1. **Templates:** Definition and usage.
2. **Namespace.** Visibility extension operator. ***using directive***.
3. **Standard template library (STL)**. Algorithms. Containers. Iterators.

1. Templates

Templates let us define functions, which can work with different types of information. Templates are defined as following:

```
template <class identifier> definition of function;
template <typename identifier> definition of function;
```

Both definitions are identical. You can use both ***class*** and ***typename*** keywords.

For example, in C you can create overloaded function for calculating absolute value:

```
int abs(int n)
{
  if (n < 0) return -n;
  return n;
}

double abs(double n)
{
  if (n < 0) return -n;
  return n;
}
```

Using template, you can create unique definition which will be automatically work with any type of information:

```
#include <stdio.h>

template <class T> T abs(T n)
{
  if (n < 0) return -n;
  return n;
}

int main(void)
{
  double d = abs<double>(-4.55);
  int i = abs<int>(-7);
  long long l = abs<long long>(-7000);
  printf("%lf %d %lld\n",d,i,l);
  return 0;
}
```

Example. Find the maximum of two elements:

```
#include <stdio.h>
```

```
template <class Type> Type max (Type a, Type b)
{
  return (a > b ? a : b);
}

int main(void)
{
  double d = max<double>(-3.55,-5.678);
  int i = max<int>(-7,9);
  long long l = max<long long>(67,34);
  printf("%lf %d %lld\n",d,i,l);
  return 0;
}
```

2. Namespace

namespace

Namespace lets us to combine global classes, objects, functions under one name. In other words, they let us divide global space to fields each of which has own active objects. Namespaces are defined as following:

```
namespace <identifier>
{
  <namespace body>
}
```

Namespace body may contain many classes, objects, and functions. For example:

```
namespace ivan
{
  int a, b;
}
```

For accessing elements of namespace from outside you have to use operator of visibility :: . For example, to access variable *a* you have to write: ivan::*a*. The following program creates two namespaces each of which has own variable. Variables and functions from different namespaces may have same names.

```
#include <stdio.h>

namespace first
{
  int var = 5;
}
namespace second
{
  double var = 3.1416;
}

int main (void)
{
  printf("%d\n",first::var);
```

```
  printf("%lf\n",second::var);
  return 0;
}
```

using namespace

Directive ***using*** lets us to associate the current global space of objects with namespace. After command

```
using namespace <identifier>
```

is done, you can access all objects and functions of namespace.

```
#include <stdio.h>

namespace first
{
  int var = 5;
}
namespace second
{
  double var = 3.1416;
}

int main (void)
{
  {
    using namespace first;
    printf("%d\n",var);
  }
  {
    using namespace second;
    printf("%lf\n",var);
  }
  return 0;
}
```

In case of having two namespaces in one block there can occur some problems because of identical names.

3. Standard Template Library STL

STL (*standard template library*) is the collection of function and class templates in C++ which include different containers of information (list, queue, set, mapping, hash table, priority queue) and base algorithms(sorting, searching).

STL is a namespace called **std**. When using it, all files included in it are written without extension .h, but some of them are written with adding **c** to the beginning of name. For example, <stdio.h>, <limits.h> take place in STL as <cstdio>, <climits>.

To include STL you have to use directive:

```
using namespace std;
```

STL includes five main component types:

1. ***algorithm***: defines calculating procedure.
2. ***container***: manages list of objects in memory.
3. ***iterator***: provides for algorithm the facilities for accessing the contents of the container.
4. ***function*** object: encapsulates function in object to be used by other components.
5. ***adaptor***: adapts component for providing different interface.

Containers are frequently encountered methods of data organization: dynamic arrays, lists, queues, stacks.

Algorithms are not the part of containers, they create separate subsystem. However, almost each algorithm can be applied to the almost each container. Calling a method for some algorithm, we call this method by his own, not for the instance of some class, while the container, to which the algorithm is applied, is passed as a parameter.

Iterator is a pointer, which can move throughout elements of container. Iterators play the same role as index of array elements. By index of array you can get array element, by iterator you can get element of container.