OOP

Gavrilut Dragos

Course 7

rev 4

Summary

- Standard Template Library (STL)
- Sequence containers
- Adaptors
- I/O Streams
- Strings
- Initialization lists

Standard Template Library (STL)

STL (Standard Template Library)

- A set of templates that contains:
  - Containers (class templates that may contain other classes)
  - Iterators (pointer used for template iteration)
  - Algorithms (functions that can be called for a container)
  - Adaptors
  - Allocators
  - Others
- To use an STL template, we have to include the header(s) where that template is defined.
- STL object can be initialized in the following way: (“***vector<int> x***”) if we use “`using namespace std;`”, or using their full scope definition (“`std::vector<int> x`”) otherwise.
- **There may be various implementation of STL objects. As these implementation are compiler specific, their performance varies from on compiler to another.**

STL- Containers

- **Sequence containers**
  - vector
  - Array
  - list
  - forward_list
  - deque
  - Adaptors (stack, queue, priority_queue)
- **Associative containers**
  - Ordered: set, multiset, map, multimap
  - Un-ordered: unordered_set, unordered_map, unordered_multiset, unordered_multimap

Sequence containers

STL - Vector

- Vector = an unidimensional array for objects
- Constructors:

App.cpp

```
using namespace std;
#include <vector>

void main(void)
{
	vector<Type> v;
	vector<Type> v(Size);
}
```

- Memory allocation / resizing is done dynamically
- Every object added in a **vector** is actually a copy of the original one

STL - Vector

- Insertion in a vector is done using the following commands: **push_back**, **insert**
- For deletion, we can use: **pop_back**, **erase** or **clear**
- For reallocation: the **resize** and **reserve** methods
- For access to elements: the index operator and the "**at**" method

App.cpp

```
using namespace std;
#include <vector>

void main(void)
{
	vector<int> v;
	v.push_back(1);v.push_back(2);
	int x = v[1];
	int y = v.at(0);
}
```

STL - Vector

App.cpp

```
using namespace std;
#include <vector>

class Integer
{
	int Value;
public:
	Integer() : Value(0) {}
	Integer(int v) : Value(v) {}
	Integer(const Integer &v) : Value(v.Value) {}
	void Set(int v) { Value = v; }
	int  Get() { return Value; }
};

void main(void)
{
	vector<Integer> v;
	Integer i(5);
	v.push_back(i);
	i.Set(6);
	printf("i=%d,v[0]=%d\n", i.Get(), v[0].Get());

}
```

After execution: i=6,v[0]=5

STL - Vector

Example:

App.cpp

```
class Integer
{
	int Value;
public:
	Integer() : Value(0) { printf("[%p] Default ctor\n", this); }
	Integer(int v) : Value(v) { printf("[%p] Value ctor(%d)\n", this,v); }
	Integer(const Integer &v) : Value(v.Value) { printf("[%p] Copy ctor from (%p,%d)\n",
                                                    this, &v, v.Value); }
	Integer& operator= (const Integer& i) { Value = i.Value; printf("[%p] op= (%p,%d)\n",
                                               this, &i, i.Value); return *this; }
	void Set(int v) { Value = v; }
	int  Get() { return Value; }
};
void main(void)
{
	vector<Integer> v;
	Integer i(5);
	for (int tr = 0; tr < 5; tr++)
	{
		i.Set(1000 + tr);
		v.push_back(i);
	}
}
```

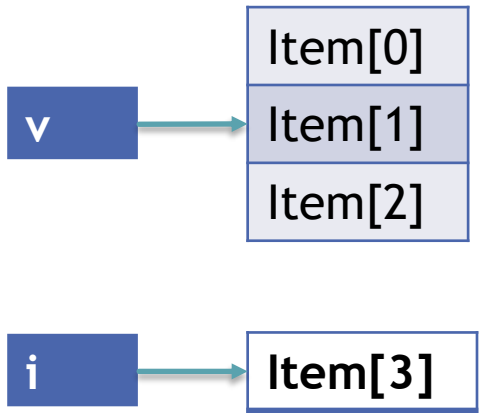
STL - Vector

Example:

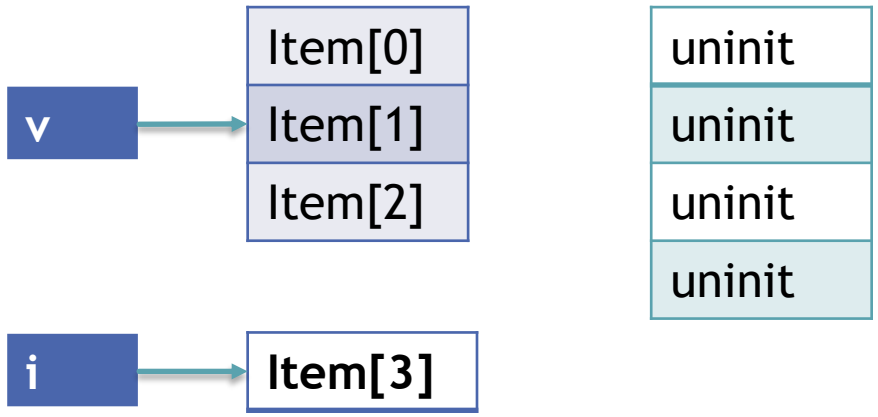
Pas	Output
-	[0098FC90] Value ctor(5)
tr=0	[00D3A688] Copy ctor from (0098FC90,1000)
tr=1	[00D3A6C8] Copy ctor from (00D3A688,1000) [00D3A6CC] Copy ctor from (0098FC90,1001)
tr=2	[00D3A710] Copy ctor from (00D3A6C8,1000) [00D3A714] Copy ctor from (00D3A6CC,1001) [00D3A718] Copy ctor from (0098FC90,1002)
tr=3	[00D3A688] Copy ctor from (00D3A710,1000) [00D3A68C] Copy ctor from (00D3A714,1001) [00D3A690] Copy ctor from (00D3A718,1002) [00D3A694] Copy ctor from (0098FC90,1003)
tr=4	[00D3A6D8] Copy ctor from (00D3A688,1000) … [00D3A6E8] Copy ctor from (0098FC90,1004)

STL - Vector

v.push_back(i)



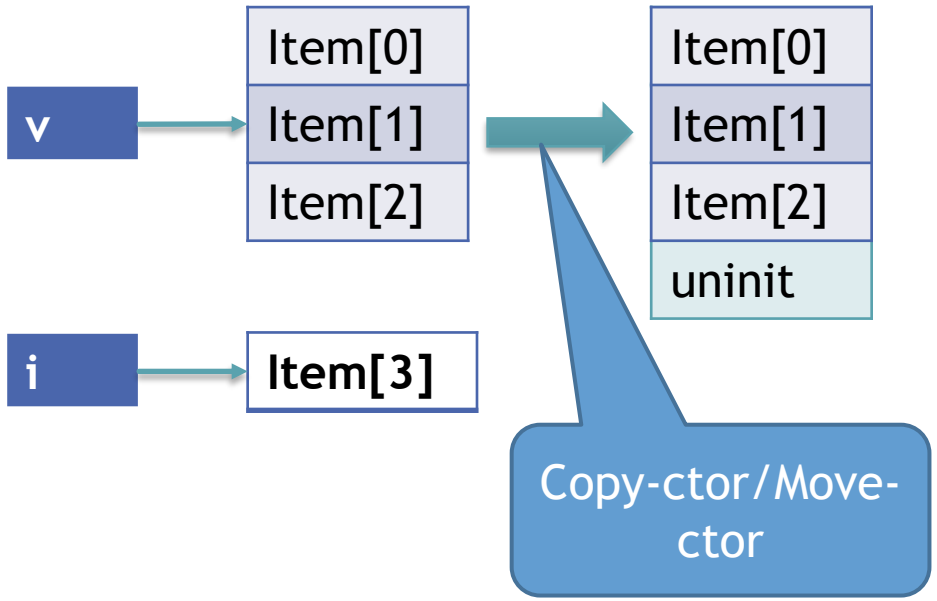
STL - Vector



v.push_back(i)

- Allocate space for count+1 elements

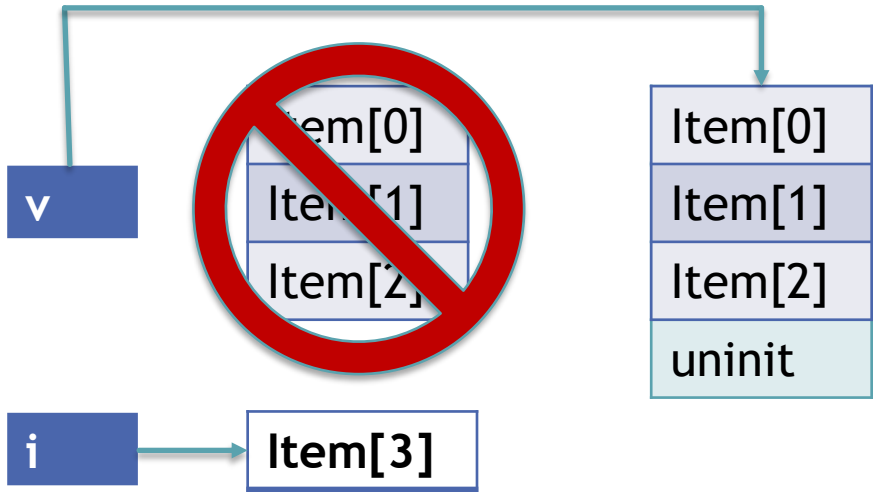
STL - Vector



v.push_back(i)

