues:
   list1.remove_if([](const string& s) { return s == "Elizabeth"; });
   // get Benjamin, James, Mary, Timothy, Walter
6. Reverses the order of elements:
   void list_name.reverse();
   Example continues:
   list1.reverse(); // get Walter, Timothy, Mary, James, Benjamin

Lists (8)

Do not forget that the list sorting algorithm compares only the members. If the members are pointers, the pointers (and not the objects to which they are pointing) are compared. Example:

```
list<Date *> *pDeadlines = new list<Date *> {
            new Date(9, 1, 2020), new Date(24, 12, 2019) };
pDeadlines->sort();
```

has no the supposed effect.

Solution:

```
list_name.sort(comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its arguments must be elements of list. The body of comparator must check whether the first argument is considered to go before the second (return value *true*) or not (return value *false*).

Example: if in class *Date* method *bool operator<(const Date &) const* is implemented,

```
pDeadlines->sort( [](Date *pd1, Date *pd2)->bool { return *pd1 < *pd2; } );
```

works.

Initializer lists (1)

The initializer list discussed here is not the attribute initializer list (called also as member initializer list) presented in chapter *Advanced C++* on slide *Initializing (3).*

An 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 www.cplusplus.com/reference/initializer_list/initializer_list/
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;
}
```

Range-based *for* loop (1)

for (loop_variable_declaration : range) { body}

The range can be any sequence: a C-style array, vector, list or other container, etc. The only condition is that there must be tools (pointers, iterators) to travel from the beginning of sequence to the end. The loop variable and the members of sequence must be of the same type.

Examples:

```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int i : vec)
    cout << i << " ";

list<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
for (Date d : christmas)
    cout << d.ToString() << endl;
// Compare with:
// for (auto it = christmas.begin(); it != christmas.end(); it++)
//     cout << it->ToString() << endl;

int arr[] = { 1, 2, 3, 4, 5 };
for (int i : arr)
    cout << i << " ";

for (int i : { 1, 2, 3, 4, 5 })
    cout << i << " ";
```

Range-based *for* loop (2)

But:

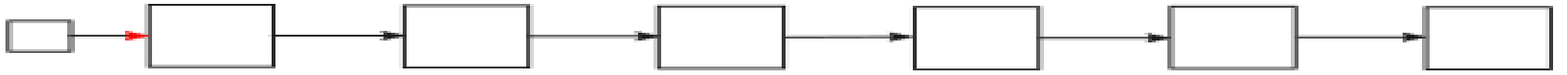
```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int i : vec)
   i++; // formally correct
for (int i : vec)
   cout << i << " "; // still 1, 2, 3, 4, 5
for (int &i : vec)
   i++;
for (int i : vec)
   cout << i << " "; // now 2, 3, 4, 5, 6
```

The reason is that the statements in the loop body use local copies of the elements from range. To avoid it and also to avoid calling of copy constructor and destructor specify the loop variable as reference. Similarly:

```
list<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
for (Date d : christmas)
    d.SetYear(2020);
for (Date d : christmas)
    cout << d.ToString() << endl; // still 2019
 for (Date &d : christmas)
    d.SetYear(2020);
 for (Date &d : christmas)
    cout << d.ToString() << endl; // now 2020
```

Forward lists (1)

Container *forward_list* implements data structure called as singly linked list:



The *forward list* is very similar to *list*. The main differences is that there is no moving backwards. Therefore methods *rbegin*, *crbegin*, *rend*, *crend*, *pop_back*, *push_back*, *emplace_back*, *back* are missing (but there are methods *pop_front*, *push_front*, *emplace_front*). Inserting and removing are implemented in slightly different way: instead of methods *insert*, *emplace*, *splice* and *erase* the forward list uses methods *insert_after*, *emplace_after*, *splice_after* and *erase_after*. In those functions the first parameter (i.e the iterator pointing to position to insert or remove) points to the position preceding the position to insert or erase.
Example:

```
#include<forward_list>
    // see http://www.cplusplus.com/reference/forward_list/forward_list/
using namespace std;
forward_list<Date> January(31, Date(1, 1, 2019));
forward_list<int> list = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Forward lists (2)

Example:

```
forward_list<int> list1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (auto it = list1.begin(); it != list1.end(); ++it)
{
    if (*it == 2)
    {
        list1.insert_after(it, 11); // insert after value 2
        break; // get 1, 2, 11, 3, 4, 5, 6, 7, 8, 9, 10
    }
}
```

To insert or remove starting from the first position we need method *before_begin*, returning iterator to an non-existing element on position (-1). Example:

