

# OOP

Gavrilut Dragos  
Course 7

# Summary

- ▶ Standard Template Library (STL)
- ▶ Sequence containers
- ▶ Adaptors
- ▶ Associative containers
- ▶ 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-Containere

## ▶ 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; }
};

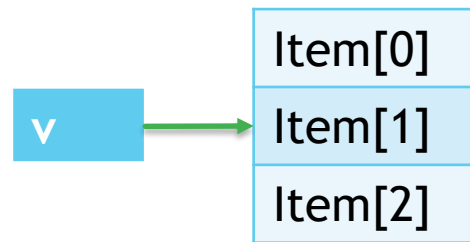
void main(void)
{
    vector<Integer> v;
    Integer i(5);
    for (int tr = 0; tr < 5; tr++)
    {
        i.Set(1000 + tr);
        v.push_back(i);
    }
}
```

# STL - Vector

Example:

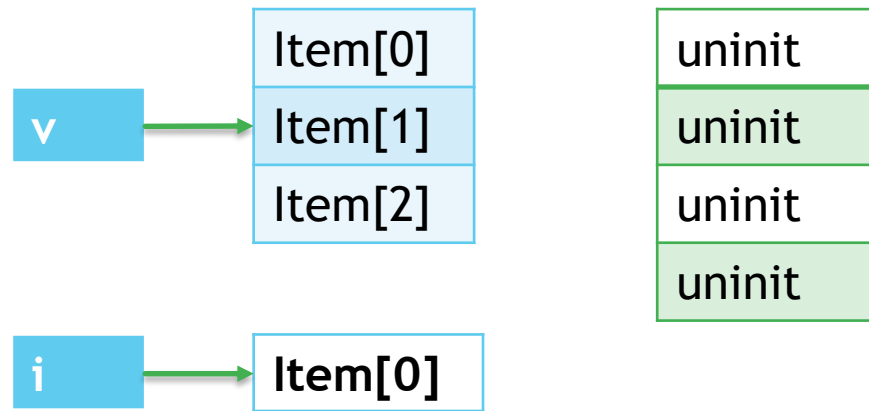
Pas	Output
-	[0098FC90] Value ctor(5)
tr=0	[00D3A688] Copy ctor from (0098FC90,1000)
tr=1	[00D3A6C8] Copy ctor from (00D3A688,1000) [00D3A6CC] Copy ctor from (0098FC90,1001)
tr=2	[00D3A710] Copy ctor from (00D3A6C8,1000) [00D3A714] Copy ctor from (00D3A6CC,1001) [00D3A718] Copy ctor from (0098FC90,1002)
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)`

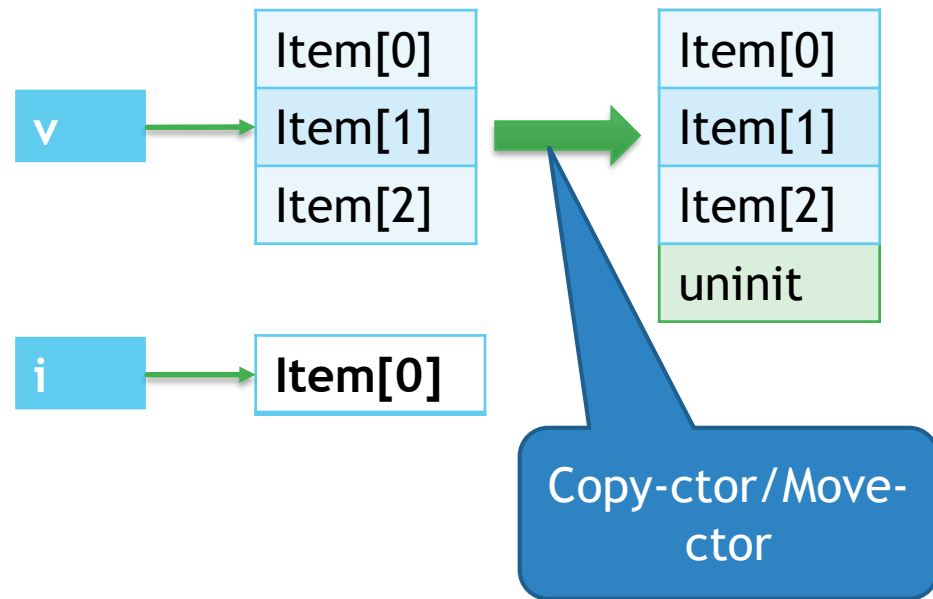
# STL - Vector



`v.push_back(i)`

- ▶ Allocate space for `count+1` elements

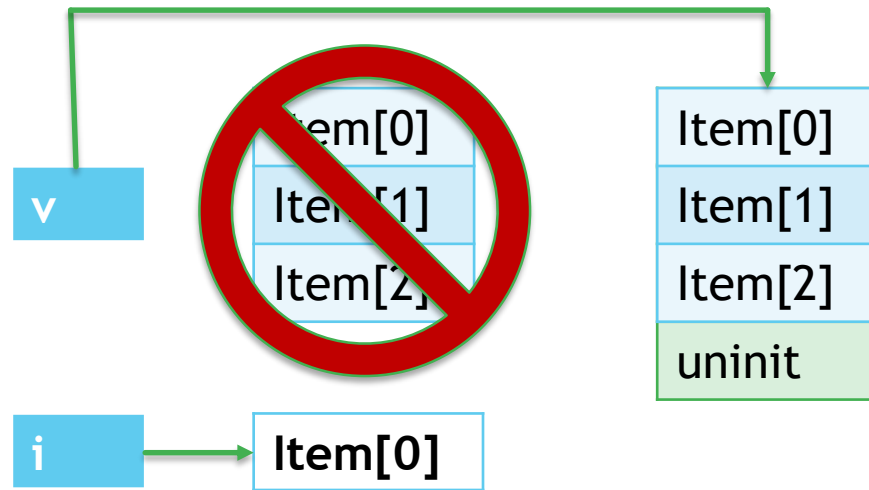
# STL - Vector



## `v.push_back(i)`

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

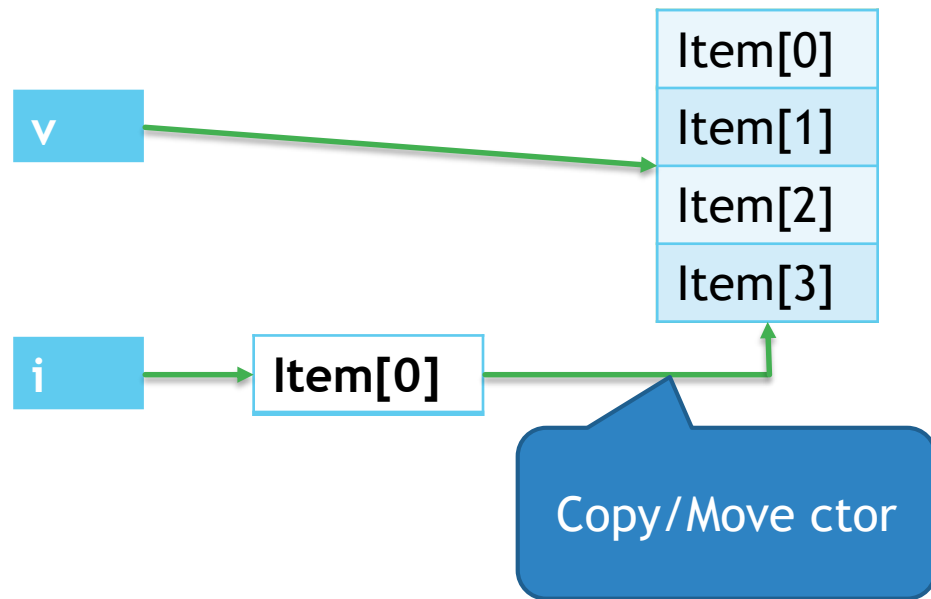
# STL - Vector



## `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);
    }
}
```