- Allocate space for count+1 elements
- Copy/Move the first ***count*** elements in the new allocated space

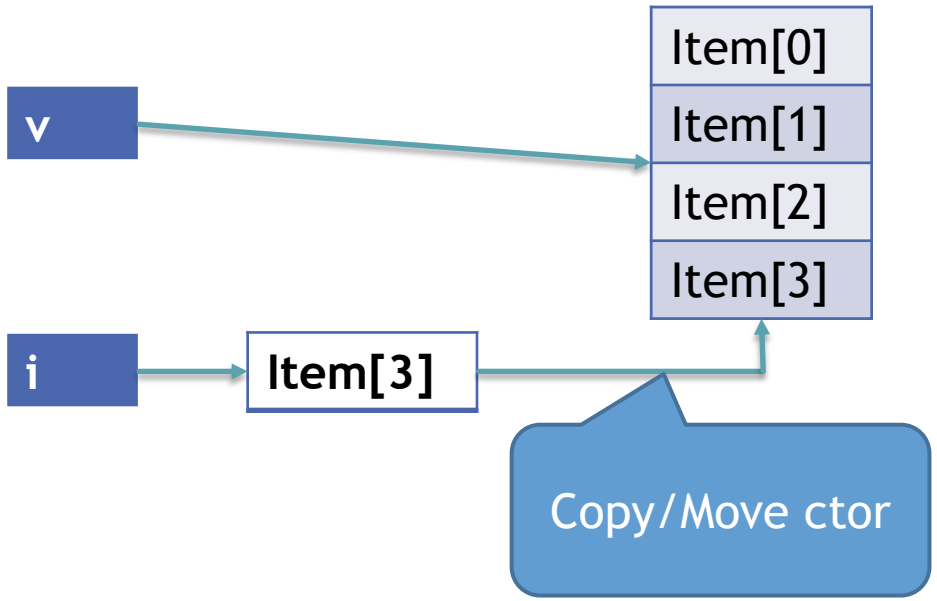
STL - Vector



v.push_back(i)

- Allocate space for count+1 elements
- Copy/Move the first ***count*** elements in the new allocated space
- Free the space used by the current elements

STL - Vector



v.push_back(i)

- Allocate space for count+1 elements
- Copy/Move the first ***count*** elements in the new allocated space
- Free the space used by the current elements
- Copy/Move the new item into the extra allocated space

STL - Vector

Test case:

App-1.cpp

```
#define INTEGER_SIZE 1000
class Integer
{
	int Values[INTEGER_SIZE];
public:
	Integer() { Set(0); }
	Integer(int value) { Set(value); }
	Integer(const Integer &v) { CopyFrom((int*)v.Values); }
	Integer& operator= (const Integer& i) { CopyFrom((int*)i.Values); return *this; }

	void Set(int value)
	{
		for (int tr = 0; tr < INTEGER_SIZE; tr++)
			Values[tr] = value;
	}
	void CopyFrom(int* lista)
	{
		for (int tr = 0; tr < INTEGER_SIZE; tr++)
			Values[tr] = lista[tr];
	}
	void Set(const Integer &v)
	{
		CopyFrom((int*)v.Values);
	}
};
```

STL - Vector

Test case:

Vec-1.cpp

```
void main(void)
{
	vector<Integer> v;
	Integer i;
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v.push_back(i);
	}
}
```

Vec-2.cpp

```
void main(void)
{
	vector<Integer> v;
	Integer i;
	v.reserve(100000);
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v.push_back(i);
	}
}
```

Ptr-1.cpp

```
void main(void)
{
	Integer *v = new Integer[100000];
	Integer i;
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v[tr].Set(i);
	}
}
```

Ptr-2.cpp

```
void main(void)
{
	Integer **v = new Integer*[100000];
	for (int tr = 0; tr < 100000; tr++)
	{
		v[tr] = new Integer(tr);
	}
}
```

STL - Vector

The study was conducted in the following way:

- Each of the 4 applications was executed for 10 time. Execution time was measured in milliseconds.
- Execution time was divided into two times: time needed to initialize the container and the time needed to add the data into the container.

App-1.cpp

```
void main(void)
{
        vector<Integer> v;
        Integer i;

        for (int tr = 0; tr < 100000; tr++)
        {
                ...
        }
}
```

Initialization time frame

Add data to container time frame

- All of these tests were conducted using the following specifications:
  - OS: Windows 8.1 Pro
  - Compiler: cl.exe [18.00.21005.1 for x86]
  - Hardware: Dell Latitude 7440 – i7 – 4600U, 2.70 GHz, 8 GB RAM

STL - Vector

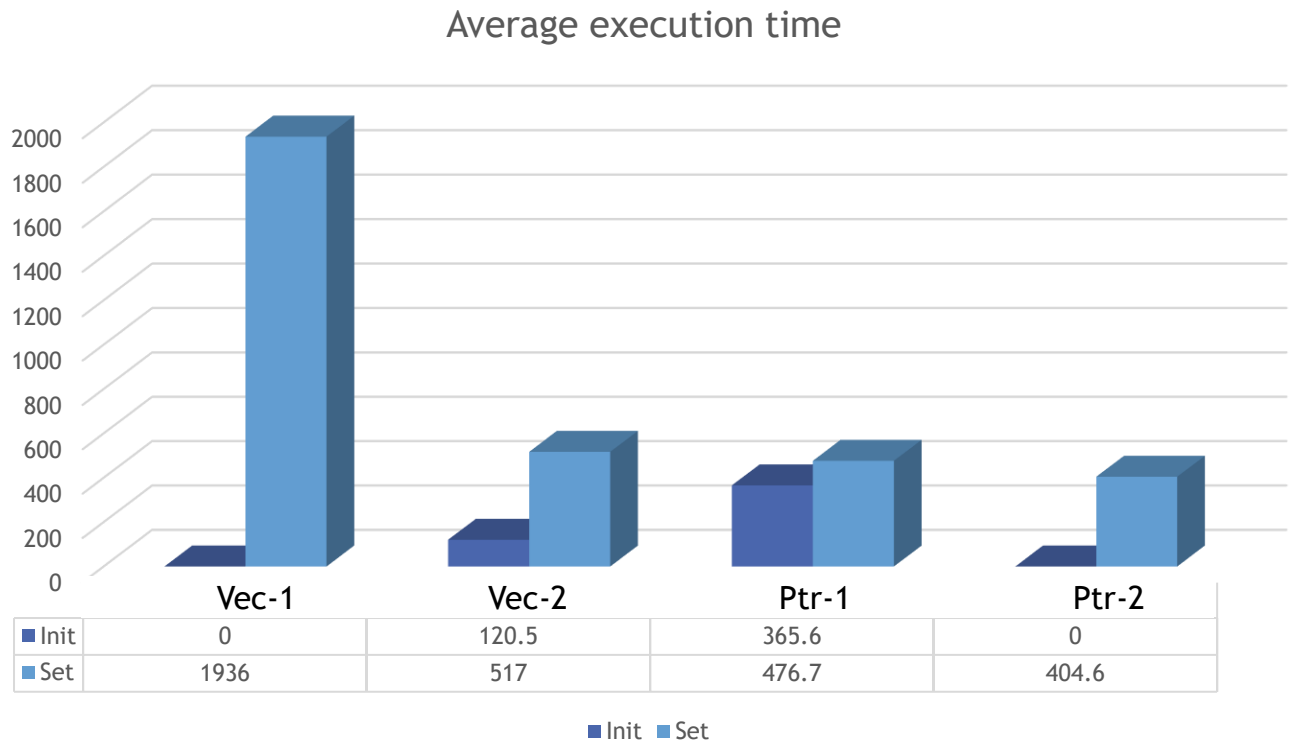
Results:

Alg	Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Vec-1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
Vec-2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
Ptr-1	Init	406	359	360	375	359	360	359	359	359	360
	Set	485	485	468	469	485	468	469	485	469	484
Ptr-2	Init	0	0	0	0	0	0	0	0	0	0
	Set	422	453	453	437	375	391	390	375	375	375

- Vec-1,Vec-2,Ptr-1,Ptr-2 ➔ the 4 methods previously described
- “Init” ➔ the series of times recorded when we initialize the container
- “Set” ➔ the series of times needed to add the data in in container
- T1 .. T10 ➔ result times

STL - Vector

Results:



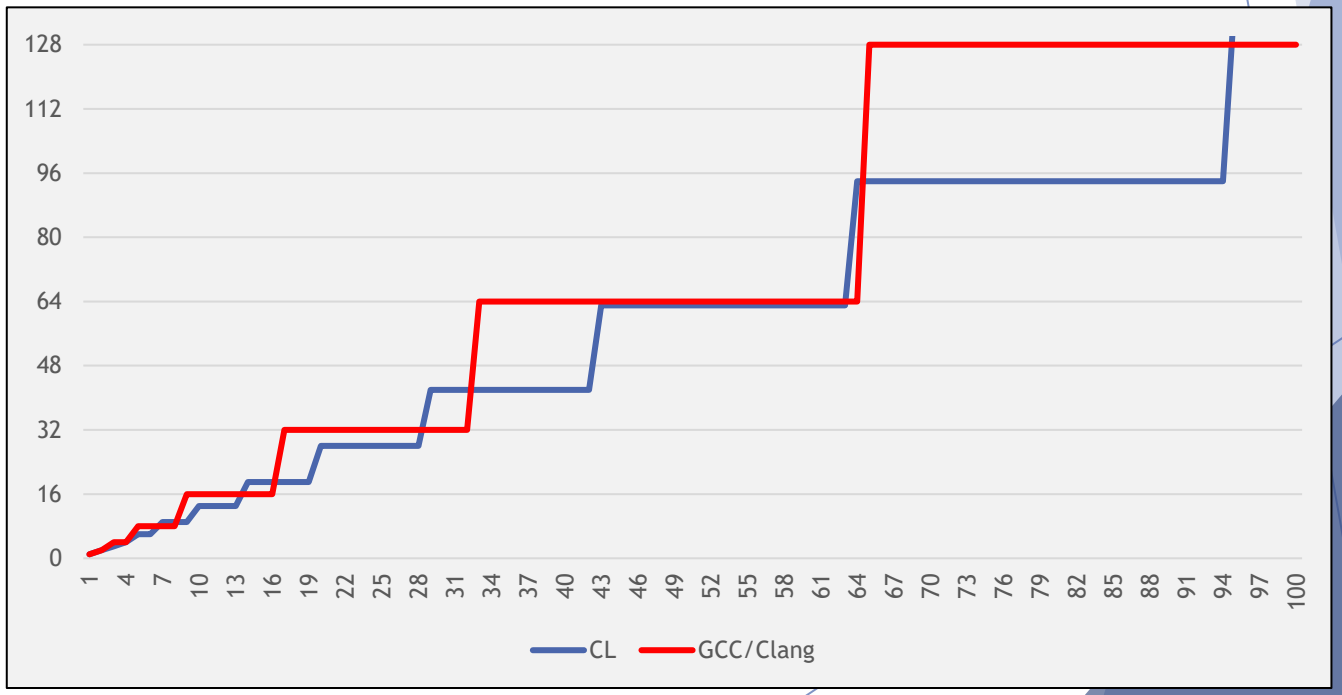
- Vec-1,Vec-2,Ptr-1,Ptr-2 ➔ the 4 methods previously described
- “Init” ➔ the series of times recorded when we initialize the container
- “Set” ➔ the series of times needed to add the data in in container

STL - Vector

- Let's consider the following code:
- If we run this code on g++, clang and cl.exe (Microsoft)
- The results show that the allocation strategy is different depending on the compiler.
- Clang/G++ prefers powers of 2.

App-1.cpp

```
void main(void) {
    vector<int> v;
    for (int tr = 0; tr < 1000; tr++) {
        v.push_back(0);
        printf("Elem: %4d, Allocated: %4d\n", v.size(), v.capacity());
    }
}
```



STL - Vector

- To iterate through all of the elements stored in a ***vector*** we can use ***iterators***. An ***iterator*** is a special object (similar to a pointer)

App.cpp

```
void main(void)
{
	vector<int> v;
	v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4); v.push_back(5);
	vector<int>::iterator it;
	it = v.begin();
	while (it < v.end())
	{
		printf("%d ", (int)(*it));
		it++;
	}
}
```

- Iterators work by overloading the following operators:
  - operator++ , operator-- (to go forward/backwards within the container)
  - Pointer related operators (operator->) to gain access to an element from the container

STL - Vector

- In this example access to elements is done via **operator→**
- ***vector*** template also provides two methods: ***front*** and ***back*** that allows access to the first and last elements stored.

App.cpp

```
class Number
{
public:
	int Value;
	Number() : Value(0) {}
	Number(int val) : Value(val) {}
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));

	vector<Number>::iterator i = v.begin();
	printf("%d\n", i->Value);
	printf("%d\n", (i + 2)->Value);
	printf("%d %d", v.front().Value, v.back().Value);
}
```

- This example will print on the screen: 1 3 1 5

STL - Vector

- ***vector*** template also provides a special iterator (called ***reverse_iterator***) that allows iterating/moving through the elements of the vector in the reverse order.

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));

	vector<Number>::iterator it;
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	vector<Number>::reverse_iterator rit;
	for (rit = v.rbegin(); rit != v.rend(); rit++)
		printf("%d ", rit->Value);
}
```

- This example will print: "***1 2 3 4 5 5 4 3 2 1***"

STL - Vector

- Inserting elements in a vector:

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));
	vector<Number>::iterator i = v.begin();

	v.insert(i + 2, Number(10));

	while (i < v.end())
	{
		printf("%d,", i->Value);
		i++;
	}
}
```

- This example will print: 1,2,10,3,4,5

STL - Vector

- Deleting an element from the vector: (***v.erase(iterator)***)

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));
	vector<Number>::iterator i = v.begin();
	v.erase(i+2);
	v.erase(i+3);
	while (i < v.end())
	{
		printf("%d,", i->Value);
		i++;
	}
}
```

- This example will print: 1,2,4

STL - Vector

- This code compiles. However, keep in mind that after an element is deleted, an iterator can be invalid. As a general rule, don't ***REUSE*** an interator after it was deleted like in the next example: ***"v.erase(i+1)"***

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));
	vector<Number>::iterator i = v.begin();
	v.erase(i);
	v.erase(i+1);
	while (i < v.end())
	{
		printf("%d,", i->Value);
		i++;
	}
}
```

- During the execution, an exception will be triggered on ***"v.erase(i+1)"***

STL - Vector

- To fix the previous problem, we don't re-use an iterator when we delete an element. Instead, we compute an iterator in place (using ***v.begin()*** to do this).

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v;
	v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
	v.push_back(Number(4)); v.push_back(Number(5));
	v.erase(v.begin());
	v.erase(v.begin()+1);
	vector<Number>::iterator i = v.begin();
	while (i < v.end())
	{
		printf("%d,", i->Value);
		i++;
	}
}
```

- This code compiles and works correctly.

STL - Vector

- Other operators that are overwritten are relationship operators (>, < , !=, …)
- Two vectors are considered to be equals if they have the same number of elements and elements with the same index in those two vectors are equal.

App.cpp

```
class Number
{
...
};
void main(void)
{
	vector<Number> v1;
	vector<Number> v2;
	for (int tr = 0; tr < 10; tr++)
	{
		v1.push_back(Number(tr));
		v2.push_back(Number(tr));
	}
	if (v1 == v2)
		printf("V1 is equal with V2");
	else
		printf("V2 is different than V2");
}
```

- This example will not compile as Number does not have an ***operator==*** defined !

STL - Vector

- Make sure that you define ***operator***== as ***const***. ***vector*** template requires a const ***operator***== to work.
- This code will not compile.

App.cpp

```
class Number
{
    bool operator==(const Number &n1) { return Value == n1.Value;  }
};
void main(void)
{
      vector<Number> v1;
      vector<Number> v2;
      for (int tr = 0; tr < 10; tr++)
      {
            v1.push_back(Number(tr));
            v2.push_back(Number(tr));
      }
      if (v1 == v2)
            printf("V1 is equal with V2");
      else
            printf("V2 is different than V2");
}
```

STL - Vector

- Now this code compiles and work as expected.
- You can also replace ***operator***== with a friend function with the following syntax: "`bool friend operator== (const Number & n1, const Number & n2);`"

App.cpp

```
class Number
{
    bool operator==(const Number &n1) const { return Value == n1.Value;  }
};
void main(void)
{
      vector<Number> v1;
      vector<Number> v2;
      for (int tr = 0; tr < 10; tr++)
      {
            v1.push_back(Number(tr));
            v2.push_back(Number(tr));
      }
      if (v1 == v2)
            printf("V1 is equal with V2");
      else
            printf("V2 is different than V2");
}
```

STL - Vector

- For < and > compare operations, ***vector*** template uses the following algorithm (see the pseudo-code below) to decide if a vector is bigger/smaller than another one.

Pseudocode

```
Function IsSmaller (Vector v1, Vector v2)
      Size = MINIM(v1.Size, v2.Size)
      for (i=0; i<Size; i++)
            if (v1[i] < v2[i])
                  return "v1 is smaller than v2";
            end if
      end for
      if (v1.Size < v2.Size)
            return "v1 is smaller than v2";
      end if
      return "v1 is NOT smaller than v2";
End Function
```

App.cpp