```
forward_list<int> list3 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
list3.insert_after(list3.before_begin(), 20); // get 20, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
list3.erase_after(list3.before_begin()); // get 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Method *size* is not implemented. To get the number of elements in *forward_list* use method *std::distance*, for example:

```
cout << distance(list1.begin(), list1.end()) << endl; // prints 10
```

Forward lists (3)

```
forward_list<int> list2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (auto it2 = list2.begin(); it2 != list2.end(); ++it2)
{ // compare with slide Lists(5)
    if (*it2 == 2) // insert after 2
   {
       forward_list<int>::iterator it_start, it_end;
       for (auto it3 = it2; it3 != list2.end(); ++it3)
      {
           if (*it3 == 5) // take 5, not element after 5 (i.e. as is in list)
               it_start = it3;
           else if (*it3 == 8) // take element before 8 as in list
          {
               it_end = it3;
               break;
           }
       }
       list2.splice_after(it2, it_start, it_end);
       break; // get 1, 2, 5, 6, 7, 3, 4, 5, 6, 7, 8. 9, 10
      }
}
```

Queues

Container *queue* implements abstract data type FIFO (first in, first out). The user of queue can access only the first and the last element. The new element can be added only to the end of queue. It is possible to remove only the first element.
As there is no travelling from one element to another, the queue does not have iterators. The only methods the queue supports are *empty*, *size*, *front*, *back*, *push* (like *push_back*), *pop* (like *pop_front*), *emplace* (like *emplace_back*).

Example:

```
#include <queue> // see http://www.cplusplus.com/reference/queue/queue/
queue<int> q; // initializing with sequence is not possible
q.push(1);
q.push(2);
q.push(3);
cout << q.front() << ", " << q.back() << endl; // get 1, 3
q.pop(); // does not return the removed element, use front to get a copy
cout << q.front() << ", " << q.back() << endl; // get 2, 3
```

Priority queues

Container *priority_queue* implements abstract data type priority queue. The element at the head (the only element that can be removed) has the highest priority. The order of other elements is arbitrary.

The priority is defined by *operator*<: object that is less has lower priority

Priority queue supports methods *empty*, *size*, *top (*access the head*)*, *pop* (remove the head), *push* (insert into some place in queue), *emplace* (insert into some place in queue).

Example:

```
#include <queue> // see http://www.cplusplus.com/reference/queue/priority_queue/
priority_queue<int> q; // initializing with sequence is not possible
q.push(1);
q.push(2);
q.push(3);
cout << q.top() << endl; // get 3
q.pop(); // does not return the removed element, use method top to get a copy
cout << q.top() << endl; // get 2
```

Stacks

Container *stack* implements abstract data type LIFO (last in, first out). The user of stack can access only the first element. The new element can be added only to the beginning of stack. It is possible to remove only the first element.
The only methods the stack supports are *empty*, *size*, *top* (like *front*), *pop* (like *pop_front*), *push* (like *push_front*), *emplace* (like *emplace_front*).

Example:

```
#include <stack> // see http://www.cplusplus.com/reference/stack/stack/
stack<int> s; // initializing with sequence is not possible
s.push(1);
s.push(2);
s.push(3);
cout << s.top() << endl; // get 3
s.pop(); // does not return the removed element, use top to get a copy
cout << s.top() << endl; // get 2
```

Deques

Container *deque* implements abstract data type double-ended queue. Those containers can be expanded and contracted on both ends: it is allowed to insert a new element to begining and to the end and also remove the first and last element.
Deque is almost identical with vectors. It has methods *push_front*, *pop_front* and *emplace_front* missing in vector. Read https://www.geeksforgeeks.org/deque-vs-vector-in-c-stl/

Example:

```
#include <deque> // see http://www.cplusplus.com/reference/deque/deque/
deque<Date> christmas = { Date(25, 12, 2019) };
christmas.push_front(Date(24, 12, 2019));
christmas.push_back(Date(26, 12, 2019));
```

Arrays

Container *array* is like vector but its size is fixed. Inserting and removing of elements is not possible.

