Templates (1)

Let us have:

```
class Array
{
 protected: int Size,  * pArray;
 public:     Array(int n) {  Size = n;   pArray = new int[n]; }
             virtual ~Array()  { delete  pArray; }
             int GetSize() { return  Size; }
             int Get(int);
             void Set(int, int);
};

int Array::Get(int i)
{
  if (i < 0 || i >  Size - 1)  throw "Illegal index";
  else return *( pArray + i);
}
void Array::Set(int Value, int i)
{
  if (i < 0 || i >  Size - 1) throw  "Illegal index";
  else *( pArray + i) = Value;
}
```

Templates (2)

Generic programming: how to write class *Array* so that one of the users could apply it as a container of double numbers, another user for storing of pointers to strings, etc.

The class template defines a class where the types of some attributes, return values of methods and/or parameters of methods are specified as parameters.

```
template<typename T> class Array // deprecated: template<class T>
{ // template<typename T> is the template specifier.
  // Word "Array" here is the class template name (not the class name) .
  // T is the placeholder for actual types like int, double, etc.
protected: int Size;
           T *pArray;
public:    Array(int n) { Size = n; pArray = new T[n]; }
           virtual ~Array()  { delete pArray; }
           int GetSize() { return Size; }
           T Get(int);
           void Set(T, int);
};
```

Templates (3)

```
template<typename T> T Array<T>::Get(int i)
{// template<typename T> is the template specifier, it says that we have a template,
  // not a traditional class
  // Array<T> refers to class template with parameter T and name Array
  // Name Array without following to it <T> is meaningless
  // Array<T>::Get(int i) means that Get() is a member function of class template
  // T is the type of Get() return value.
  if (i < 0 || i > Size - 1)  throw "Illegal index";
  else return *(pArray + i);
}

template<typename T> void Array<T>::Set(T Value, int i)
{
  if (i < 0 || i > m_Size - 1) throw "Illegal index";
  else *(pArray + i) = Value;
}
```

Here *T* may be a simple variable or a class. In the last case the assignment operator overloading must be implemented.

Templates (4)

```
int main( )
{
  Array<int> IntArr(100); // instantiate the template
  try
  {
     for (int i = 0;  i < 100;  i++)
          IntArr.Set(i, i); // use any an ordinary object
      cout << IntArr.Get(5) << endl;
  }
  catch(char *pMsg)
  {
       cout << pMsg << endl;
  }
  return 0;
}
```

Important: the compiler checks the template code syntax, but does not compile it. The compiling is performed when the actual type is specified. Therefore, in the example above the compiler needs the complete code of template Array<T>.

Templates (5)

```
template<typename T> class Array
{
 ……..
  Array<T>(const Array<T> &Original)
  { // copy constructor
    Size = Original. Size;
     pArray = new T[ Size];
     memcpy(pArray, Original. pArray, sizeof(T) *  Size);
  }
  Array<T> &operator=(const Array<T> &Right)
  { // overloading assignment
     Size = Right. Size;
     delete  pArray;
      pArray = new T[Size];
     memcpy( pArray, Right. pArray, sizeof(T) *  Size);
     return *this;
  }
  …….
};
```

Remember: instead of class name *Array* here we write class template name as *Array<T>*.

Templates (6)

```
template<typename T, int SIZE> class Array
{ // non-type parameters can only be integrals (char, int, etc.), pointers and references
protected: T *pArray;
public:     Array() {  pArray = new T[SIZE]; } // constructor
            Array<T, SIZE>(const Array<T, SIZE> &Original) // copy constructor
            {  pArray = new T[SIZE];
              memcpy( pArray, Original. pArray, sizeof(T) * SIZE); }
            virtual ~Array()  { delete  pArray; } // destructor
            Array<T, SIZE> &operator=(const Array<T, SIZE> &Right) // overloading =
            { memcpy( pArray, Right. pArray, sizeof(T) * SIZE);
               return *this; }
            int GetSize() { return SIZE; } // get the number of elements
            T Get(int i) // get an element
            { if (i < 0 || i > SIZE) throw "Illegal index";
               else return *(m_pArray + i); }
            void Set(T Value, int i) // set value to an element
            { if (i < 0 || i > SIZE) throw "Illegal index";
              else *(m_pArray + i) = Value; }
};
// Array<int, 10> IntArr; // array of integers, the length is 10
```