```
void main(void)
{
    vector<int> v1;
    vector<int> v2;

    v1.push_back(10);
    v2.push_back(5); v2.push_back(20);

    if (v2 < v1)
        printf("v2<v1");
}
```

- One important observation here is that ***operator>*** is not needed (only ***operator<*** is needed to decide).
- The previous code will print “v2<v1” (even if “v2” has two elements, and “v1” only one).

STL - Vector

- Let’s analyze the following example:

App.cpp

```
class Number
{
    bool operator<(const Number &n1) const { return Value < n1.Value; }
};
void main(void)
{
      vector<Number> v1;
      vector<Number> v2;

      v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
      v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));


      if (v1 < v2)
            printf("OK");
      else
            printf("NOT-OK");
}
```

- This code compiles and upon execution prints “OK” on the screen as v1[2]=3 and v2[2]=4, and 3<4

STL - Vector

- This code will not compile as ***operator<*** is not defined.

App.cpp

```
class Number
{
    bool operator>(const Number &n1) const { return Value < n1.Value; }
};
void main(void)
{
      vector<Number> v1;
      vector<Number> v2;

      v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
      v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));


      if (v1 > v2)
            printf("OK");
      else
            printf("NOT-OK");
}
```

- ***vector*** template requires operator< to be defined in class Number. Similarly, ***operator!=*** is not needed, only ***operator==*** is.

STL - Vector

- You can also copy the values from one vector into another.

App.cpp

```
class Number
{
    ...
};
void main(void)
{
      vector<Number> v1;
      vector<Number> v2;

      v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));

      v2 = v1;

      for (int i = 0; i < v2.size(); i++)
            printf("%d ", v2[i].Value);
}
```

- This code compiles and prints 1 2 3

STL - Vector

- The following methods from template ***vector*** can be used to obtain different information:
  - size() → the amount of elements stored in the vector
  - capacity() → the pre-allocated space in the vector
  - max_size() → maximum number of elements that can be stored in the vector
  - empty() → returns "***true***" if the current vector has no elements
  - data() → provides direct access (a pointer) to all of the elements in the vector.

App.cpp

```
class Number { ... };
void main(void)
{
	vector<Number> v;
	v.push_back(Number(0)); v.push_back(Number(1));
	printf("%d ", v.max_size());
	printf("%d ", v.size());
	printf("%d ", v.capacity());
	Number *n = v.data();
	printf("[%d %d]", n[0].Value, n[1].Value);
}
```

STL - Vector

- std::vector also has an **emplace_back** method that has the same result as **push_back**, but it is faster:

App.cpp (push_back)

```
#include <vector>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y(v2),
                           z(v1+1), t(v2+1) {}
};

int main() {
    std::vector<Test> v;
    v.push_back(Test(10,20));

    return 0;
}
```

App.cpp (emplace_back)

```
#include <vector>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y(v2),
                           z(v1+1), t(v2+1) {}
};

int main() {
    std::vector<Test> v;
    v.emplace_back(10,20);

    return 0;
}
```

- push_back has one parameter of type T
- emplace_back has multiple parameters that reflect the parameters that the constructor(s) of T might receive.

STL - Vector

- std::vector also has an **emplace_back** method that has the same result as **push_back**, but it is faster:

App.cpp (push_back)

```
#include <vector>

class Test {
    int x, y, z, t;

int main() {
    std::vector<Test> v;
    v.push_back(Test(10,20));

    return 0;
}
```

First construct the object T, then copy or move it into the vector !

App.cpp (emplace_back)

```
#include <vector>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y(v2),
                           z(v1+1), t(v2+1) {}
};

int main() {
    std::vector<Test> v;
    v.emplace_back(10,20);

    return 0;
}
```

- push_back has one parameter of type T
- emplace_back has multiple parameters that reflect the parameters that the constructor(s) of T might receive.

STL - Vector

- std::vector also has an **emplace_back** method that has the same result as **push_back**, but it is faster:

App.cpp (push_back)

```
#include <vector>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y
                           z(v1+1),
};

int main() {
    std::vector<Test> v;
    v.push_back(Test(10,20));

    return 0;
}
```

App.cpp (emplace_back)

```
int main() {
    std::vector<Test> v;
    v.emplace_back(10,20);

    return 0;
}
```

Notice that (10,20) are not parameters for the method **emplace_back**. They are however parameters for the *ctor* of class Test. In reality the object is constructed directly in the vector position not copied/moved from a stack location.

- push_back has one parameter of type T
- emplace_back has multiple parameters that reflect the parameters that the constructor(s) of T might receive.

STL - Vector

- Let’s see an example:

App.cpp (push_back)

```
#include <vector>
#include <iostream>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y(v2), z(v1+1), t(v2+1) {
        printf("CTOR(%d,%d)\n", v1, v2);
    }
    Test(const Test& t): x(t.x), y(t.y), z(t.z), t(t.t) {
        printf("COPY-CTOR(%d,%d,%d,%d)\n", t.x, t.y, t.z, t.t);
    }
};

int main() {
    std::vector<Test> v;
    v.push_back(Test(10,20));

    return 0;
}
```

First construct the object Test, then copy or it into the vector !

```
CTOR(10,20)
COPY-CTOR(10,20,11,21)
```

STL - Vector

- Let’s see an example:

App.cpp (emplace_back)

```
#include <vector>
#include <iostream>

class Test {
    int x, y, z, t;
public:
    Test(int v1, int v2) : x(v1), y(v2), z(v1+1), t(v2+1) {
        printf("CTOR(%d,%d)\n", v1, v2);
    }
    Test(const Test& t): x(t.x), y(t.y), z(t.z), t(t.t) {
        printf("COPY-CTOR(%d,%d,%d,%d)\n", t.x, t.y, t.z, t.t);
    }
};

int main() {
    std::vector<Test> v;
    v.emplace_back(10,20);

    return 0;
}
```

We create the object directly in the vector

CTOR(10,20)

STL - Vector

- Vector also has a swap method that allows it to exchange the pointer towards inner data with another vector::

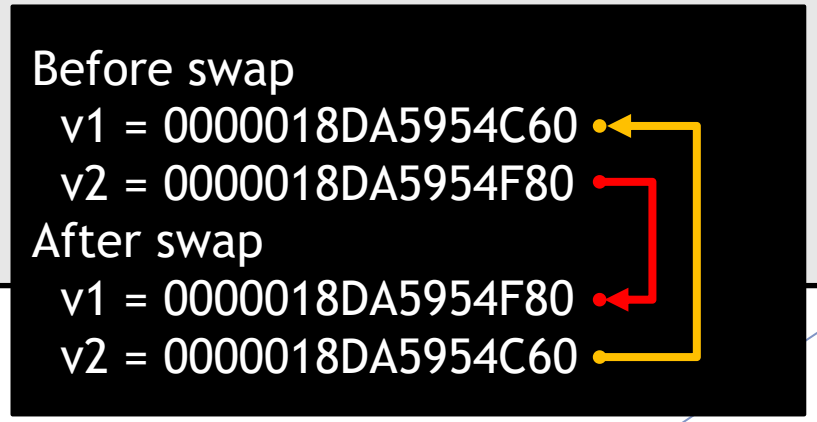
App.cpp