Example:

```
#include <array> // see http://www.cplusplus.com/reference/array/array/
array<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
// error, the length of array is not specified
```

compare with

```
vector<Date> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
```

Array template has two parameters: type and size

```
array<Date, 3> christmas = { Date(24, 12, 2019), Date(25, 12, 2019), Date(26, 12, 2019) };
array<int, 5> arr1; // correct, get empty array for 5 integers
array<int, 5> arr2(0); // error, setting of initial value in this mode is not possible
arr1.fill(0);  // fill method sets all the elements of array to specified value
```

Array supports all the iterators and methods presented on slides *Vectors (2)*, *Vectors (3)* and *Vectors (7)* (except *resize*). Examples:

```
for (int i = 0;  i < 5;  i++)
    cout << arr[i] << ' '; // or arr.at(i)
for (int i : arr)
    cout << i << ' ';
```

Value arrays (1)

Value arrays are containers designed for storing numeric values. Examples:

```
#include <valarray> // see http://www.cplusplus.com/reference/valarray/valarray/
valarray<int> va1; // empty
valarray<double> va2(10); // initialized with zeroes
valarray<int> va3 = { 1, 2, 3 }; // va3[0] is 1, va3[1] is 2, va3[2] is 3
initializer_list<int> il = { 4, 5, 6 };
valarray<int> va4(il); // va4[0] is 4, va4[1] is 5, va4[2] is 6
```

There are also copy and move constructors. The large set of operator functions allows very efficiently to operate with value arrays. Some examples:

```
valarray<int> va1 = { 1, 2, 3 }, va2 = { 4, 5, 6 };
valarray<int> va3 = va1 + va2; // get value array 5, 7, 9
valarray<int> va4 = -va1; // get value array -1, -2, -3
valarray<int> va5 = va1 - va2; // get value array -3, -3, -3
valarray<int> va6 = va1 * va2; // get value array 4, 10, 18
va6 += va2; // get value array 8, 15, 24
valarray<int> va7 = va6 - 1; // get value array 7, 14, 23
va7 -= 2; // get value array 7, 14, 23
va7 = 2; // get value array 2, 2, 2
valarray<double> va8 = { 16, 25, 36 };
valarray<double> va9 = sqrt(va8); // get value array 4, 5, 6
```

Value arrays (2)

Examples about methods implemented for value arrays:

```
valarray<int> va10 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
cout << va10.min() << ' ' << va10.max() << ' ' << va10.sum() << endl; // prints 1, 9, 45
valarray<int> va11 = va10.shift(3); // get 4, 5, 6, 7, 8, 9, 0, 0, 0
valarray<int> va12 = va10.shift(-3); // get 0, 0, 0, 1, 2, 3, 4, 5, 6
valarray<int> va13 = va10.cshift(3); // get 4, 5, 6, 7, 8, 9, 1, 2, 3 (circular shift)
valarray<int> va14 = va10.cshift(-3); // get 7, 8, 9, 1, 2, 3, 4, 5, 6
```

For non-standard operations use method *apply()*:

```
valarray<int> va15 = va10.apply([](int i) { return i % 2 ? i : -i; }); // get 1, -2, 3, -4, 5, -6, 7, -8, 9
```

Value array is a dynamic array (i.e. its size is changeable):

```
cout << va15.size() << endl; // prints 9
va15.resize(5); // get 0, 0, 0, 0, 0
va15.resize(3, 10); // get 10, 10, 20
```

Value array objects do not have iterators. But there are standard functions *std::begin()* and *std::end()* taking *valarray* objects as arguments:

```
for (auto it = begin(va10);  it != end(va10);  it++)
{
    cout << *it << ' ';
}
```

Value arrays (3)

A new value array may be created from an existing value array by selecting the elements to include. For that we need slice selector:

slice slice_name(starting_index, size, stride);

Example (see also http://www.cplusplus.com/reference/valarray/slice/):

```
valarray<int> va16(20);
int j = 0;
for (int &i : va16) {
    i = ++j; // get 1, 2, 3, …., 20
}
slice slicer(1, 3, 5);
valarray<int> va17 = va16[slicer]; // get 2, 7, 12
```

Comment: as the starting index is 1, the first element of selection is *va16[1]*. As the stride (i.e. step) is 5, the following elements of selection are *va16[6]*, *va16[11]*, … The size of selection here is 3.

Selection can be provided also with mask.

Example (see also http://www.cplusplus.com/reference/valarray/mask_array/):

```
valarray<bool> mask(20);
mask[5] = mask[7] = mask[11] = true;
valarray<int> va18 = va16[mask]; // get 6, 8, 12
```

Strings as containers

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 http://www.cplusplus.com/reference/string/string/

Variant (1)

Variants introduced in C++ version 17 are to replace unions from classical C.

```
typedef union { double d;  int i; } UN;
UN un1 = { 10.0 }; // union contains a double value
UN un2 = { 20 };   // union contains an integer value
```

But

```
typedef union { string s;  int i; } UN;
UN un1 = { "Hello" }; // compile error, unions with non-trivial members do not work
```

The corresponding variant:

```
#include <variant> // see https://en.cppreference.com/w/cpp/utility/variant
variant<string, int> vr;
```

The variant is not empty: if the initial value is not specified, the default constructor of the first type is called. Here *vr* contains empty string.