## 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);
    }
}
```

## 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-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 times. 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

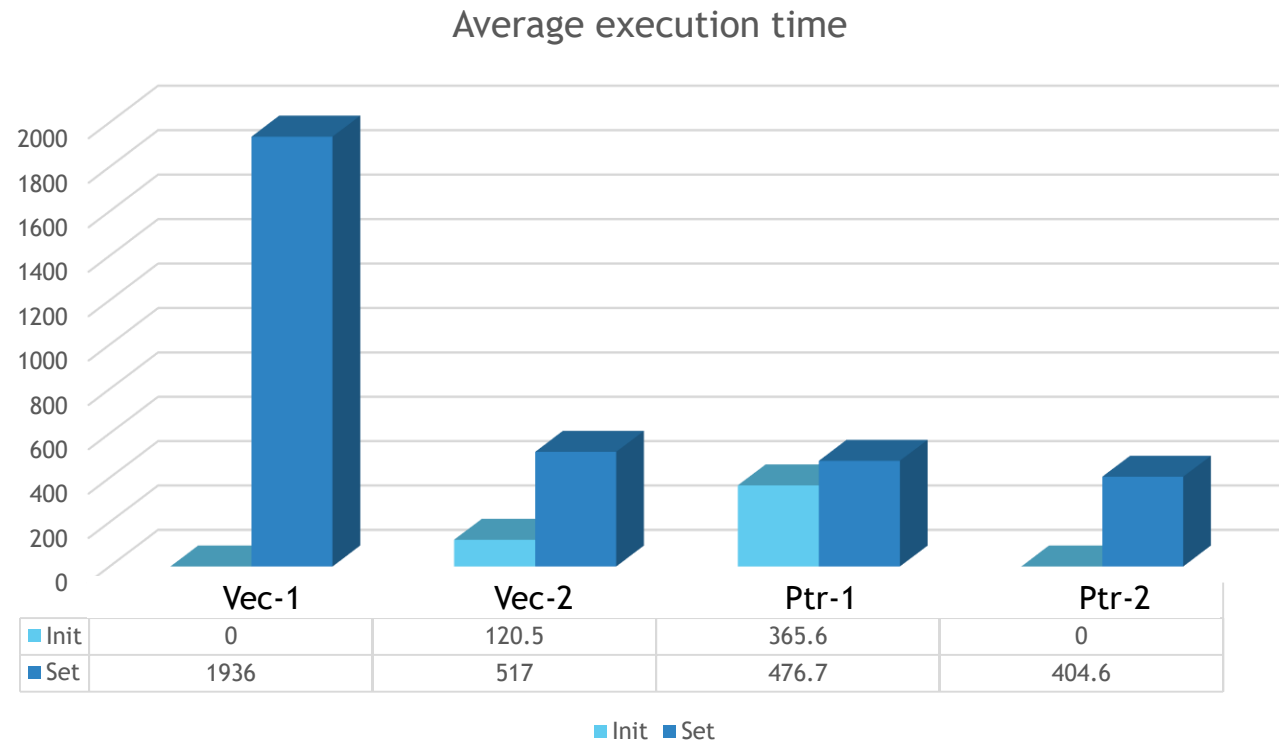
Results:

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:



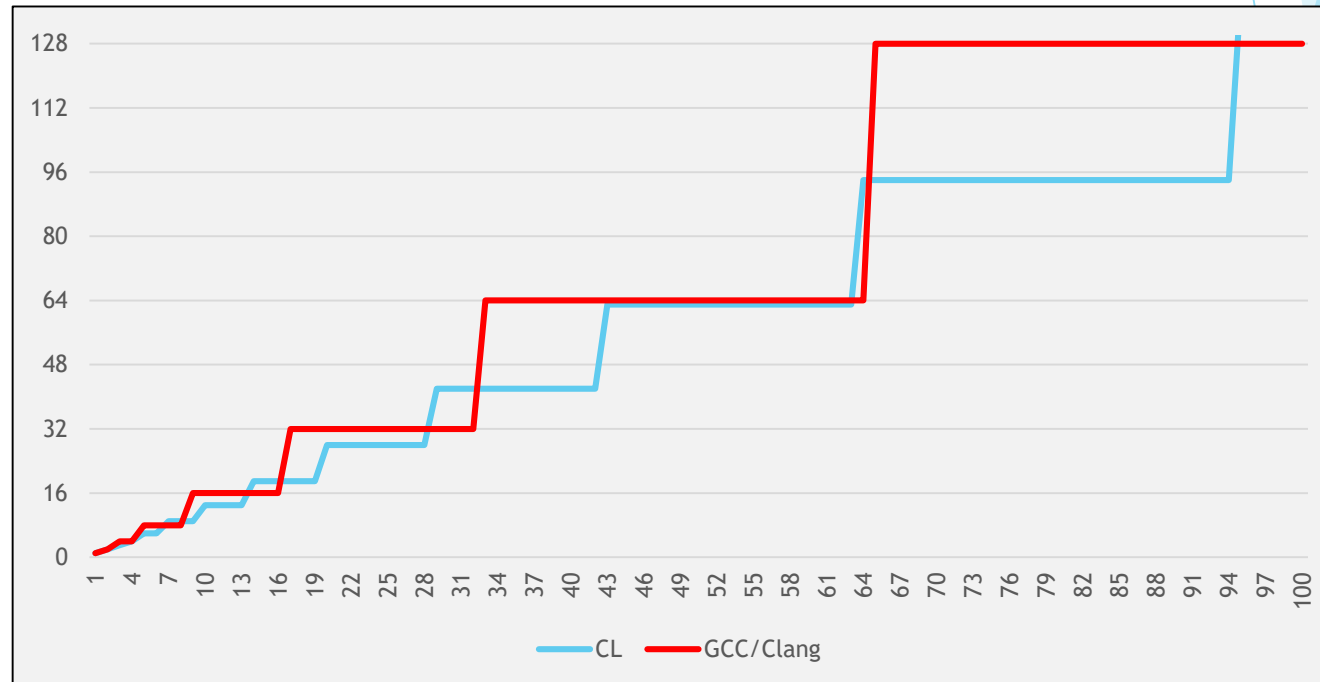
- ▶ Vec-1,Vec-2,Ptr-1,Ptr-2 → the 4 methods previously described
- ▶ “Init” → the series of times recorded when we initialize the container