```
#include <vector>
#include <iostream>



int main() {
    std::vector<int> v1{ 1,2,3 };
    std::vector<int> v2{ 10,20 };
    printf("Before swap\n");
    printf("  v1 = %p\n", v1.data());
    printf("  v2 = %p\n", v2.data());
    v1.swap(v2);
    printf("After swap\n");
    printf("  v1 = %p\n", v1.data());
    printf("  v2 = %p\n", v2.data());
    return 0;
}
```

Notice that the pointer towards the actual data has been exchanged

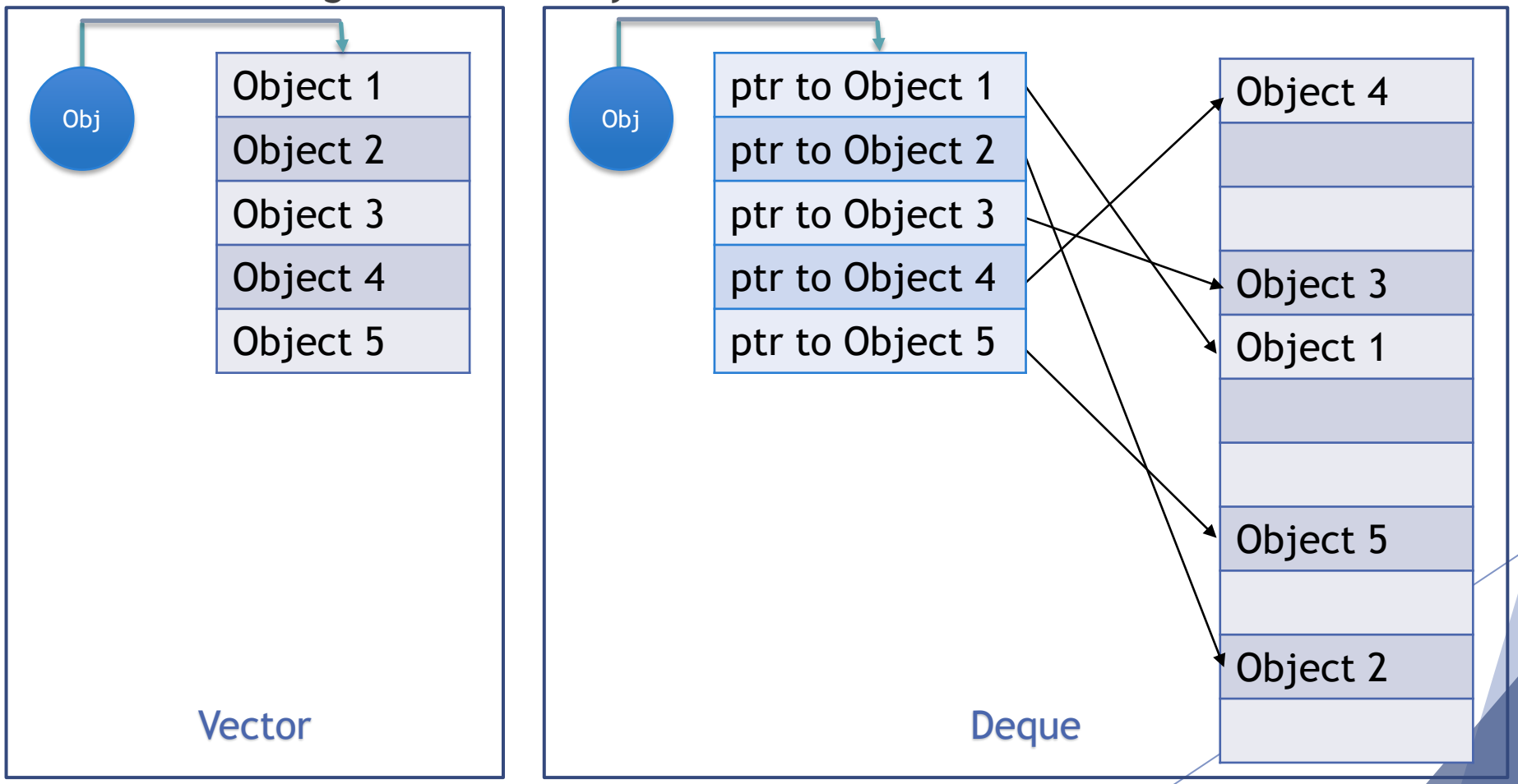


STL – Deque

- The “***deque***” template is very similar to “***vector***”.
- The main different is that elements don't have consecutive memory addresses. That is why the “data” method is not available for a deque. This also goes for the “***reserve***” and “***capacity***” methods.
- But deque has new methods (***push_front*** and ***pop_front***)
- The rest of the methods behave just like the ones in the “**vector**” class
- In order to use a deque: **“#include <deque>”**

STL – Deque

This is an illustrative picture that shows the difference between vector and deque. It should be NOTED that this is but am illustrative picture and does not reflect how the algorithms actually work.



STL – Deque

Let's retest the previous algorithm – this time we will use deque as well.

Vec-1.cpp

```
void main(void)
{
	vector<Integer> v;
	Integer i;
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v.push_back(i);
	}
}
```

Deq-1.cpp

```
void main(void)
{
	deque<Integer> v;
	Integer i;
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v.push_back(i);
	}
}
```

Vec-2.cpp

```
void main(void)
{
	vector<Integer> v;
	Integer i;
	v.reserve(100000);
	for (int tr = 0; tr < 100000; tr++)
	{
		i.Set(tr);
		v.push_back(i);
	}
}
```

STL – Deque

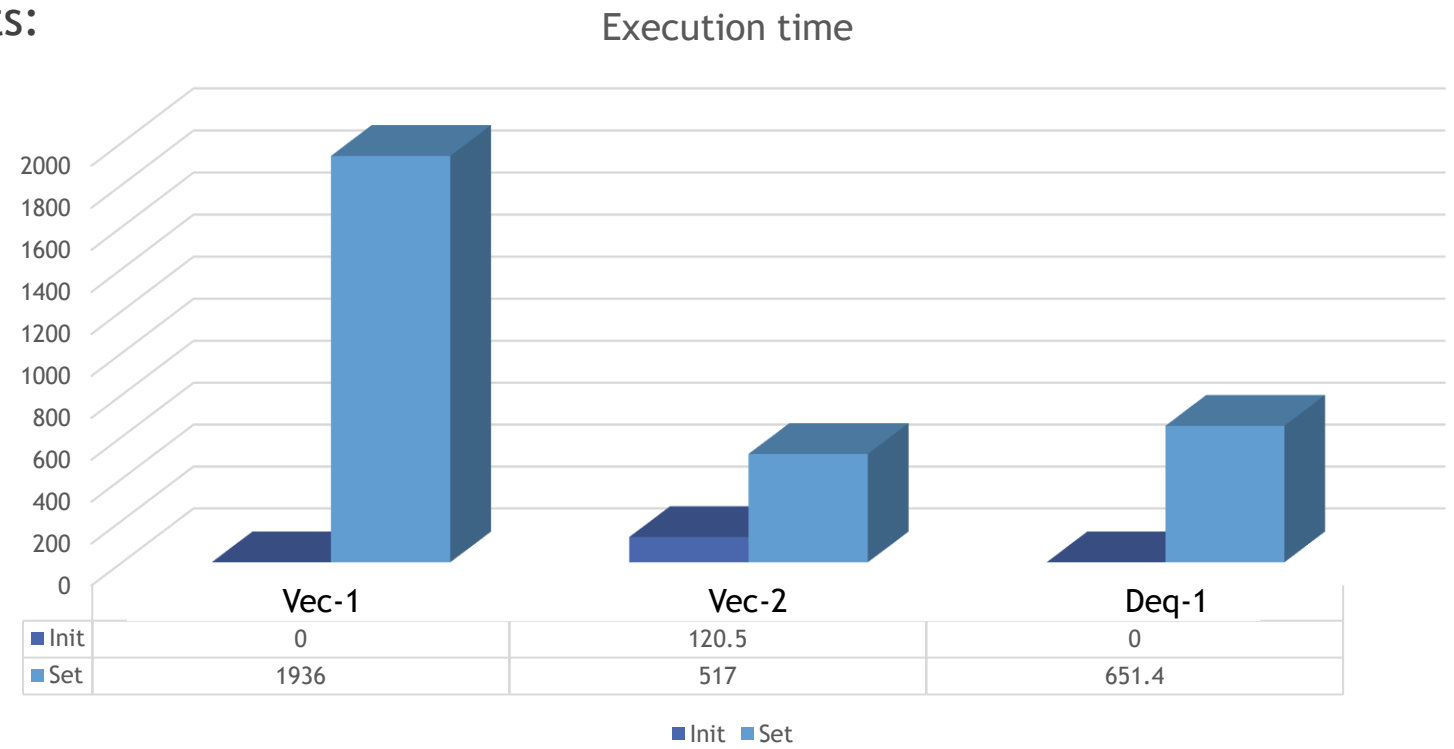
Results:

Alg	Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Vec-1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
Vec-2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
Deq-1	Init	0	0	0	0	0	0	0	0	0	0
	Set	687	657	656	640	656	640	656	641	641	640

- Vec-1 and Vect-2 ➔ two methods using ***vector***
- Deq-1 ➔ same algorithm using ***deque***
- “Init” ➔ the series of times recorded when we initialize the container
- “Set” ➔ the series of times needed to add the data in in container
- T1 .. T10 ➔ result times

STL – Deque

Results:



- Vec-1 and Vect-2 ➔ two methods using ***vector***
- Deq-1 ➔ same algorithm using ***deque***
- “Init” ➔ the series of times recorded when we initialize the container
- “Set” ➔ the series of times needed to add the data in in container

STL - Array

- The “array” template has been introduced in the C++11 standard.
- It represents a fixed size vector
- It has almost the same support as the vector class (iterators, overloaded operators, etc)
- It doesn't have any add methods (having a fixed size, it doesn't need them)
- For the same reason, it doen't have any methods that can be used to erase an element
- It has some methods that vector doesn not have (e.g. fill)
- In order to use an array: “**#include <array>**”

STL - Array

- An example on how to use array template:

App.cpp