Examples about defining and initializing of variants:

```
variant<string, int > vr1 = "Hello", vr2{ "Hello" }, vr3 = 20, vr4{ 20 };
variant<vector<int>, vector<double>> vr5 = vector<int> { 1, 2, 3 };
```

Later we can reset the value, for example:

```
vr3 = "Goodbye";
vr2 = 20;
```

Variant (2)

To know what is the type of value stored in variant use method *index()*:

```
variant<string, int > vr1 = "Hello", vr2 { "Hello" }, vr3 = { "Hello" }, vr4{ 20 };
cout << vr1.index() << endl; // prints 0
cout << vr4.index() << endl; // prints 1
```

There are also several function templates:

```
if (holds_alternative<string>(vr1))
    vr1 = "Good morning";
if (get_if<string>(&vr2))
    vr2 = "Good evening";
if (get_if<0>(&vr3))
    vr3 = "Good day";
```

To retrieve the value of type *T* use function template *get<T>()*:

```
cout << get<string>(vr1) << endl; // prints "Good morning"
cout << get<0>(vr2) << endl; // prints "Good evening"
```

But:

```
cout << get<int>(vr1) << endl; // throws exception bad_variant_access
cout << get<1>(vr1) << endl;   // throws exception bad_variant_access
```

Variant (3)

Let us have

```
variant<Date, Time> vr;
vr = Date(25, 10, 2021);
cout << get<Date>(vr).GetDate() << endl; // prints 25
```

```
Date d = get<Date>(vr);
```

Now d is the copy of Date stored in variant.

```
d.SetDay (27);
cout << get<Date>(vr).GetDate() << endl; // still prints 25
```

```
get<Date>(vr).SetDay(26);
cout << get<Date>(vr).GetDate() << endl; // prints 26
```

```
vr = d; // replace the value in variant
cout << get<Date>(vr).GetDate() << endl; // prints 27
```

An alternative is to use method emplace:

```
vr.emplace<Date>(28, 10, 2021);
cout << get<Date>(vr).GetDate() << endl; // prints 28
```

Variant (4)

Sometimes the initialization is ambiguous, for example:

```
variant<unsigned char, char> vr {100}; // which of the alternatives?, error
```

To help the compiler, use templates *in_place_type* or *in_place_index*, for example

```
#include <utility>
variant<unsigned char, char> vr1 { in_place_type<char>, 100 };
variant<unsigned char, char> vr2 { in_place_index<0>, 100 };
```

If two variants have the same alternatives in the same order, their comparison is possible:

```
variant<string, int > vr1 { "Hello" }, vr2 { "Hello" };
if (vr1 == vr2)
    cout << "Identical" << endl;
```

Variants are excellent for creating heterogeneous collctions. Example:

```
class Book { …. };
class Article { …. };
class Link { …. };
vector<variant<Book, Article, Link>> Entries;
Entries.push_back(Book ("Nicolai Josuttis", "Complete C++ 17", "978-3-96730-017-8",
2020)); // insert an object of class Book into vector
Entries.push_back(Link("Bartolomiej Filipek", "Everything you need to know about
std::variant from C++ 17", " https://www.bfilipek.com/2018/06/variant.html "));
                // insert an object of class Link into the same vector
```

Variant (5)

The values in a variant may be processed using visitors and standard function std::visit(). A visitor is a callable object that is able to accept arguments of any type defined in the current variant. The simplest visitor is a functor, for example:

```
class Visitor
{
  public:
      void operator() (Book b) { ….. } // process somehow an object of class Book
      void operator()  (Article a) { ….. }
      void operator()  (Link l) { ….. }
};
vector<variant<Book, Article, Link>> Entries;
for (variant<Book, Article, Link> v : Entries)
{
     visit(Visitor(), v); // for each element in vector the appropriate processing function is called
}
```

Pairs (1)

A *pair* groups together two values. In most cases those values are of different types:

```
pair <type_1, type_2> pair_name(value_1, value_2);
```

or

```
pair <type_1, type_2> pair_name;
```

Any pair has two public attributes: *first* and *second*.

