# STL – Standardt Template Library

- STL - Standard Template Library - provides a tremendous amount of power to the C ++ language. This library is widely used when developing C ++ applications. Therefore, in this course, we will cover the most important aspects of using the STL.

- The basic principle inherent in C ++ is that the data of a class is placed at the top level of the hierarchy, and the functions that transform this data are subject to them.

- In the STL, the opposite is true. At the top level of the hierarchy are algorithms that work the same for any data type.

- The combination of these two opposing concepts gives rise to certain difficulties in understanding. Therefore, it is very important to understand this.

- STL contains general-purpose class templates and implementing functions.

- The STL kernel consists of *containers*, *algorithms* and *iterators*.

- *Containers* - these are objects intended for the storage of other objects. Containers are possible to store the data of any types. There are several types of containers:

  - *vector* - contains dynamic array definitions (basic container).
  - *deque* - queues and stacks (basic container).
  - list - lists (basic container).
  - map - associative container - enables efficient retrieval of values based on keys (pair - key / value).

- Each container class contains definitions of the functions operating on it (push an element at the end, pop an element, insert an element at the given place in a container, etc.).

- *Algorithms* - they operate on containers. There are algorithms for initializing, sorting, finding and replacing elements of a container class, etc. Many algorithms operate on ranges of elements within a *container*.

- *Iterator*s - are objects that act as pointers in relation to objects. They allow you to navigate through the contents of a container in a similar way to the pointers navigate within an array. There are five types of iterators:

| Iterator | Description |
|---|---|
| **Random Access Iterator** | **Writing and reading of values. Allows access to elements of container in a random (in any) order.** |
| **Bidirectional Iterator** | **Writing and reading of values. It allows you to move forward and backward in a sequential manner (incrementing or decrementing).** |
| **Forward Iterator** | **Writing and reading of values. It only allows you to move forward in a sequential manner.** |
| **Input Iterator** | **Only reading the value. Moving only forward.** |
| **Output Iterator** | **Only write value. Moving only forward.** |

- Attention! Similarly with I / O streams:
  - Input - this is data retrieval from the container (read).
  - Output - this is data output into the container (write).

- A more powerful iterator may be used in place of a more restricted iterator. For example, the forward iterator can replace the input iterator.

- *Iterators* are used as well as pointers. It is possible to increment/decrement them, the indirection operator * (*iter_val*) retrieves an object using an iterator *iter_val* that points to this object. *Iterators* are declared using an iterator type defined in different containers:

*std::vector*<type> :: *iterator* my_iter;

- STL also supports *reverse* iterators. These are bidirectional iterators or random access iterators that move in the opposite direction through the sequence. This means that incrementing the inverse iterator pointing to the last element of the sequence will move to the penultimate element.

**Example W44.**

- In addition to containers, algorithms, and iterators, STL includes a number of other standard components. The most important of these are *allocators*, *predicates*, and *comparator functions*.

- *Allocators.* For each container, an allocator is defined that manages the memory allocated to the container. A default allocator is an object of the class allocator, but the developer can define his own object if a specialized application requires it. In most cases, the default allocator is fine.

- Some algorithms and containers use special functions called *predicates*. There are unary and binary predicates. These functions return true or false.

- Some algorithms and classes use special binary *predicates* to compare two elements.

- Typical STL headers: <*vector*>, <*deque*>, <*algorithm*>, <*functional*>, <*utility*> - see MSDN.

- STL container class name types:

| | |
|---|---|
| size_type | size_t |
| reference | Reference to the element of container |
| const_reference | Element reference declared as const |
| iterator | Iterator |
| const_iterator | Iterator declared as const |
| reverse_iterator | Reverse iterator |
| const_reverse_iterator | Reverse iterator declared as const |
| value_type | The type of value stored in the container |
| allocator_type | The type of allocator |
| key_type | The type of key |
| key_compare | Type of function that compares two keys |
| value_compare | Type of function that compares two values |

## ➢ Vector

➢ Supports dynamic arrays - can be enlarged as needed.

➢ Template specification for *vector*:

*template* <*class* T, *class* Allocator=*allocator*<T>> *class vector*

➢ T - the type of stored data, Allocator - defines the allocator, the default is the standard allocator.

➢ Constructors:

*explicit vector*(const Allocator &a=Allocator()); //creates an empty vector

//creates *vector* consisted of num elements with value val
*explicit vector*(size_type *num*, const T &*val* = T(),
const Allocator &a=Allocator());

//creates a vector with the same elements as ob
*vector*(const *vector*<T, Allocator> &*ob*);

//creates a *vector* containing elements from the range
//defined by iterators *start* and *end*
*template*<clasInIter> *vector*(InIter *start*, InIter *end*,
const Allocator &a=Allocator());

- Each object that will be stored in the vector container must have a default constructor defined, and also depending on the functionality:

- Parameterized constructor.
- Destructor (if needs to release of memory).
- Copy constructor.
- Overloaded operator =.
- Overloaded operators ==, !=, <, <=, >, >=.

- The operators **==, <, <=, !=, >, >=, [ ]** are defined for the *vector* container.

- The most frequently used member functions are:

*size*(), *begin*(), *end*(), *push_back*(), *insert*(), *erase*(), *clear*().

| | |
|---|---|
| size() | Returns the current number of items in the array |
| begin() | Returns an iterator pointing to the start of the vector |
| end() | Returns an iterator pointed to the end of the vector (the first empty item in the array) |
| push_back(…) | Places the value in the first empty position |
| insert(…) | Adding elements in the arbitrary position of the vector |
| erase(…) | Deleting items from the vector |
| clear() | Removing all elements from an array. Memory deallocation does not arise. |

# Algorithms

**<algorithm>**

- They operate on containers. Although each container supports basic operations, the algorithms allow you to perform more elaborate and complex actions. They also enable simultaneous work with two containers of different types.

- All algorithms are function templates. This means that they can be used with any type of container.

- The list of algorithms is given in the MSDN. Let us consider some of the most popular algorithms.

**Example W45**.