```
class Number
{
	...
};
void main(void)
{
	array<Number,5> v;
	v.fill(Number(2));
	for (int tr = 0; tr < v.size(); tr++)
		printf("%d ", v[tr].Value);


	for (array<Number, 5>::iterator it = v.begin(); it < v.end(); it++)
		it->Value = 10;

	printf("%d ", v.at(3).Value);
}
```

- In case of ***array*** template, the size is predefined → this means that is somehow equivalent to a ***vector*** template that is initialized with using ***resize()*** method.

STL – Array & vector

- Both ***array*** and ***vector*** have 2 ways of accessing elements: the [] operator and the “at” method
- They both return an object reference and have 2 forms:
  - Type& operator[](size_type)
  - const Type& operator[](size_type) const
  - Type& at(size_type)
  - const Type& at(size_type)
  - size_type is defined as size_t (typedef size_t size_type;)
- There are differences between these two implementations that will be discussed in the next slides

STL – Array & vector

- Let's see how ***operator[]*** and method "***at***" are defined ?

App.cpp

```
reference at(size_type _Pos)
{
	if (_Size <= _Pos)
		_Xran();
	return (_Elems[_Pos]);
}
```

App.cpp

```
reference operator[](size_type _Pos)
{
#if _ITERATOR_DEBUG_LEVEL == 2
	if (_Size <= _Pos)
		_DEBUG_ERROR("array subscript out of range");
#elif _ITERATOR_DEBUG_LEVEL == 1
	_SCL_SECURE_VALIDATE_RANGE(_Pos < _Size);
#endif
	_Analysis_assume_(_Pos < _Size);
	return (_Elems[_Pos]);
}
with
#define _Analysis_assume_(expr) // for release
```

Compile method	Value of _ITERATOR_DEBUG_LEVEL
Release	0
Release (_SECURE_SCL)	1
Debug	2

STL - List

- The "***list***" template is a doubly-linked list
- Elements do not share a contiguous memory block
- Element access is possible only by moving through the list using iterators. The first and last element are easier to access
- The list iterator doesn't implement the **<** operator – the elements aren't stored in consecutive memory addresses, and so this type of comparison between these addresses is pointless. It does, however, implement the **==** operator.
- The list iterator does not implement the + and – operators; only **++** and **--** are implemented
- Methods used to add elements: **push_back and push_front**
- To remove elements: **erase**, **pop_front**, or **pop_back** can be used
- It supports a series of new methods: **merge**, **splice** (work with other lists)
- In order to use a list: **"#include <list>"**

STL - List

- An example of using the ***list*** template

App.cpp

```
class Number { ... };
void main(void)
{
	list<Number> v;
	v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
	v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

	list<Number>::iterator it;
	for (it = v.begin(); it < v.end(); it++)
		printf("%d ", it->Value);

	it = v.begin()+3;
	v.insert(it, Number(20));
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = ++v.begin();
	v.erase(it);
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);
}
```

- This code will not compile as the ***list*** template does not have ***operator<*** and ***operator>*** defined.

STL - List

- An example of using the ***list*** template

App.cpp

```
class Number { ... };
void main(void)
{
	list<Number> v;
	v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
	v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

	list<Number>::iterator it;
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = v.begin()+3;
	v.insert(it, Number(20));
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = ++v.begin();
	v.erase(it);
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);
}
```

- This code will not compile as the ***list*** template does not have ***operator+*** defined.

STL - List

- An example of using the ***list*** template

App.cpp

```
class Number { ... };
void main(void)
{
	list<Number> v;
	v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
	v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

	list<Number>::iterator it;
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = v.begin();it++;it++;it++;
	v.insert(it, Number(20));
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = ++v.begin();
	v.erase(it);
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);
}
```

- Now the code compiles and prints: "***5 4 3 0 1 2 5 4 3 20 0 1 2 5 3 20 0 1 2***"

STL – Forward List

- The “***forward_list***” template is a single linked list container. It has been added in the C++11 standard
- It respects almost the same logic as in the case of the doubly linked list.
- The iterator doesn't overload the -- operator, only ++ (the list is unidirectional)
- Some methods that list has are no longer found here: push_back and pop_back
- In order to use a forward_list: “***#include <forward_list>***”

STL - Forward List

- An example of using the ***forward_list*** template

App.cpp

```
class Number { ... };
void main(void)
{
	forward_list<Number> v;
	v.push_front(Number(0)); v.push_front(Number(1)); v.push_front(Number(2));

	forward_list<Number>::iterator it;
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = v.begin();
	it++; it++;
	v.insert_after(it, Number(20));
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);

	it = ++v.begin();
	v.erase_after(it);
	for (it = v.begin(); it != v.end(); it++)
		printf("%d ", it->Value);
}
```

- Now the code compiles and prints: "***2 1 0 2 1 0 20 2 1 20***"

STL

Method	vector	Deque	array	list	forward_list
Access	operator[] .at() .front() .back() .data()	operator[] .at() .front() .back()	operator[] .at() .front() .back() .data()	.front() .back()	.front()
Iterators	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()
Reverse Iterator	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	
Information	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .max_size()

STL

Method	vector	Deque	array	list	forward_list
Size/ Capacity	.resize() .reserve() .capacity()	.resize()		.resize()	.resize()
Add	.push_back() .insert()	.push_back() .push_front() .insert()		.push_back() .push_front() .insert()	.push_front() .insert_after()
Delete	.clear() .erase() .pop_back()	.clear() .erase() .pop_back() .pop_front()		.clear() .erase() .pop_back() .pop_front()	.clear() .pop_front() .erase_after()

- Adaptors

STL – Adaptors

- Adaptors are not containers – in the sense that they do not have an implementation that can store, they use an existing container that have this feature for storage
- Adaptors only work if the container that they use implement some specific functions
- Adaptors do NOT have iterators, but they can use iterators from the container that they use internally
- Adaptors:
  - stack
  - queue
  - priority_queue

STL - Stack

- This adaptor implements a stack (LIFO – **L**ast **I**n **F**irst **O**ut)
- To use : "`#include <stack>`"
- The default container is "***deque***" (in this example "s" uses ***deque*** while "s2" uses ***vector***)

App.cpp

```
void main(void)
{
	stack<int> s;
	s.push(10); s.push(20); s.push(30);

	stack<int, vector<int>> s2;
	s2.push(10); s2.push(20); s2.push(30);
}
```

- Has the following methods: ***push***, ***pop***, ***top***, ***empty***, ***size***
- It also implements a method called "***_Get_container***" that can be used to create iterators based on the container that is being used internally.

STL - Queue

- This adaptor implements a queue (FIFO – **F**irst **I**n **F**irst **O**ut)
- To use : "`#include <queue>`"
- The default container is "***deque***" (in this example "s" uses ***deque*** while "s2" uses ***list***)

App.cpp

```
void main(void)
{
	queue<int> s;
	s.push(10); s.push(20); s.push(30);

	queue<int, list<int>> s2;
	s2.push(10); s2.push(20); s2.push(30);
}
```

- Has the following methods: ***push***, ***pop***, ***back***, ***empty***, ***size***
- It also implements a method called "***_Get_container***" that can be used to create iterators based on the container that is being used internally.

STL – Priority Queue

- This is a queue where each elements has a priority. All elements are sorted out based on their priority (so that the element with the highest priority is extracted first).
- To use : "**#include <queue>**"
- The default container is "***vector***"

App.cpp

```
void main(void)
{
	priority_queue<int> s;
	s.push(10); s.push(5); s.push(20); s.push(15);
	while (s.empty() == false)
	{
		printf("%d ", s.top());
		s.pop();
	}
}
```

- This code prints: "**20 15 10 5**"
- Has the following methods: ***push***, ***pop***, ***top***, ***empty***, ***size***
- It also implements a method called "***_Get_container***" that can be used to create iterators based on the container that is being used internally.

STL – Priority Queue

- A ***priority_queue*** can be initialized with a class that has to implement "**operator()**". This operator is used to compare two elements (it should return true if the first one is smaller than the second one).

App.cpp

