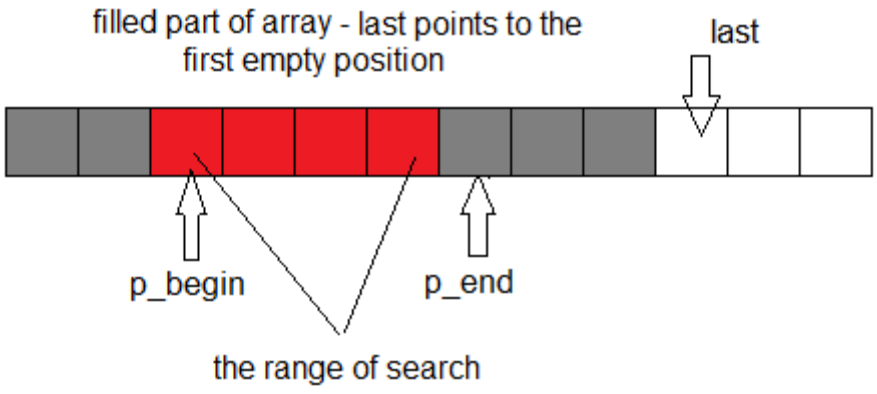
Lab 6

1. Create the template function my_add that does c = a + b, where a, b, c are generic type T. This function should work for any type of data (you must not change any line of code in the template function substituting any real data type). Therefore, every data class working with a template function should have an overload of the +, = operators as well as a copy constructor. The following code tests the my_add function for the double and NODE_COORD data types. The NODE_COORD class was developed in the implementation of Lab 4.

```
int _tmain(int argc, _TCHAR* argv[])
{

	//	type NODE_COORD
	NODE_COORD ob1(2, 3), ob2(3,4), ob;
	ob1.disp("ob1");
	ob2.disp("ob2");
	ob = my_add(ob1, ob2);
	ob.disp("ob");

	//	type int
	int a = 2, b = 3, c;
	c = my_add(a, b);

	system("pause");
	return 0;
}
```

2. Create a template function that finds the required element of an array of generic type T. The prototype of this function corresponds to the find function of STL. *p_begin* – pointer to elements of the array from which the search is started, *p_end* – pointer to the first element of the array, which is out of searching range, key – an object of type K, which is used as the key for the search. This is the object that needs to be found. It can be an object of type T (K == T), or it can be an object of another type - it depends on which criteria the search is performed on. The my_find function contains t == k statement, where t is a current element of an array of type T and k has type K - you need to overload the == operator for class T so that you can compare an element of an array of type T with a search key of type K.

The element k can appear several times in an array of type T or not even once. In the latter case, my_find must return NULL. First serach: we set *p_begin* to the beginning of the array. If the result is not NULL, the searched item is still found. We need to check if this element is not present again - repeat the search, setting *p_begin* to the result of the previous search plus 1 – see example main.



```
template <class T, class Key>
T * my_find(const T *p_begin, const T *p_end, const K &key)
/*=======================================================
Find element of array T *
Start from p_begin - points to the element to be started
          p_end   - points to the first element to right from search interval
          key - key of search
There is nesessary to overload operator == for class T according with type of
K.
1. First search: p_begin = &array[0]
2. Second search: p_begin = &array[previous search+1]
=======================================================*/

int _tmain(int argc, _TCHAR* argv[])
{

	//		type double
	int ndim = 10;
	int it;
	size_t dist;
	double *p_tab;  //declae an array of double type
	double *p_tmp, search_item;

	try
	{
		p_tab = new double[ndim];
	}
	catch(bad_alloc aa)
	{
		cout << "mem_alloc_err\n";
		system("pause");
		exit(1);
	}

	for(it=0; it<ndim-1; it++)
	{
		p_tab[it] = (double)(it+1);
	}
	search_item = p_tab[ndim-1] = 3;

	cout << "how many " << search_item << " is met in p_tab?\n";
	cout << "which elements contain search_item?\n";

	p_tmp = p_tab;    //set p_tmp to the begin of array
	while(p_tmp)
	{
		p_tmp = my_find(p_tmp, p_tab+ndim, search_item);
		if(p_tmp)
		{
			dist = p_tmp-p_tab;
			cout << "tab [ " << dist << " ] = " << *p_tmp << endl;
			p_tmp++;
		}
		else
			cout << "End of search\n";
	}
	delete [] p_tab;

	//		type node_coord
	NODE_COORD tab_coord[] = {
		NODE_COORD(0, -1),
		NODE_COORD(3.0000000000001, 0),
		NODE_COORD(2, -1),
		NODE_COORD(0, 5)
	};
```

```
        double search_key = 3;
        // Find all vertices that contain at least one coordinate equal to 3.0
        //write your code, please!
        system("pause");
        return 0;
}
```