Examples:

```
#include <utility> // see http://www.cplusplus.com/reference/utility/pair
pair<string, double> item("shirt", 12.49);
cout << item.first.c_str() << ' ' << item.second << endl; // prints "shirt 12.49"
item.first = "cap"; // change attribute values
item.second = 2.49;
cout << item.first.c_str() << ' ' << item.second << endl; // prints "cap 2.49"
pair<string, Date> deadline; // as initial values are not specified, default constructors are called
```

There is another way to construct a pair – use method *make_pair*:

```
pair <type_1, type_2> pair_name = make_pair(value_1, value_2);
```

It is more convinient, because we may use *auto*. Example:

```
auto item = make_pair("shirt", 12.5); // item is of type pair<const char *, double>
```

but

```
auto item("shirt", 12.49); // error, not able to guess the type
```

Copy constructor is also implemented. Example:

```
auto item1 = item;
```

Pairs (2)

Initial values of pair elements may be presented by other variables, pointers or references.
Example:

```
string string1 = "Shirt";
double price = 12.49;
pair<string, double> item(string1, price);
```

but

```
price = 10.49;
cout << item.first << ' ' << item.second << endl; // still Shirt 12.49
```

Solution:

```
pair<string &, double &> item1(string1, price);
string1 = "Cap";
price = 2.49;
cout << item1.first << ' ' << item1.second << endl; // prints Cap 2.49
```

Alternative solution:

```
pair<string *, double *> item2(&string1, &price);
cout << *item2.first << ' ' << *item2.second << endl; // prints Shirt 12.49
string1 = "Cap";
price = 2.49;
cout << *item2.first << ' ' << *item2.second << endl; // prints Cap 2.49
```

Pairs (3)

Pairs include operator functions for relational operations (*operator*==, *operator*<, etc.):

- Members *first* are compared
- If it is not enough to make the decision, members *second* are compared

Of course, the members itself must support relational operations.

Examples:

```
pair<string, Date> deadline1("ExamMath", Date(5, 1, 2019));
pair<string, Date> deadline2("ExamMath", Date(5, 1, 2019));
cout << boolalpha << (deadline1 == deadline2) << endl; // true
pair<string, Date> deadline3("ExamMath", Date(6, 1, 2019));
cout << boolalpha << (deadline3 < deadline2) << endl;   // false
pair<string, Date> deadline4("ExamChemistry", Date(5, 1, 2019));
cout << boolalpha << (deadline4 < deadline2) << endl; // true
```

Remark: in those examples *Date::operator*== and *Date::operator*< must be constant methods:

```
bool Date::operator<(const Date &other) const
{
  if (Year != other.Year)
      return Year < other.Year;
  if (iMonth != other.iMonth)
     return iMonth < other.iMonth;
  return Day < other.Day;
}
```

Tuples (1)

Tuples are generalizations of pairs: they can store any number of values. In most cases those values are of different types:

```
tuple <type_1, type_2, …., type_n> tuple_name(value_1, value_2, ……,value_n);
```

Initial values are optional:

```
tuple <type_1, type_2, …., type_n> tuple_name;
```

Example:

```
#include <tuple> // see http://www.cplusplus.com/reference/tuple/
tuple<long long int, string, string, double> student(123456789LL, "John", "Smith", 4.25);
```

Tuples are similar to *structs* from classical C. Difference: the elements of a tuple have indeces starting from 0 but no names. To retrieve a member of tuple use standard template *get*:

```
get<index>(tuple_name);
```

Examples:

```
cout << get<0>(student) << ' ' << get<1>(student) << ' ' << get<2>(student) << ' ' <<
get<3>(student) << endl;
cout << typeid(get<0>(student)).name() << endl; // prints __int64
```

Initial values of tuple elements may be presented by other variables. Example:

```
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
tuple<string, string, double> item(string1, string2, price);
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
```

Tuples (2)

However:

```
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
tuple<string, string, double> item(string1, string2, price);
price = 59.99;
cout << get<2>(item) << endl; // still 49.99
```

To modify the values in a tuple we must use references or pointers:

```
tuple<string, string &, double &> item(string1, string2, price);
price = 59.99;
string2 = "Lee";
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // now Jeans Lee 59.99
```

or

```
string *pCompany = new string("Wrangler");
tuple<string, string *, double *> item(string1, pCompany, &price);
cout << get<0>(item) << ' ' << *get<1>(item) << ' ' << *get<2>(item) << endl;
                                                    // Jeans Wrangler 49.99
price = 45.99;
pCompany = new string("Arizona");
cout << get<0>(item) << ' ' << *get<1>(item) << ' ' << *get<2>(item) << endl;
                                                    // now Jeans Arizona 45.99
```

Tuples (3)

There is another way to construct a tuple – use method *make_tuple*:

tuple <type_1, type_2, …., type_n> tuple_name = make_tuple(value_1, value_2, …….,value_n);

It is more convinient, because we may use *auto*. Example:

```
auto item = make_tuple(10, 20, 30); // item is tuple<int, int, int>
```

or

```
tuple<string, string, double> item;
item = make_tuple("Jeans", "Wrangle", 49.99);
```

but

```
auto item = make_tuple("Jeans", "Wrangle", 49.99); // error, not able to guess the type
```