```
class CompareModule
{
	int modValue;
public:
	CompareModule(int v) : modValue(v) {}
	bool operator() (const int& v1, const int& v2) const
	{
		return (v1 % modValue) < (v2 % modValue);
	}
};
void main(void)
{
	priority_queue<int, vector<int>, CompareModule> s(CompareModule(3));
	s.push(10); s.push(5); s.push(20); s.push(15);
	while (s.empty() == false)
	{
		printf("%d ", s.top());
		s.pop();
	}
}
```

- This example prints: "**5 20 10 15**"

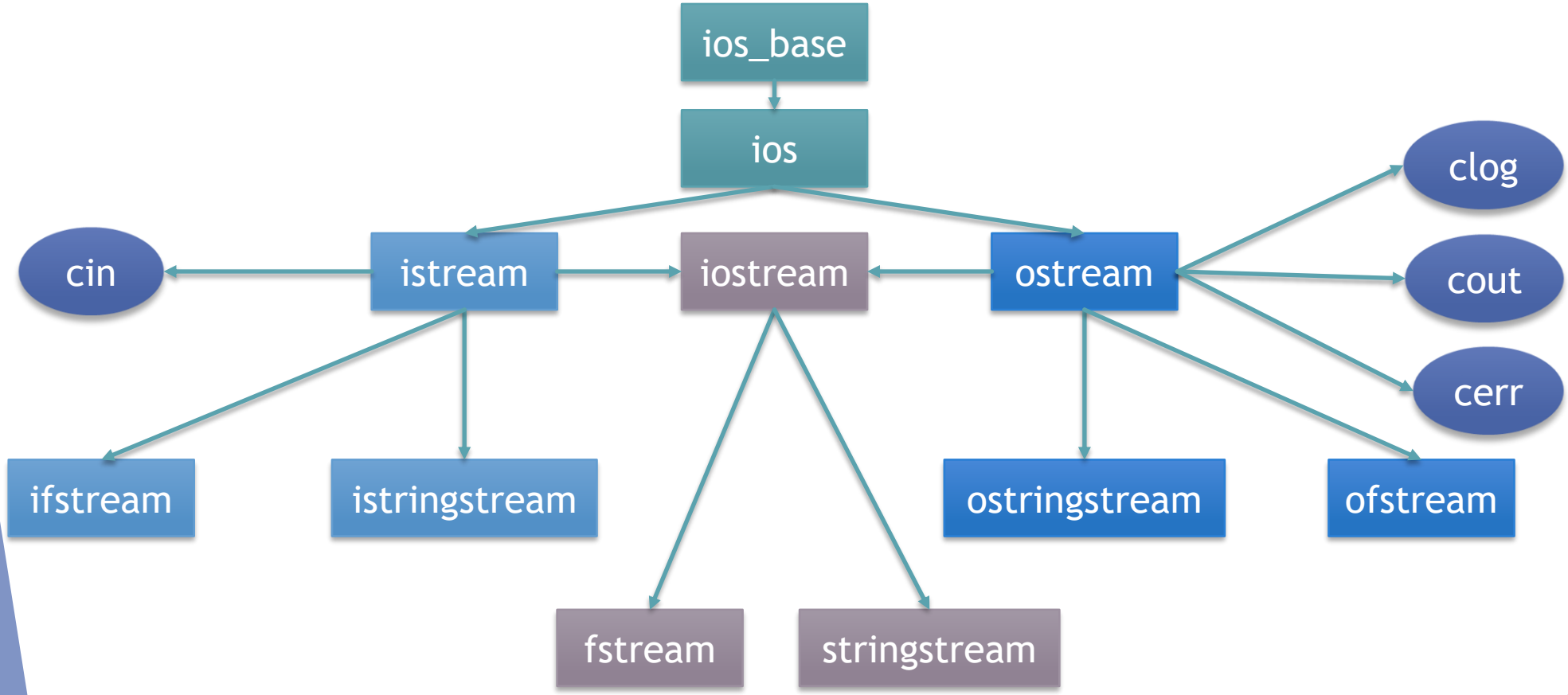
STL - Adaptors

Method	stack	queue	Priority_queue
push	Container.push_back	Container.push_back	Container.push_back
pop	Container.pop_back	Container.pop_front	Container.pop_back
top	Container.back	N/A	Container.front
back	N/A	Container.back	N/A
size	Container.size	Container.size	Container.size
empty	Container.empty	Container.empty	Container.empty

Adaptor	vector	deque	list	array	forward_list
stack	YES	YES	YES	NO	NO
queue	NO	YES	YES	NO	NO
priority_queue	YES	YES	NO	NO	NO

I/O Streams

STL (IOS)


ios_base
ios
clog
cout
cerr
cin
istream
iostream
ostream
ifstream
istringstream
ostringstream
ofstream
fstream
stringstream

STL(IOS)

- From the previous presented classes, the most used are **istream, ostream** and **iostream.**
- These classes allows the I/O access to miscellaneous streams (most well-known is the file system)
- Another use is represented by the **cin** and **cout** objects, which can be used to write to/read from the terminal
- Two operators are overloaded for these classes (operator>> and operator<<), representing the input from stream and the output to stream, respectively
- Besides the two operators, manipulators were also added to these classes (elements that can change the way of processing the data that follows them)

STL (IOS – manipulators)

manipulators	effect
endl	Adds a line terminator (“\n” , “\r\n”, etc) and flush
ends	Adds ‘\0’ (NULL)
flush	Clears the stream’s intern cache
dec	The numbers will be written in the base 10.
hex	The numbers will be written in the base 16.
oct	The numbers will be written in the base 8.
ws	Extracts and discards whitespace characters (until a non-whitespace character is found) from input
showpoint	Shows the decimal point
noshowpoint	Doesn’t show the decimal point
showpos	Add the “+” character in front of the positive numbers
noshowpos	Does not add the “+” character in front of the positive numbers

STL (IOS – manipulators)

manipulator	effect
boolalpha	Bool values will be written using “true” and ”false”
noboolalpha	Bool values will be written using 1 and 0
scientific	Scientific notation for float/double numbers
fixed	Floating-point values will be written using fixed-point notation
left	Left alignment
right	Right alignment
setfill(char)	Sets the fill character
setprecision(n)	Sets the precision for real numbers
setbase(b)	Sets the base

STL(IOS)

- Working with files is similar (use **ifstream** for input and **ofstream** for output)

App.cpp (using console)

```
#include <iostream>
using namespace std;

int main() {
    int x;
    cin >> x;
    for (int tr = 0; tr < x; tr++) {
        cout << tr << ",";
    }
    cout << endl;
    return 0;
}
```

App.cpp (using files)

```
#include <fstream>
using namespace std;

int main() {
    ifstream fin("input.txt");
    ofstream fout("output.txt");
    int x;
    fin >> x;
    for (int tr = 0; tr < x; tr++) {
        fout << tr << ",";
    }
    fout << endl;
    return 0;
}
```

- This code implies that input.txt file exists and contains a valid integer number

STL(IOS)

- Additionally, when creating a file a secondary parameter **ios_base::openmode** can be provided to control how that file is going to be used. Values:
  - **std::ios::in** – open for reading
  - **std::ios::out** – open for writing
  - **std::ios::app** – open for appending
  - **std::ios::ate** – open and seek to the end
  - **std::ios::trunc** – truncate the file if it exists
  - **std::ios::binary** – open in binary mode

STL(IOS)

- You can also read a file (line by line), or any other stream for that matter, using get_line() method from std and a std::string(…) method.

App.cpp

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream file("a.txt");

    if (file.is_open() == false) {
        printf("Fail to open the file !");
        return 1;
    }

    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << std::endl;
    }

    file.close();
    return 0;
}
```

STL(IOS)

- File operation also include methods similar to fread / fwrite / ftell / fseek that allows one to control the file. **ftell** and **fseek** like methods: are adjusted to work with stream:

```
basic_istream& seekg(off_type _Off, ios_base::seekdir _Way)
```

and

```
pos_type tellg()
```

- These methods are in particular useful when dealing with binary files.

STL(IOS)

- To read the content of a binary file into a buffer (a vector of unsigned char type) use the following example:

App.cpp

```
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    std::ifstream file("a.exe", std::ios::binary);

    file.seekg(0, std::ios::end);
    std::streamsize size = file.tellg();
    file.seekg(0, std::ios::beg);

    std::vector<char> buffer(size);
    file.read(buffer.data(), size);
    file.close();

    return 0;
}
```

STL(IOS)

- To read the content of a binary file into a buffer (a vector of unsigned char type) use the following example:

App.cpp

```
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    std::ifstream file("a.exe", std::ios::binary);

    file.seekg(0, std::ios::end);
    std::streamsize size = file.tellg();
    file.seekg(0, std::ios::beg);

    std::vector<char> buffer(size);
    file.read(buffer.data(), size);
    file.close();

    return 0;
}
```

Find the size of the file so that we know how much to allocate in the vector

STL(IOS)

- To read the content of a binary file into a buffer (a vector of unsigned char type) use the following example:

App.cpp

```
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    std::ifstream file("a.exe", std::ios::binary);

    file.seekg(0, std::ios::end);
    std::streamsize size = file.tellg();
    file.seekg(0, std::ios::beg);

    std::vector<char> buffer(size);
    file.read(buffer.data(), size);
    file.close();

    return 0;
}
```

This methods might fail (e.g. somebody has deleted the file) and its result should be checked.