- ▶ “Set” → the series of times needed to add the data in in container

# 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

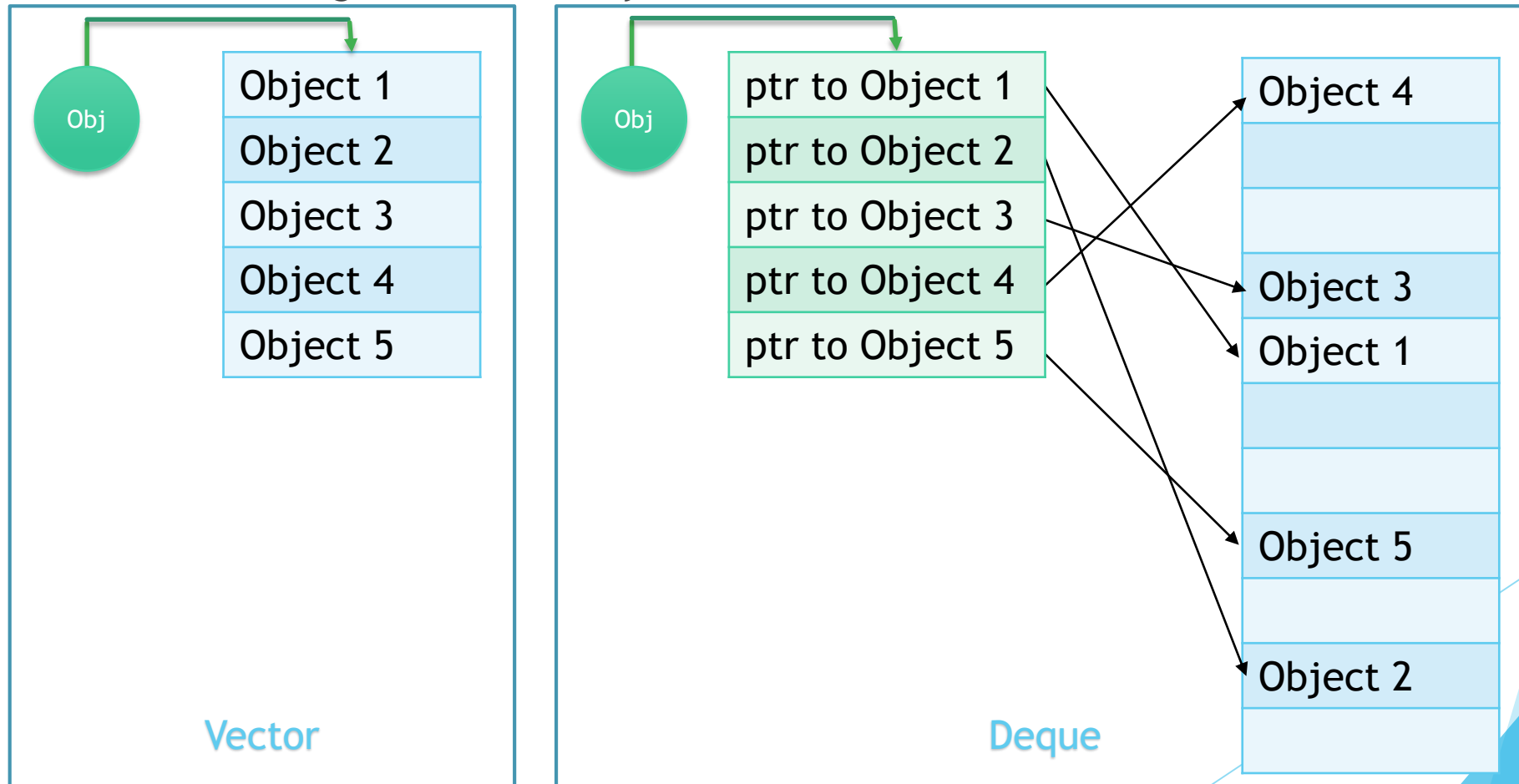
```
class Number { ... };  
void main(void)  
{  
    vector<Number> v;  
    v.push_back(Number(0)); v.push_back(Number(1));  
    printf("%d ", v.max_size());  
    printf("%d ", v.size());  
    printf("%d ", v.capacity());  
    Number *n = v.data();  
    printf("[%d %d]", n[0].Value, n[1].Value);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