However,

```
string *pCompany = new string("Wrangler");
double price = 49.99;
tuple<string, string *, double *> item;
item = make_tuple("Jeans", pCompany, &price); // correct, works
```

but we cannot use references:

```
tuple<string &, double &> item; // error
string string2 = "Wrangler";
double price = 49.99;
auto item = make_tuple(string2, price); // correct, but we cannot tell that the
                                        // parameters must be references
```

Tuples (4)

The solution is to use utility functions *ref* and / or *cref*:

```
#include <functional>
string string1 = "Jeans", string2 = "Wrangler";
double price = 49.99;
auto item = make_tuple(ref(string1), ref(string2), ref(price));
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // prints "Jeans Wrangler 49.99"
string2 = "Lee";
price = 59.99;
cout << get<0>(item) << ' ' << get<1>(item) << ' ' << get<2>(item) << endl;
                                                    // prints "Jeans Lee 59.99"
```

reference_wrapper is a class template thar wraps a reference into an object. This object can be copied and assigned. Function *ref* returns such an object from the proper class. *cref* is for constant references.

Tuples include operator functions for relational operations (*operator*==, *operator*<, etc.), for example:

```
tuple<string, string, double> item1 = make_tuple("Jeans", "Wrangler", 49.99),
                                  item2 = make_tuple("Jeans", "Lee", 59.99);
cout << boolalpha << (item1 == item2) << endl; // prints false
```

Tuples (5)

Tuples include also *operator=* for assignment, for example:

```
tuple<int, int, int> t1(1, 1, 1);
tuple<double, double, double> t2(0, 0, 0);
t2 = t1; // Assignes the values of t1 to t2. If the assignment of one or more values is not
         // possible or the number of members is different, we get compile error
cout << get<0>(t2) << ' ' << get<1>(t2) << ' ' << get<2>(t2) << endl; // 1 1 1
cout << typeid(get<0>(t2)).name() << endl; // still double
```

Two different tuples can be concatenated into one with standard function tuple_cat. Example:

```
tuple<string> item1("Jeans");
tuple<string, double> item2("Wrangler", 49.99);
auto item3 = tuple_cat(item1, item2);
cout << get<0>(item3) << ' ' << get<1>(item3) << ' ' << get<2>(item3) << endl;
                                                        // Jeans Wrangler 49.99
```

Maps (1)

A *map* (corresponds to abstract data type dictionary) stores key-value pairs. The elements are sorted by keys order. Insertion, removing and access are based on keys. The keys must be unique. In memory the maps are implemented as balanced binary trees.

A *map* is defined as follows:

```
map<type_of_key, type_of_value> map_name = { list_of_pairs_of_initial_values }
```

or

```
map<type_of_key, type_of_value> *pointer_name  =
    new map<type_of_key, type_of_value> { list_of_pairs_of_initial_values }
```

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

Examples:

```
#include <map> // See www.cplusplus.com/reference/map/map/
using namespace std;
map<string, Date> deadlines = {
            { "Mathematics", Date(5, 1, 2019) },
            { "Chemistry", Date(10, 1, 2019) },
            { "Physics", Date(15, 1, 2019) }
};
map<string, Date> *pDeadlines = new map<string, Date>;
delete pDeadlines;
```

Maps (2)

To insert a new element, use method *insert*:

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

or

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

The return value is a pair in which member *first* is an 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* pointing to the key and *second* pointing to the value.

Examples (map *deadlines* was defined on the previous slide):

```
auto ret1 = deadlines.insert({ "Programming", Date(20, 1, 2019) });
cout << boolalpha << ret1.second << endl; // prints true
cout << (ret1.first->first).c_str() << endl;
// prints "Programming" (ret1.first->first is the inserted key)
cout << (ret1.first->second).ToString() << endl;
// prints "20 Jan 2019" (ret1.first->second is the inserted value)
auto ret2 = deadlines.insert(make_pair(string("Mathematics"), Date(6, 1, 2019)));
cout << boolalpha << ret2.second << endl; // prints false
cout << (ret2.first->first).c_str() << ' ' << (ret2.first->second).ToString() << endl;
  // prints "Mathematics 05 Jan 2019" (the old value)
```

Maps (3)

There is another way to insert using *operator[]*:

map_name[new_key] = value;

If the specified key is not a new one, the value in the pair is replaced.