- Strings

STL (basic_string)

- Template that provides the most used operations over strings
- The declaration:

App.cpp

```
template <class CharacterType, class traits = char_traits<CharacterType>>
class basic_string
{
...
}
```

- The most common objects that implements this template are **string** and **wstring**

App.cpp

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

- Other objects introduced Cx11 based on this template:

App.cpp

```
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

STL (basic_string)

- "char_traits" is a template that provides a list of operations over the strings (not necessarily characters of type **char**) used by **basic_string** for some operations
- The main methods of char_traits are:

Definition	Functionality
`static bool eq (CType c1, CType c2)`	Returns **true** if c1 is equal with c2
`static bool lt (CType c1, CType c2)`	Returns **true** if c1 is lower than c2
`static size_t length (const CType* s);`	Returns the size of a string
`static void assign (CType& character, const CType& value)`	Assignment (character = value)
`static int compare (const CType* s1, const CType* st2, size_t n);`	Compares two strings; returns 1 if s1 > s2, 0 if s1 = s2 and -1 if s1 < s2
`static const char_type* find (const char_type* s, size_t n, const char_type& c);`	Returns a pointer to the first character equal with **c** from the string **s**

STL (basic_string)

- The main methods of char_traits are:

Definition	Functionality
`static char_type* move (char_type* dest, const char_type* src, size_t n);`	Moves the content of a string between two locations
`static char_type* copy(char_type* dest, const char_type* src, size_t n);`	Copies the content of a string from one location to another
static int_type eof()	Returns a value for EOF (usually -1)

- “char_traits” has specializations for <char>, <wchar> which are using efficient functions like memcpy, memmove, etc.
- For normal use of strings, a char_traits object is not needed. However, it is useful for the cases where we want a particular behavior (some comparisons or assignments to be made differently – e.g. case insensitive)

STL (basic_string)

- “basic_string” supports various forms of initialization:

App.cpp

```
void main(void)
{
	string s1(“Tomorrow");
	string s2(s1+" I’ll go ");
	string s3(s1, 3, 3);
	string s4("Tomorrow I’ll go to my cousin.", 13);
	string s5(s4.substr(9,10));
	string s6(10, '1');

	printf("%s\n%s\n%s\n%s\n%s\n%s\n", s1.c_str(),
					s2.c_str(),
					s3.c_str(),
					s4.c_str(),
					s5.c_str(),
					s6.c_str());
}
```

Output

```
Tomorrow
Tomorrow I’ll go
orr
Tomorrow I’ll
I’ll go to
1111111111
```

STL (basic_string)

- Methods/operators supported by the objects derived from the basic_string template:

Method/operator
Assignment (**operator=**)
Append (**operator+=** and **append**, **push_back** methods)
Characters insertion (**insert** method)
Access to characters (**operator[]** and **at** , **front** (C++11) , **back** (C++11) methods)
Substrings (**substr** method)
Replace (**replace** method)
Characters deletion (**erase**, **clear**, and **pop_back** (C++11) methods)
Search (**find**, **rfind**, **find_first_of**, **find_last_of**, **find_first_not_of**, **find_last_not_of** methods)
Comparations (**operator>** , **operator<** , **operator==** , **operator!=** , **operator>=** , **operator<=** and **compare** method)
Iterators (**begin**,**end**,**rbegin**,**rend**,**cbegin**,**cend**,**crbegin**,**crend** (the last four from C++11))
Informations (**size**, **length**, **empty**, **max_size**, **capacity** methods)

STL (basic_string)

- An example of working with **String** objects:

App.cpp

```
void main(void)
{
	string s1;
	s1 += “Now";
	printf("Size = %d\n", s1.length());
	s1 = s1 + " " + s1;
	printf("S1 = %s\n", s1.data());
	s1.erase(2, 4);
	printf("S1 = %s\n", s1.c_str());
	s1.insert(1, "_");
	s1.insert(3, "__");
	printf("S1 = %s\n", s1.c_str());
	s1.replace(s1.begin(), s1.begin() + 2, "123456");
	printf("S1 = %s\n", s1.c_str());
}
```

Output

```
Size = 3
S1 = Now Now
S1 = Now
S1 = N_o__w
S1 = 123456o__w
```

STL (basic_string)

- Some example of using ***char_traits*** (a string with ***ignore case***):

App.cpp

```
struct IgnoreCase : public char_traits<char> {
	static bool eq(char c1, char c2) {
		return (upper(c1)) == (upper(c2));
	}
	static bool lt(char c1, char c2) {
		return (upper(c1)) < (upper(c2));
	}
	static int compare(const char* s1, const char* s2, size_t n) {
		while (n>0)
		{
			char c1 = upper(*s1);
			char c2 = upper(*s2);
			if (c1 < c2) return -1;
			if (c1 > c2) return 1;
			s1++; s2++; n--;
		}
		return 0;
	}
};
void main(void)
{
	basic_string<char, IgnoreCase> s1("Salut");
	basic_string<char, IgnoreCase> s2("sAlUt");
	if (s1 == s2)
		printf("Siruri egale !");
}
```

STL (string_view)

- STL also has a string_view object that holds a pointer to a buffer of characters and the size.
- This object does not own the actual string
- The pointer does not necessarily points to a memory buffer where a string that ends up with a \0 character lies.

App.cpp

```
#include <string>
#include <iostream>

int main() {
    std::string_view s("Test", 4);

    std::cout << s;

    return 0;
}
```

STL (string_view)

- You can obtain a string_view from an existing string, or as a part (view) of the string.

App.cpp

```
#include <string>
#include <iostream>

int main() {
    std::string s = "C++ exam";

    std::string_view sv = s;
    std::cout << sv << std::endl;

    sv = sv.substr(1, 2);
    std::cout << sv << std::endl;

    return 0;
}
```

```
C++ exam
++
```

STL (string_view)

- A string_view contains most of the functionality that a string has.
  - Iteartors : begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend()
  - substr() → to create another string_view
  - Search methods: find, rfind, find_first_of, find_last_of
  - Compare methods and operators: compare(…) or operators
  - Index operator: [] or “at” method
  - Raw access to data: data() method
  - Check methods (.empty() – that tells you if the inner string is empty.
  - Dimension: .size() → returns the number of characters in the string

- string_view also has templates for Unicode 16, Unicode 32, Wide char
- string_view is available from **C++17**

STL (string_view)

- Be careful when you use string_view (as it does not see a string as an ASCIIZ). This means that using .data() method could produce an undefined behavior.

App.cpp

```
#include <string>
#include <iostream>

int main() {
    char text[3] = {'C','+','+'};

    std::string_view sv(text,3);
    std::cout << sv << std::endl;

    printf("%s", sv.data());

    return 0;
}
```

STL (string_view)

- Be careful when you use string_view (as it does not see a string as an ASCIIZ). This means that using .data() method could produce an undefined behavior.

App.cpp

```
#include <string>
#include <iostream>

int main() {
    char text[3] = {'C','+','+'};

    std::string_view sv(text,3);
    std::cout << sv << std::endl;

    printf("%s", sv.data());

    return 0;
}
```

The problem is that sv.data() returns a **const char*** and printf method searches for \0 character. However, the string_view is a view inside the stack (and we have build the variable text as an array of 3 elements and not as a string). As such, the printf will iterate in the stack until it find a \0 character.

- Initialization lists

Initialization lists

- Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

struct Student
{
	const char * Name;
	int Grade;
};

void main()
{
	vector<int> v = { 1, 2, 3 };
	vector<string> s = { "POO", "C++" };
	vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };

}
```

- The code complies ok and all objects are initialized properly.

Initialization lists

- Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

struct Student
{
	const char * Name;
	int Grade;
};

void main()
{
	vector<int> v = { 1, 2, 3 };
	vector<string> s = { "POO", "C++" };
	vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };

}
```

The “st” vector will hold two elements of type Student (Popescu & Ionescu).

- The code complies ok and all objects are initialized properly.

Initialization lists

- STL also provide a special container (called **std::initializer_list** that can be used to pass a initialization list to a function).

App.cpp

```
using namespace std;
#include <initializer_list>

int Sum(std::initializer_list<int> a)
{
	int result = 0;
	std::initializer_list<int>::iterator it;
	for (it = a.begin(); it != a.end(); it++)
		result += (*it);
	return result;
}

void main()
{
	printf("Sum = %d\n", Sum( { 1, 2, 3, 4, 5 } ));
}
```

- This code will compile and will print to the screen the value 15.

Initialization lists

- If **std::initializer_list** is used in a constructor the following expression for initializing an object can be used:

App.cpp

```
using namespace std;
#include <initializer_list>

class Data
{
	int v[10];
public:
	Data(std::initializer_list<int> a)
	{
		int index = 0;
		std::initializer_list<int>::iterator it;
		for (it = a.begin(); (it != a.end()) && (index<10); it++,index++)
			v[index] = (*it);
	}
};
void main()
{
	Data d1({ 1, 2, 3, 4, 5 });
	Data d2 = { 1, 2, 3, 4, 5 };
}
```

Initialization lists

- Let’s consider the following code:

App.cpp

```
#define DO_SOMETHING(x) x
void main() { DO_SOMETHING(printf("Test");); }
```

- This code will work ok, and the compiler will print “Test” to the screen. Now let’s consider the following one:

App.cpp

```
#define DO_SOMETHING(x) x
void main() { DO_SOMETHING( vector<int> x{1, 2, 3}; ); }
```

- In this case the code will not compile. When analyzing a MACRO the compiler does not interpret in any way the “{“ character. This means that from the compiler point of view, DO_SOMETHING macro has received 3 parameters:
  - `vector<int> x{1`
  - `2`
  - `3}`

Initialization lists

- In this cases the solution is to change the macro from:

App.cpp

```
#define DO_SOMETHING(x) x
```

to

App.cpp

```
#define DO_SOMETHING(…) __VA_ARGS__
void main() { DO_SOMETHING( vector<int> x{1, 2, 3}; ); }
```

- This code will compile and run correctly.

Q & A