# STL - 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 an 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

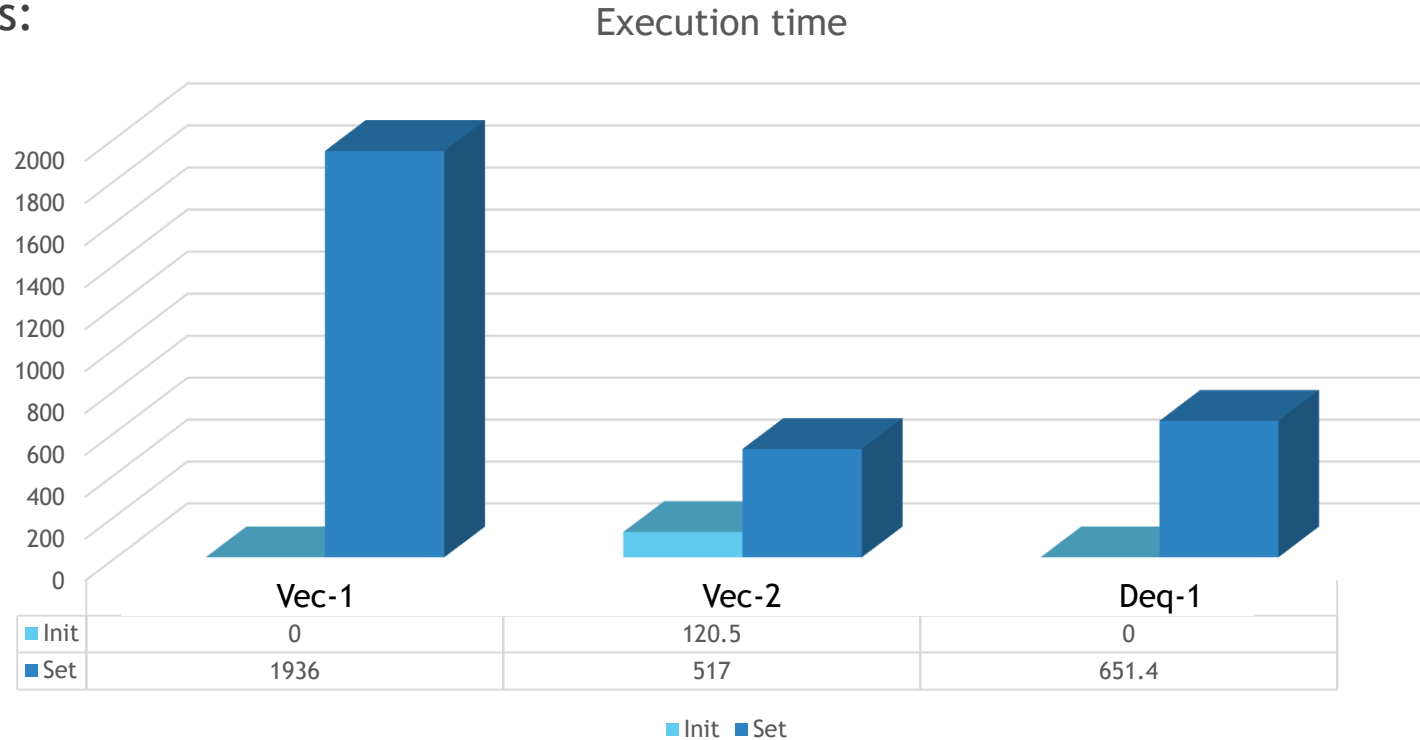
Results:

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	3509	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 doesn't have any methods that can be used to erase an element
- ▶ It has some methods that vector doesn't 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 - Last In First Out)
- ▶ 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 - First In First Out)
- ▶ 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



# Associative

- ▶ containers

# STL (associative containers - pair)

- ▶ “pair” is a template that contains two values (two different types)
- ▶ For use “**#include <utility>**”
- ▶ Two objects of type **pair** can be compared because the = operator is overloaded
- ▶ It's internally used by associative containers
- ▶ The declaration of **pair** template:

## App.cpp

```
template <class T1, class T2>
struct pair
{
    T1 first;
    T2 second;
    ...
}
```

# STL (associative containers - map)

- ▶ Map is a container that store pairs of the form: key/value; the access to the **value** field can be done using the **key**
- ▶ For use “**#include <map>**”
- ▶ The key field is constant: after a key/value pair is added into a map, the key cannot be modified - only the value can be modified. Other pairs can be added and existing pairs can be deleted.
- ▶ The declaration of **map** template:

## App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class map
{
    ...
}
```



# STL (associative containers - map)

- ▶ An example of using map:

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    Grades["Ionescu"] = 9;
    Grades["Marin"] = 7;
    printf("Ionescu's grade = %d\n", Grades["Ionescu"]);
    printf("Number of pairs = %d\n", Grades.size());
    map<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
        printf("Grades[%s]=%d\n", it->first, it->second);
    it = Grades.find("Ionescu");
    Grades.erase(it);
    int x = Grades ["Ionescu"];
    printf("Ionescu's grade = %d (x=%d)\n", Grades["Ionescu"],x);
    printf("Number of pairs = %d\n", Grades.size());
}
```

## Output

```
Ionescu's grade = 9
Number of pairs = 3
Grades[Popescu]=10
Grades[Ionescu]=9
Grades[Marin]=7
Ionescu's grade = 0 (x=0)
Number of pairs = 3
```

# STL (associative containers - map)

- ▶ An example of using map:

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    Grades["Ionescu"] = 9;
    Grades["Marin"] = 7;
    printf("Ionescu's grade = %d\n", Grades["Ionescu"]);
    printf("Number of pairs = %d\n", Grades.size());
    map<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
        printf("Grades[%s]=%d\n", it->first, it->second);
    it = Grades.find("Ionescu");
    Grades.erase(it);
    int x = Grades["Ionescu"];
    printf("Ionescu's grade = %d (x=%d)\n", Grades["Ionescu"], x);
    printf("Number of pairs = %d\n", Grades.size());
}
```

The "Ionescu" key does not exist at this point. Because it not exists, a new one is created and the value is instantiated using the default constructor (which for int makes the value to be 0)

# STL (associative containers - map)

- ▶ The methods supported by the map container:

Method/operator
Assignment ( <b>operator=</b> )
Accessing and inserting values ( <b>operator[]</b> and <b>at</b> , <b>insert</b> methods)
Deletion ( <b>erase</b> , <b>clear</b> methods)
Search into a map ( <b>find</b> method)
Iterators ( <b>begin</b> , <b>end</b> , <b>rbegin</b> , <b>rend</b> , <b>cbegin</b> , <b>cend</b> , <b>crbegin</b> , <b>crend</b> (the last 4 from C++11) )
Informations ( <b>size</b> , <b>empty</b> , <b>max_size</b> )

# STL (associative containers - map)

- ▶ Accessing elements ([] operator and at, find methods):

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade = %d\n", Grades["Popescu"]);
}
```

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.at("Popescu"));
}
```

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.find("Popescu")->second);
}
```

# STL (associative containers - map)

- ▶ Accessing elements ([] operator and at, find methods):

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade = %d\n", Grades["Popescu"]);
}
```

## App.cpp ("Ionescu" key does not exist)

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.at("Ionescu"));
}
```

## App.cpp ("Ionescu" key does not exist)

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.find("Ionescu")->second);
}
```

# STL (associative containers - map)

- ▶ Checking whether an element exists in a map can be done using **find** and **count** methods:

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    if (Grades.find("Ionescu")==Grades.cend())
        printf("Ionescu does not exist in the grades list!");
}
```

## App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    if (Grades.count("Ionescu")==0)
        printf("Ionescu does not exist in the grades list!");
}
```

# STL (associative containers - map)

- ▶ The elements from a map containers are stored into a red-black tree
- ▶ This represents a compromise between the access/insertion time and the amount of allocated memory for the container
- ▶ Depending on what we are interested in, a map container is not always the best solution (e.g. if the number of insertions is much smaller than the number of reads, there are containers that are more efficient)
- ▶ We do the following experiment - the same algorithm is written using a map and a simple vector and we evaluate the insertion time
- ▶ The experiment is repeated ten times for each algorithm and the execution times are measured in milliseconds

# STL (associative containers - map)

- ▶ The two algorithms:

## Map-1.cpp