Example:

```
deadlines["Cybernetics"] = Date(25, 1, 2019); // inserts new element
deadlines["Mathematics"] = Date(6, 1, 2019); // never fails, replaces value in existing element
```

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

```
for (auto it = deadlines.begin();  it != deadlines.end();  ++it)
    cout << (it->first).c_str() << ' ' << (it->second).ToString() << endl;
```

or

```
for (auto &x : deadlines)
    cout << x.first.c_str() << ' ' << x.second.ToString() << endl;
```

The results are printed in sorted order: *Chemistry*, *Cybernetics*, *Mathematics*, *Physics*, *Programming*.

Maps (4)

There are several possibilities to access and modify the members of map.

Method *find* returns map iterator to the member with specified key or in case of failure iterator *map_name.end()*:

```
auto return_value_name = map_name.find(key);
```

Examples:

```
auto it = deadlines.find("History");
if (it == deadlines.end())
   cout << "Not found" << endl;
else
  cout << (it->first).c_str() << ' ' << (it->second).ToString() << endl;
    // prints "Not found"
auto it = deadlines.find("Mathematics");
if (it == deadlines.end())
   cout << "Not found" << endl;
else
  cout << (it->first).c_str() << ' ' << (it->second).ToString() << endl;
    // prints "Mathematics 06 Jan 2019"
```

The returned iterator may be used for modifying the value:

```
 it->second = Date(7, 1, 2019);
```

Modifying the key, however, is not possible:

```
it->first = string("Linear algebra"); // compile error
```

Maps (5)

In addition to inserting *operator[]* can also be used for accessing and modifying:

```
reference_to_value = map_name[key]; // to get value corresponding to the specified key
map_name[key] = new_value; // to replace the value with new one
```

Examples:

```
deadlines["Mathematics"] = Date(8, 1, 2019);
cout << deadlines["Mathematics"].ToString() << endl; // prints "08 Jan 2019"
```

Here, if the member with specified key does not exist, it will be always created (constructor without arguments is used) and inserted. Therefore, use *operator[]* for accessing and modifying only if you are sure that the member exists.

At last, the value of a member may be accessed or replaced with method *at*:

```
reference_to_value = map_name.at(key); // to get value corresponding to the specified key
map_name.at(key) = new_value; // to replace the value with new one
```

The key must refer to an existing element. If the element does not exist, method *at* throws *out_of_range* exception.

Examples:

```
try {
    deadlines.at("Mathematics") = Date(9, 1, 2019);
    cout << deadlines.at("Mathematics").ToString() << endl; // prints "09 Jan 2019"
}
    catch (out_of_range &e) {
    cout << "Error" << endl;
}
```

Maps (6)

To remove elements use method *erase*:

```
void map_name.erase(iterator);
```

or

```
void map_name.erase(iterator_to_first_to_erase, iterator_to_first_not_to erase);
```

or

```
int map_name.erase(key); // returns 1 in case of success or 0 if the key was unknown
```

Examples:

```
deadlines.erase("Software security");
auto ret = deadlines.find("Programming");
if (ret != deadlines.end())
    deadlines.erase(ret);
```

```
void map_name.clear();
```

removes all the elements.

Maps (7)

Examples:

```
map<string, Date> *pDeadlines = new map<string, Date> {
 { "Mathematics", Date(5, 1, 2019) },
 { "Chemistry", Date(10, 1, 2019) },
 { "Physics", Date(15, 1, 2019) }
};
……..
delete pDeadlines; // destructors for all the Date-s called automatically


map<string, Date> *pDeadlines = new map<string, Date> {
 { "Mathematics", Date(5, 1, 2019) },
 { "Chemistry", Date(10, 1, 2019) },
 { "Physics", Date(15, 1, 2019) }
};
……..
pDeadlines->erase("Chemistry"); // destructor for the corresponding Date called automatically
……..
pDeadlines->clear(); // destructors for all the Date-s called automatically
……..
delete pDeadlines;
```

Maps (8)

Example:

```
map<string, Date *> *pDeadlines = new map<string, Date *> {
 { "Mathematics", new Date(5, 1, 2019) },
 { "Chemistry", new Date(10, 1, 2019) },
 { "Physics", new Date(15, 1, 2019) }
};
……….
delete pDeadlines->at("Chemistry"); // we must ourselves release the memory before erasing
// alternative: delete (*pDeadlines)["Chemistry"];
pDeadlines->erase("Chemistry");
……….
for (auto it = pDeadlines->begin(); it != pDeadlines->end(); ++it)
    delete it->second; // we must ourselves release the memory before clear or complete destroy
```

Alternative solution:

```
for (auto &it : *pDeadlines)
    delete it.second;
```

After that:

```
pDeadlines->clear();
delete pDeadlines;
```

Maps (9)

