Download slides on
https://github.com/dginhac/esirem-itc313

ITC313 - Lesson 05

Fundamentals of programming
Learning C++: from beginner to beyond

Dominique Ginhac

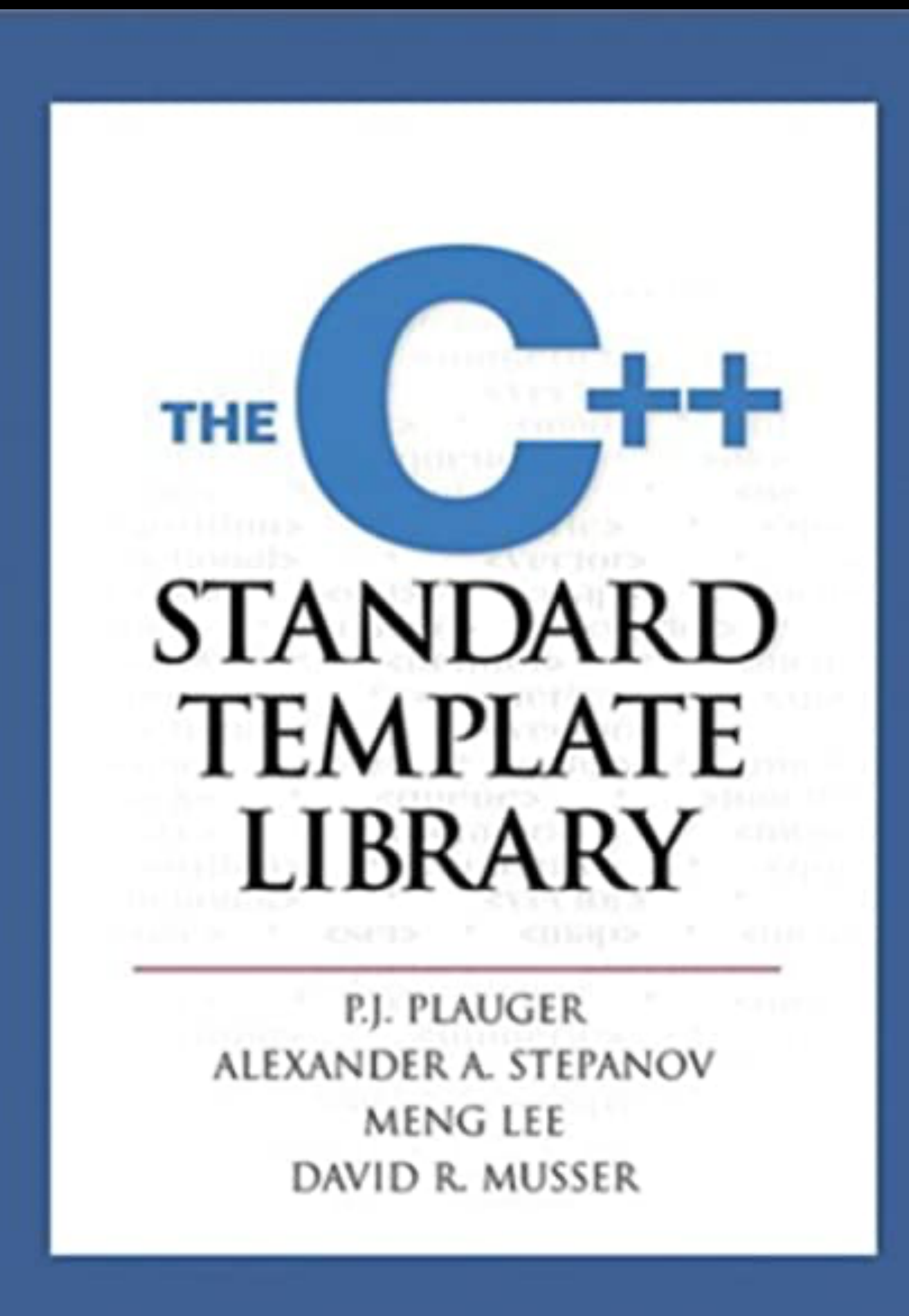
(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac

Lesson 05

STL Containers

In this lesson, students will learn about an important language component — arrays — and how to use them in their code
They will also learn how to use the most common STL algorithms to perform logic on the data containers.

What is the C++ STL?



The Standard Template Library (STL) is a set of C++ CLASSES that provide common programming data structures such as arrays, lists, stacks…

STL is a GENERIC LIBRARY that can be customized with any built-in or user-defined types.

STL has 4 COMPONENTS including Containers, Iterators, Algorithms and Functors

The STL Containers

STL Containers are C++ OBJECTS that store collections of elements (i.e. other objects)

Containers implement COMMON STRUCTURES such as arrays, queues, stacks, linked lists, trees, associatives sets, …

Containers manage the STORAGE SPACE and provide ACCESS to each element through iterators and member functions

Arrays in C++

An array is used to store a collection of data.

An array is a collection of variables of the same type that can be accessed through a single identifier.