```
void main(void)
{
    map<int, int> Test;
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr] = tr;
}
```

## Map-2.cpp

```
void main(void)
{
    int *Test = new int[1000000];
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr] = tr;
}
```

- ▶ Even if it is obvious that App-2 is more efficient, the hash collisions must be considered. For the above case, there are no hash collisions because the keys are **integers**.



# STL (associative containers - map)

## ► Results:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3

## ► The specifications of the system:

- ❖ 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 (associative containers - multimap)

- ▶ “multimap” is a container similar to map. The difference is that a key can contain more values.
- ▶ For use “**#include <map>**”
- ▶ Accessing elements: the [] operator and the at method can not be used anymore.
- ▶ The declaration of **multimap** template:

## App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class multimap
{
    ...
}
```

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));

    multimap<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
    {
        printf("%s [%d]\n", it->first, it->second);
    }
}
```

## Output

```
Ionescu [10]
Ionescu [8]
Ionescu [7]
Popescu [9]
```

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    it = Grades.begin();
    printf("%s->%d\n", it->first, it->second);

    it = Grades.upper_bound(it->first);
    printf("%s->%d\n", it->first, it->second);

    it = Grades.upper_bound(it->first);
    printf("%s->%d\n", it->first, it->second);
}
```

## Output

```
Ionescu->10
Popescu->9
Georgescu->8
```

# STL (associative containers - multimap)

- An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
    {
        printf("Unique key: %s\n", it->first);
    }
}
```

## Output

```
Unique key: Ionescu
Unique key: Popescu
Unique key: Georgescu
```

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    it = Grades.begin();
    while (it != Grades.end())
    {
        pair <multimap<const char*, int>::iterator, multimap<const char*, int>::iterator> range;
        range = Grades.equal_range(it->first);
        printf("%s's grades: ", it->first);
        for (it = range.first; it != range.second; it++)
            printf("%d,", it->second);
        printf("\n");
    }
}
```

## Output

Ionescu's grades: 10,8,7  
Popescu's grades:9,6  
Georgescu's grades:8

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it,it2;
    it = Grades.find("Popescu");
    it2 = Grades.upper_bound(it->first);

    printf("Popescu's grades: ");
    while (it != it2)
    {
        printf("%d,", it->second);
        it++;
    }
}
```

## Output

Popescu's grades: 9,6

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it,it2;
    for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
    {
        printf("%s has %d grades\n", it->first, Grades.count(it->first));
    }
}
```

## Output

Ionescu has 3 grades  
Popescu has 2 grades  
Georgescu has 1 grades



# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it, it2;
    it = Grades.find("Ionescu");
    it++;
    Grades.erase(it); // the "8" grade from Ionescu is deleted
    for (it = Grades.begin(); it != Grades.end(); it++)
    {
        printf("%s [%d]\n", it->first, it->second);
    }
}
```

## Output

```
Ionescu [10]
Ionescu [7]
Popescu [9]
Popescu [6]
Georgescu [8]
```

# STL (associative containers - multimap)

- ▶ An example of using multimap:

## App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    if (Grades.find("Ionescu") != Grades.cend())
        printf("Ionescu exists!\n");

    if (Grades.find("Marin") == Grades.cend())
        printf("Marin DOES NOT exist!\n");
}
```

## Output

```
Ionescu exists!
Marin DOES NOT exist!
```

# STL (associative containers - multimap)

- ▶ The methods supported by the multimap container:

Method/operator
Assignment ( <b>operator=</b> )
Insertion ( <b>insert</b> )
Deletion ( <b>erase</b> , <b>clear</b> methods)
Accessing elements ( <b>find</b> method)
Iterators ( <b>begin</b> , <b>end</b> , <b>rbegin</b> , <b>rend</b> , <b>cbegin</b> , <b>cend</b> , <b>crbegin</b> , <b>crend</b> (the last 4 from C++11) )
Informations ( <b>size</b> , <b>empty</b> , <b>max_size</b> methods)
Special methods ( <b>upper_bound</b> and <b>lower_bound</b> - to access the intervals in witch there are elements with the same key; <b>equal_range</b> - to obtain an interval for all the elements stored for a key)

# STL (associative containers - set)

- ▶ “set” is a container that store unique elements
- ▶ For use “**#include <set>**”
- ▶ The declaration of **set** template:

## App.cpp

```
template < class Key, class Compare = less<Key> > class set
{
    ...
}
```

# STL (associative containers - set)

- ▶ An example of using set:

## App.cpp

```
void main(void)
{
    set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(5);
    s.insert(10);
    set<int>::iterator it;

    for (it = s.begin(); it != s.end(); it++)
        printf("%d ", *it);
}
```

## Output

5 10 20

# STL (associative containers - set)

- An example of using set:

## App.cpp

```
struct Comparator {  
    bool operator() (const int& leftValue, const int& rightValue) const  
    {  
        return (leftValue / 20) < (rightValue / 20);  
    }  
};  
  
void main(void)  
{  
    set<int, Comparator> s;  
    s.insert(10);  
    s.insert(20);  
    s.insert(5);  
    s.insert(10);  
    set<int, Comparator>::iterator it;  
  
    for (it = s.begin(); it != s.end(); it++)  
        printf("%d ", *it);  
}
```

## Output

10 20

# STL (associative containers - set)

- ▶ The elements from a “set” follows a specific order.
- ▶ This brings a performance penalty.

## Set-1.cpp

```
void main(void)
{
    set<int> s;
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158

# STL (associative containers - set)

- ▶ The methods supported by the set container:

Method/operator
Assignment ( <b>operator=</b> )
Insertion ( <b>insert</b> method)
Deletion ( <b>erase</b> , <b>clear</b> methods)
Accessing elements ( <b>find</b> method)
Iterators ( <b>begin</b> , <b>end</b> , <b>rbegin</b> , <b>rend</b> , <b>cbegin</b> , <b>cend</b> , <b>crbegin</b> , <b>crend</b> (the last 4 from C++11) )
Informations ( <b>size</b> , <b>empty</b> , <b>max_size</b> methods)



# STL (associative containers - multiset)

- ▶ “**multiset**” is similar with set, but duplicate elements are allowed
- ▶ For use “**#include <set>**”
- ▶ The declaration of **multiset** template:

## App.cpp

```
template < class Key, class Compare = less<Key> > class multiset
{
    ...
}
```

# STL (associative containers - multiset)

- ▶ An example of using multiset:

## App.cpp