Templates (7)

C++ supports also templates for functions:

```
template<typename T> T Larger(T a, T b)
{
        return a > b ? a : b;
}
```

Usage:

```
double x, y, z;
z = Larger<double>(x,y);
```

This function is applicable for types for which the "greater than" operation is defined.

The arguments and return values may be from different types:

```
template <typename T1, typename T2, typename T3> void Fun(T1 a, T2 b, T3 c)
{
…….
}
```

Usage:

```
double x, y;
int i;
Fun<double, int, double>(x, i, y);
```

New variable type

In C and C++ prior to version 11 keyword *auto* meant that the variable has automatic duration (i.e. it will be created and destroyed automatically):

```
auto int i; // "auto" was almost always omitted
```

In C++ v11 and later keyword *auto* means that the compiler has to deduct the actual type:

```
auto i = 10; // i is of type int
auto j = 10L; // j is of type long int
auto k; // error -  compiler is unable to deduct the type
```

auto simplifies the work of code writers. In templates its usage may be inevitable.

Example:

```
template <typename T1, typename T2> void Fun(T1 a, T2 b)
{
      auto c = a + b;
      ……..
}
```

If *T1* and *T2* are both *int*, *c* is also *int*. But if *T1* is *double* and *T2* is *int*, *c* is *double*. Consequently, when writing the code, we do not know the type of *c* and therefore using the auto deduction is the only way out.

Initializing (1)

There are several cases when the constructors written in traditional mode do not work. Examples:

```
class Test1
{
 public:
    const int ciValue = 0;
    Test2 test2; // class  Test2 has no constructor without arguments
    int &ri; // error, it is not possible to declare a reference without initialization
    Test1(int i)
    {
        ciValue = i;  // error, it is not possible to change a constant
        test2.SetInitialValues(); // error, object test2 was not created
    }
};
```

Comment: the constructor of *Test1* must at first create all the attributes and after that execute the initialization defined in its body. But to create attribute *test2* it needs to call the constructor of *Test2*. However, *Test2* has no constructor without arguments.

Initializing (2)

The constructor initializer is defined as:

class_name::class_name(list_of_arguments) : attribute_initializer_list { body }

where the comma-separated components of attribute initializers list are:

- if the attribute is not an object: attribute_name(attribute_initial_value)
- if the attribute is an object: attribute_name(constructor_arguments)

Attribute initial values and constructor arguments may be constants, elements from the constructor argument list or any other executable expressions.

Examples:

```
class Point
{
public: int x, y;
        Point(int i, int j) : x(i), y(j) { } // x gets value of i, y gets value of j, body is empty
};
class Rectangle
{
public: Point p1, p2;
        Rectangle(int x1, int y1, int x2, int  y2) : p1(x1, y1), p2(x2, y2) { }
}; // attribute initializer list contains calls to constructors of attribute objects
   // remark that class Point does not have constructor without arguments
```

Initializing (3)

Constructor initializer is necessary when:

- Some attributes are objects of classes without default (i.e. not having arguments) constructor (like *Point* on previous slides).
- Some attributes are objects of classes having constructor with arguments (already discussed earlier, see the problems with aggregation).
- A constant attribute or a reference attribute must be initialized.

```
class Test1
{
public: const int ciValue;
        int &ri;
        Test1(int i, int &j) : ciValue(i), ri(j) {  }
};
```

The classical constructor first creates all the attributes and after that executes the initializations defined in its body. The constructor initializer creates an attribute and right after that initializes it.

Mixed constructors in which some of the initializations are specified in the attribute initializers list and the others in the constructor body are also allowed.