Example:

```
map<string, list<Date *> *> *pDeadlines = new map <string, list<Date *> *>;
/* pDeadlines is a pointer to map in which the key is a string and the value is a pointer to list.
   The list itself contains pointers to objects of class Date. */
list<Date *> *pList1 = new list<Date *> { new Date(5, 1, 2019), new Date(10, 1, 2019) };
list<Date *> *pList2 = new list<Date *> { new Date(15, 1, 2019), new Date(20, 1, 2019) };
pDeadlines->insert( { "Mathematics", pList1 } );
pDeadlines->insert( { "Physics", pList2 } );
……..
for (auto it1 = pDeadlines->begin(); it1 != pDeadlines->end(); it1++) {
    for (auto it2 = it1->second->begin(); it2 != it1->second->end(); it2++) {
        cout << it1->first.c_str() << ' ' << (*it2)->ToString() << endl;
    }
}
```

Alternative solution:

```
for (auto &it1 : *pDeadlines) {
   for (auto &it2 : *it1.second) {
       cout << it1.first.c_str() << ' ' << it2->ToString() << endl;
   }
}
```

Maps (10)

Example continues:

```
list<Date *> *pMathList = pDeadlines->at("Mathematics");
pMathList->push_front(new Date(20, 12, 2018));
cout << "The last exam in mathematics is on" << (*pMathList->rbegin())->ToString() << endl;
……..
for (auto it1 = pDeadlines->begin(); it1 != pDeadlines->end(); it1++) {
    for (auto it2 = it1->second->begin(); it2 != it1->second->end(); it2++) {
        delete *it2;
    }
 delete it1->second;
}
```

Alternative solution:

```
for (auto  it1 : *pDeadlines) {
    for (auto  &it2 : *it1.second) {
        delete it2;
    }
    delete it1.second;
}
```

Now we can destroy the map:

```
delete pDeadlines;
```

Maps (11)

To get iterators to the beginning and end of a range use methods *lower_bound* and *upper_bound*:

auto map_name.lower_bound(key);

returns iterator to the first element whose key is not considered to go before the argument.

auto map_name.upper_bound(key);

returns iterator to the first element whose key is considered to go after argument. If the searching fails, the result in both cases is iterator *map_name.end()*. Example:

```
map<string, int> students = { { "John", 5 }, { "Mary", 4 }, { "Elizabeth", 5 }, { "James", 1 },
{ "Walter", 2 } };
auto it1 = students.lower_bound(string("I"));
cout << (it1->first).c_str() << endl; // get James
auto it2 = students.upper_bound(string("N"));
cout << (it2->first).c_str() << endl; // get Walter
```

The range can be used for erasing like

```
students.erase(it1, it2);
for (auto& x : students)
    cout << x.first.c_str() << endl; // get Elizabeth Walter
```

or for constructing another map:

```
map<string, int> students1(it1, it2);
for (auto& x : students1)
   cout << x.first.c_str() << endl; // get James John Mary
```

Maps (12)

Maps support also:

- *copy constructor, operator=*
- *size, empty*
- *cbegin, rbegin, crbegin*
- *cend, rend, crend*
- *emplace*

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 www.cplusplus.com/reference/map/multimap/
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 www.cplusplus.com/reference/set/set/
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 www.cplusplus.com/reference/set/multiset/
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 www.cplusplus.com/reference/unordered_map/unordered_map/
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 www.cplusplus.com/reference/unordered_map/unordered_multimap/
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 www.cplusplus.com/reference/unordered_set/unordered_set/
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 www.cplusplus.com/reference/unordered_set/unordered_multiset/
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.

Non-member iterator functions

To operate with iterators, all the containers have member function like *begin()* and *end()*. C++ provides also iterator functions that do not belong to a class. So, instead of

```
vector<int> v = { 1, 2, 3 };
for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
      cout << *it << endl;
}
```

we may write

```
for (vector<int>::iterator it = begin(v); it != end(v); it++) {
      cout << *it << endl;
}
```

or, using advanced initializing:

```
for (vector<int>::iterator it = { begin(v) }; it != end(v); it++) {
       cout << *it << endl;
}
```

The non-member functions returning iterators are: *begin()*, *end()*, *cbegin()*, *cend()*, *rbegin()*, *rend()*, *crbegin()*, *crend()*. Their argument is the container name. They work also on C-style arrays and initializer_lists, for example:

```
int a[] = { 1, 2, 3 };
for (auto it = begin(a); it != end(a); it++) {
      cout << *it << endl;
}
```

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://www.cplusplus.com/reference/bitset/bitset/
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
```