```
void main(void)
{
    multiset<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(5);
    s.insert(10);
    multiset<int>::iterator it;

    for (it = s.begin(); it != s.end(); it++)
        printf("%d ", *it);
}
```

## Output

5 10 10 20

# STL (associative containers - multiset)

- ▶ The methods supported by the **multiset** container:

Method/operator
Assignment ( <b>operator=</b> )
Insertion ( <b>insert</b> )
Deletion ( <b>erase</b> , <b>clear</b> methods)
Accessing elements ( <b>find</b> method)
Iterators ( <b>begin</b> , <b>end</b> , <b>rbegin</b> , <b>rend</b> , <b>cbegin</b> , <b>cend</b> , <b>crbegin</b> , <b>crend</b> (the last 4 from C++11) )
Informations ( <b>size</b> , <b>empty</b> , <b>max_size</b> methods)
Special methods ( <b>upper_bound</b> and <b>lower_bound</b> - to access the intervals in witch there are elements with the same key; <b>equal_range</b> - to obtain an interval that includes all the elements which have a specific key)

# STL (associative containers - unordered\_map)

- ▶ The elements from an **unordered\_map** container are stored into a hash-table
- ▶ Introduced in C++11
- ▶ For use “**#include <unordered\_map>**”
- ▶ Supports the same methods as **map** plus methods for the control of buckets.
- ▶ The declaration of **unordered\_map** template:

## App.cpp

```
template < class Key, class Value, class Hash, class Equal > class unordered_map
{
    ...
}
```

# STL (associative containers - unordered\_map)

- ▶ How hash tables works:

Popescu

Ionescu

Georgescu

Hash function(transforms  
a string into a index)

[illegible]

# STL (associative containers - unordered\_map)

- ▶ How hash tables works:

Popescu

Ionescu

Georgescu

We consider the following hashing function:

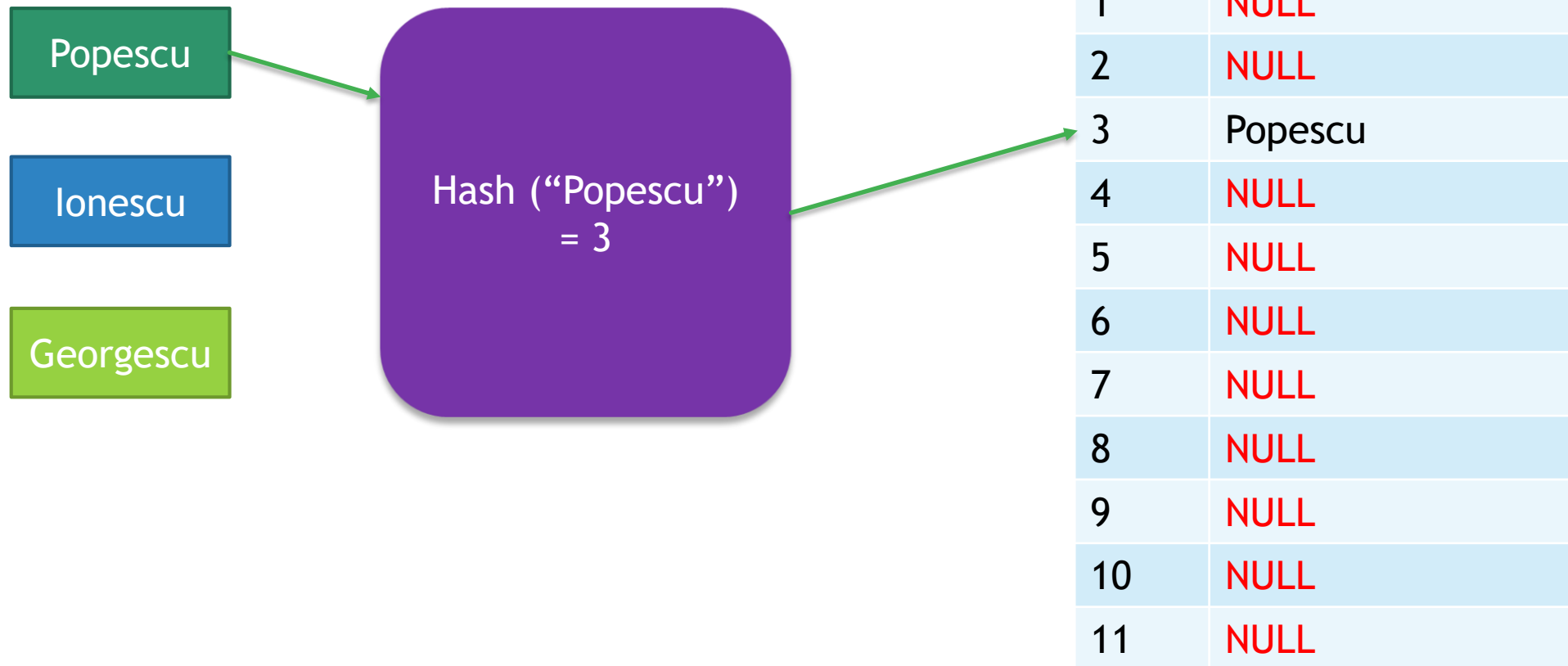
## HashFunction