Initializing (4)

```
class Circle
{
public:  const double pi = 3.14159;
         Point centre;
         int radius;
         double area;
         Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
```

Important: the attributes are initialized in the order that they appear in class definition. So, although in the list attribute *radius* is the first, attribute *centre* is initialized before it.

```
class Circle
{
public:  const double pi = 3.14159;
         double area; // error, when area is initialized, radius has no value
         Point centre;
         int radius;
Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
```

Initializing (5)

Let us have

```
class Circle {
public:  const double pi = 3.14159;
         Point centre;
         int radius;
         double area;
         Circle(int x, int y, int r) : radius(r), centre(x, y), area (pi * radius * radius) { }
};
struct Date {
  int Day,
      Month,
      Year;
};
```

Traditionally we define an objects of class *Circle* and *Date* like:

```
Circle c1(0, 0, 10);
Circle *pc = new Circle(0, 0, 10);
Date d1; // compiler-created default empty constructor is applied
Date *pd1 = new Date;
Date d2(); // not an error but for compiler it is the prototype of a function without
           // parameters returning object of class Date
```

Initializing (6)

It is less known that we can write also:

```
Circle c2 = Circle(0, 0, 10);
Date d2 = Date(); // but Date d3 = Date; is an error
Date *pd2 = new Date();
```

From introductory courses we know that

```
int m1[5] = { 0, 1, 2, 3, 4 };
int m2[] = { 0, 1, 2, 3, 4 }; // dimension omitted
int *pm1 = new int[5] { 0, 1, 2, 3, 4 };
int *pm2 = new int[] { 0, 1, 2, 3, 4 };
```

Actually, this is the uniform initialization that has two formats:

```
type object { initial_values_or_constructor_arguments }
type object =  { initial_values_or_constructor_arguments }
```

So:

```
int m3[5] { 0, 1, 2, 3, 4 };
int m4[] { 0, 1, 2, 3, 4 };
Circle c3 = { 0, 0, 10 }; // constructor is called
Circle c4 { 0, 0, 10 }; // constructor is called
Circle *pc5 = new Circle { 0, 0, 10 };
Date d4 = { };
Date d5 { };
int i1 = { 10 }, i2 { 10 }; // int i1 = 10, i2 = 10;
```

C++ containers: usage requirements

You can freely use C++ containers for primitive types (*int*, *double*, etc.) and C++ standard classes.

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)

(moving is out of scope of this course)

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 };
```

or

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

Examples:

```
vector<int> vec1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector<int> vec2 { 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 };
vec1.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

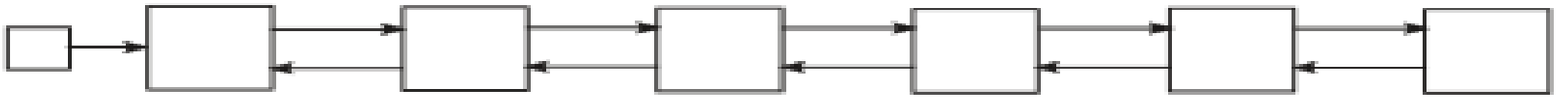
```
 for (int i = 0;  i < 6;  i++)
    delete pWeek2->at(i);
```

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 (will be discussed in the next chapter). 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. (will be discussed in the next chapter). 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.

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:

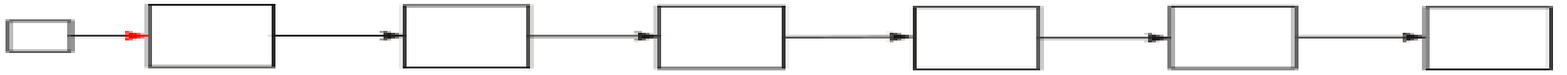
```
vector<int> vec = { 1, 2, 3, 4, 5 };
for (int i : vec)
   i++; // formally correct
for (int i : vec)
   cout << i << " "; // still 1, 2, 3, 4, 5
for (int &i : vec)
   i++;
for (int i : vec)
   cout << i << " "; // now 2, 3, 4, 5, 6
```

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 << ' ';
```

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;
}
```

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*