```
int HashFunction(const char* s)
{
    int sum = 0;
    while ((*s) != 0)
    {
        sum += (*s);
        s++;
    }
    return sum % 12;
}
```

Index	Value
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

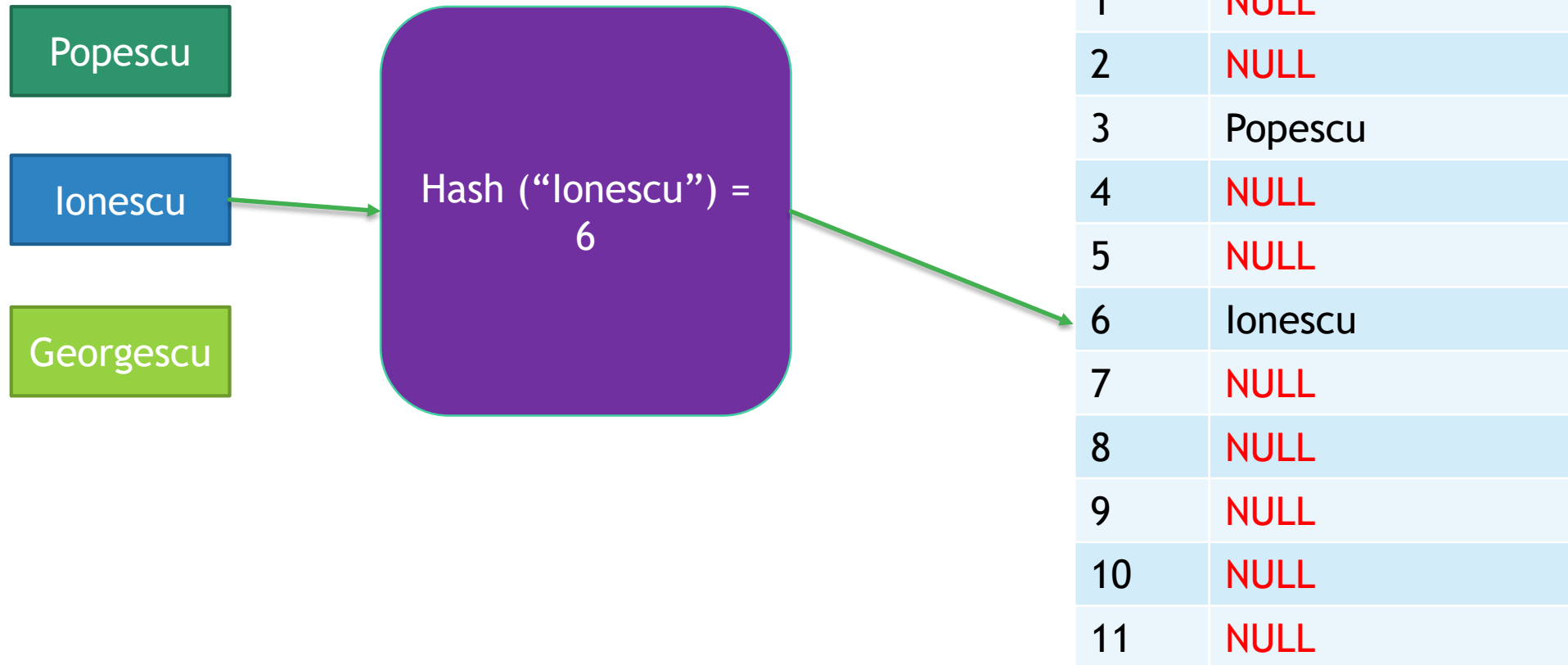
# STL (associative containers - unordered\_map)

- ▶ How hash tables works:



# STL (associative containers - unordered\_map)

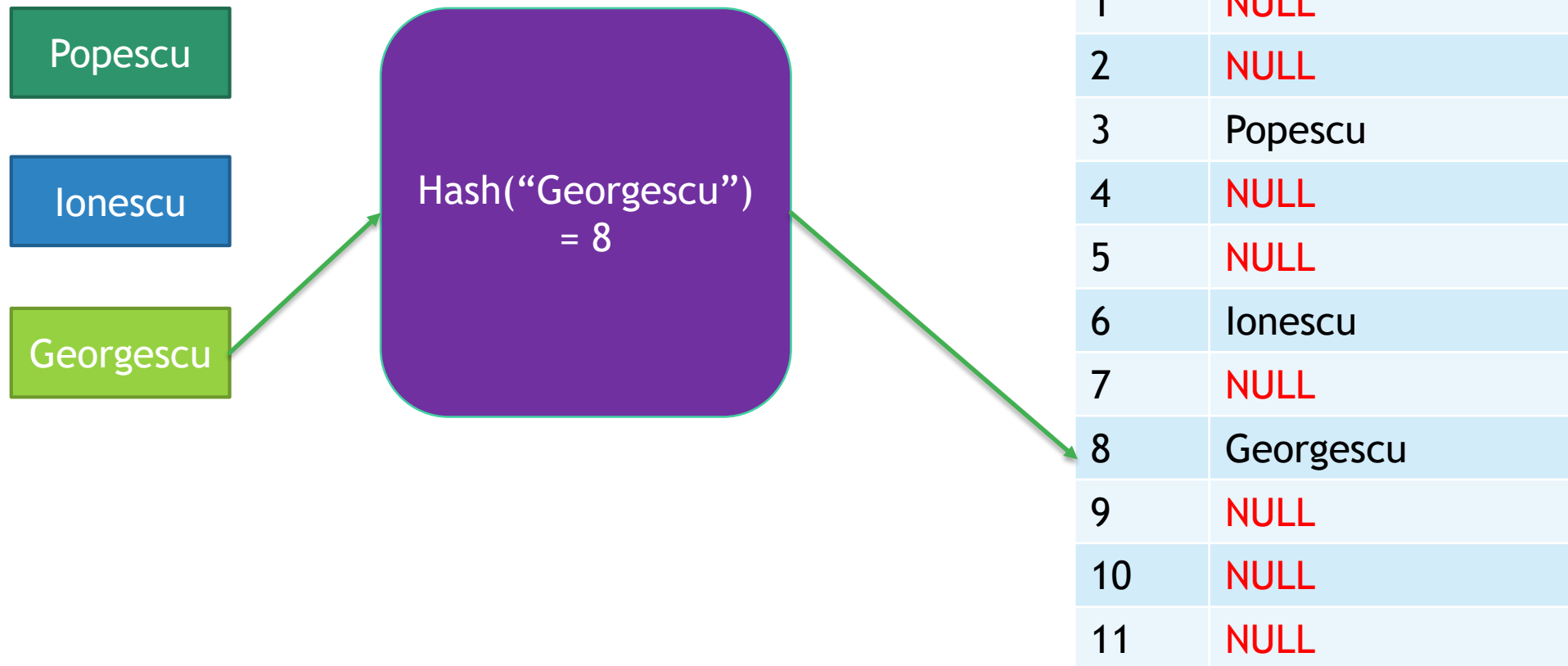
- ▶ How hash tables works:





# STL (associative containers - unordered\_map)

- ▶ How hash tables works:



# STL (associative containers - unordered\_map)

- ▶ Consider the following code:

## Umap-1.cpp

```
void main(void)
{
    unordered_map<int,int> Test;
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr]=tr;
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310

# STL (associative containers - unordered\_map)

- ▶ Consider the following code:

## Umap-2.cpp

```
void main(void)
{
    unordered_map<int,int> Test;
    Test.reserve(1000000);
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr]=tr;
}
```

- ▶ Let's remake the previous experiment :

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006

# STL (associative containers - unordered\_set)

- ▶ “**unordered\_set**” is similar with the **set** container, but the elements are not sorted
- ▶ Introduced in C++11
- ▶ For use “**#include <unordered\_set>**”
- ▶ Supports the same methods as **set** plus methods for the control of buckets
- ▶ The declaration of **unordered\_set** template:

## App.cpp

```
template < class Key, class Hash, class Equal > class unordered_set
{
    ...
}
```

# STL (associative containers - unordered\_set)

- ▶ Consider the following code:

## Uset-1.cpp

```
void main(void)
{
    unordered_set<int> s;
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	11862

# STL (associative containers - unordered\_set)

- And the variant with **reserve**:

## Uset-2.cpp

```
void main(void)
{
    unordered_set<int> s;
    s.reserve(1000000);
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	11862
Uset-2	6516	6328	6875	6844	6812	6453	6453	6531	6500	6515	6582

# STL (associative containers)

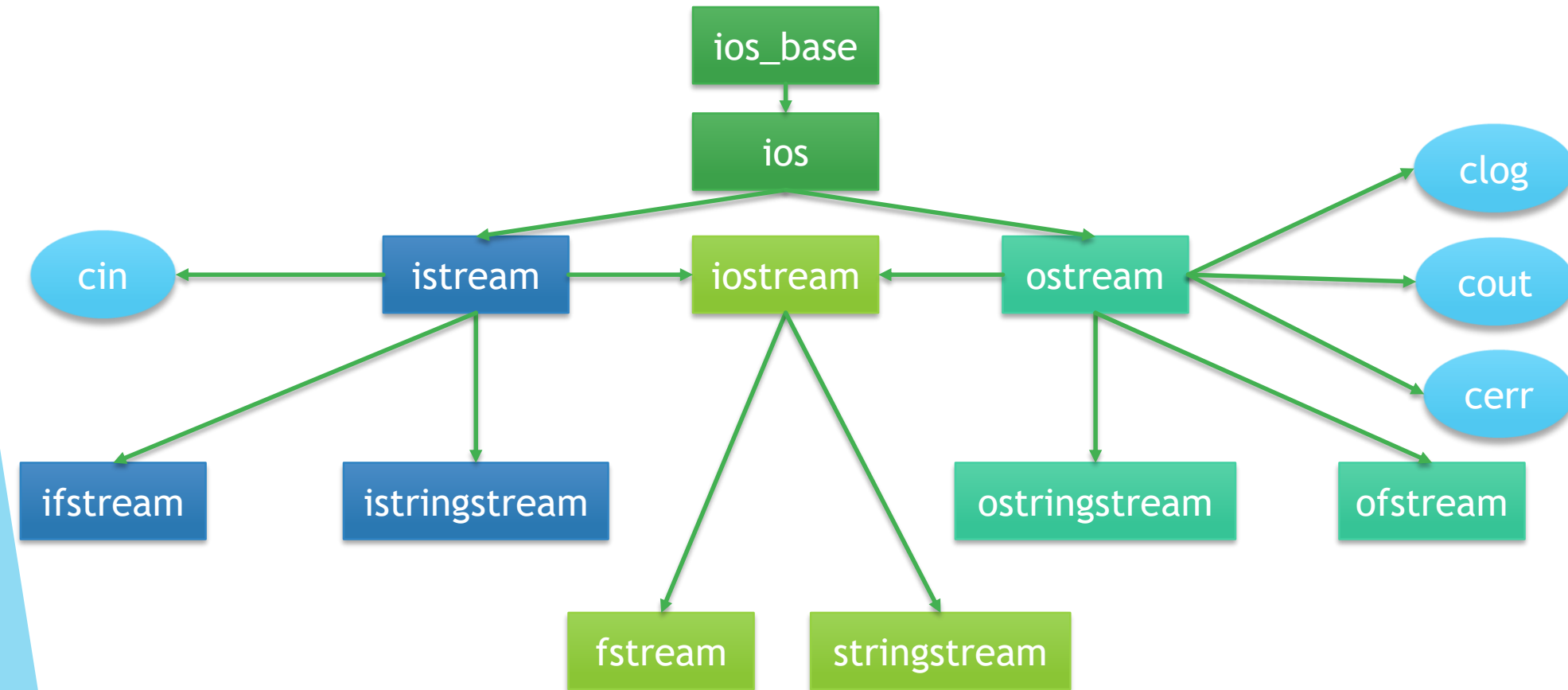
- ▶ Besides the presented containers, two other associative containers exist:
  - ▶ `unordered_multimap`
  - ▶ `unordered_multiset`
- ▶ Similar with `multimap`/`multiset` (the difference is that hash tables are used, not sorted trees)
- ▶ When choosing between **`map`** and **`unordered_map`** (**`set`** and **`unordered_set`**, ...), the amount of available memory and the execution times (insertion, deletion, access of elements, ...) must be considered.



## ► I/O Streams



# STL (IOS)



# 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



# ► 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
<code>static bool eq (CType c1, CType c2)</code>	Returns <b>true</b> if c1 is equal with c2
<code>static bool lt (CType c1, CType c2)</code>	Returns <b>true</b> if c1 is lower than c2
<code>static size_t length (const CType* s);</code>	Returns the size of a string
<code>static void assign (CType&amp; character, const CType&amp; value)</code>	Assignment (character = value)
<code>static int compare (const CType* s1,                     const CType* st2, size_t n);</code>	Compares two strings; returns 1 if s1 > s2, 0 if s1 = s2 and -1 if s1 < s2
<code>static const char_type* find (const char_type* s,                               size_t n,                               const char_type&amp; c);</code>	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
<pre>static char_type* move (char_type* dest,                         const char_type* src,                         size_t n);</pre>	Moves the content of a string between two locations
<pre>static char_type* copy(char_type* dest,                       const char_type* src,                       size_t n);</pre>	Copies the content of a string from one location to another
<pre>static int_type eof()</pre>	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 ( <b>operator=</b> )
Append ( <b>operator+=</b> and <b>append</b> , <b>push_back</b> methods)
Characters insertion ( <b>insert</b> method)
Access to characters ( <b>operator[]</b> and <b>at</b> , <b>front</b> (C++11) , <b>back</b> (C++11) methods)
Substrings ( <b>substr</b> method)
Replace ( <b>replace</b> method)
Characters deletion ( <b>erase</b> , <b>clear</b> , and <b>pop_back</b> (C++11) methods)
Search ( <b>find</b> , <b>rfind</b> , <b>find_first_of</b> , <b>find_last_of</b> , <b>find_first_not_of</b> , <b>find_last_not_of</b> methods)
Comparisons ( <b>operator&gt;</b> , <b>operator&lt;</b> , <b>operator==</b> , <b>operator!=</b> , <b>operator&gt;=</b> , <b>operator&lt;=</b> and <b>compare</b> method)
Iterators ( <b>begin</b> , <b>end</b> , <b>rbegin</b> , <b>rend</b> , <b>cbegin</b> , <b>cend</b> , <b>crbegin</b> , <b>crend</b> (the last four from C++11) )
Informations ( <b>size</b> , <b>length</b> , <b>empty</b> , <b>max_size</b> , <b>capacity</b> 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 !");
}
```



# ► 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 = { "P00", "C++" };
    vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
    map<const char*, int> st2 = { { "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 = { "P00", "C++" };
    vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
    map<const char*, int> st2 = { { "Popescu", 10 }, { "Ionescu", 9 } };
    map<const char*, int> st3 = { { "Popescu", 10 }, { "Popescu", 9 } };
}
```

In this case, "st3" will contain only one item (Popescu with the grade 10) the first one that was added. This is because the <map> template allows only one item for each key.

# 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(sprintf("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**