In C++, there are three ways to use arrays:

C-style arrays	**std::array**	**std::vector**
Fixed arrays identical to C (Old School)	Built-in fixed arrays (STL Container) available since C++11	Built-in dynamic arrays (STL Container) available since C++11

See https://shendrick.net/Coding%20Tips/2015/03/15/cpparrayvsvector.html

C-style arrays

A C-style array can be constructed from any fundamental type.

```
type variable_name[SIZE] = {initial values};
```

int a[3];

2192	451	13918

int a[3]={1, 2, 3};

1	2	3

int a[3]={1, 1, 1};

1	1	1

int a[3]={ };

0	0	0

int a[3]={ 0 };

0	0	0

int a[3]={ 1 };

1	0	0

int a[3]={ [0...1]=3 };

3	3	0

int a[ ]={ [0...1]=3 };

3	3

int *a;
int* a;
int * a;
int*a;

See https://en.cppreference.com/w/cpp/language/array

Using C-style arrays

```
int main() {
    // hold the first 5 prime numbers
    int prime[5];
    // The first element has index 0
    prime[0] = 2;
    prime[1] = 3;
    prime[2] = 5;
    prime[3] = 7;
    prime[4] = 11; // The last element has index 4 (array length-1)
    std::cout << "The first prime number is: " << prime[0] << "\n";
    std::cout << "The sum of the first 5 primes is: " << prime[0] + prime[1]
+ prime[2] + prime[3] + prime[4] << "\n";
    return 0;
}
```

```
lesson05 — -zsh — 72×6
→  lesson05 clang++ -o c-style-arrays c-style-arrays.cpp
→  lesson05 ./c-style-arrays
The lowest prime number is: 2
The sum of the first 5 primes is: 28
→  lesson05
```

C-style arrays have a FIXED LENGTH known at compilation.
The length can't be changed or calculated at runtime.

Indexing out of range

```cpp
int main() {
    // hold the first 5 prime numbers
    int prime[5] = {2,3,5,7,11};
    // try to access outside the array
    prime[7] = 13;
    std::cout << prime[7] << std::endl;
    for (int i=0; i<10; i++) {
        std::cout << prime[i] << " ";
    }
    std::cout  << std::endl;
    return 0;
}
```

```
lesson05 — -zsh — 76×22
→ lesson05 clang++ c-style-arrays-out-of-range.cpp -o c-style-ar
-range
c-style-arrays-out-of-range.cpp:6:5: warning: array index 7 is pa
      of the array (which contains 5 elements) [-Warray-bounds]
    prime[7] = 13;
    ^     ~
c-style-arrays-out-of-range.cpp:4:5: note: array 'prime' declared
    int prime[5] = {2,3,5,7,11};
    ^
c-style-arrays-out-of-range.cpp:7:18: warning: array index 7 is p
      of the array (which contains 5 elements) [-Warray-bounds]
    std::cout << prime[7] << std::endl;
                 ^     ~
c-style-arrays-out-of-range.cpp:4:5: note: array 'prime' declared
    int prime[5] = {2,3,5,7,11};
    ^
2 warnings generated.
→ lesson05 ./c-style-arrays-out-of-range
13
2 3 5 7 11 32766 -1275526951 13 -473933336 32766
[1]    70310 abort      ./c-style-arrays-out-of-range
→ lesson05
```

Rule: When using C-style arrays, ALWAYS ENSURE that your indices are valid for the range of your array!

C-style arrays are deprecated

A C-Style array is just a "NAKED" array

C++ provides no element access operator, no capacity operator, no dedicated function.

So, any basic operation on a C-style array must be HAND-WRITTEN

```
/* File: lblutil.c - continued */
    /* Sorts an array of labels person[], of size n, by last name
       using an array of pointers plabel[]. */
    void sortlabels(struct label person[], struct label *plabel[], int n)
    {   int i;

        for (i = 0; i < n; i++)
            plabel[i] = person + i;
        sortptrs(plabel, n);
    }

    /* Sorts pointers to labels by last name */
    void sortptrs(struct label *plabel[], int n)
    {   int j, maxpos, eff_size;
        struct label *ptemp;

        for (eff_size = n; eff_size > 1; eff_size--) {
            maxpos = 0;
            for (j = 0; j < eff_size; j++)
                if (strcmp(plabel[j]->name.last,
                           plabel[maxpos]->name.last) > 0)
                    maxpos = j;
            ptemp = plabel[maxpos];
            plabel[maxpos] = plabel[eff_size-1];
            plabel[eff_size-1] = ptemp;
        }
    }
```

Figure 12.14: Utility Functions to Sort label Structures

std::array: a modern C-style array

std::array is a C++ container that encapsulates fixed size arrays

std::array combines the performance and accessibility of a C-style array with the benefits of a standard container.

```
#include <array>

std::array <type, size> variable_name = {initial values};
```

std::array: a modern C-style array

std::array is provided with different functions and facilities making its use easy and efficient:

- Array size is tracked (with C-style array, you need to manually track this)
- Bounds checking is provided
- Container operations such as sorting are allowed

Using std::array

```cpp
int main() {
    //Declares an array of 6 ints. Size is always required
    std::array<int, 6> array1;
    array1[0] = 1; array1[1] = 1; array1[2] = 2;
    array1.at(3) = 3; array1.at(4) = 5; array1.at(5) = 8;
    //Declare and initialize with an initializer list
    std::array<int, 5> array2 = {-2, -1, 0, 1, 2};
    // Access to elements
    std::cout << "Size of array1 = " << array1.size() << std::endl;
    std::cout << "array1 front = " << array1.front() << std::endl;
    std::cout << "array2 back = " << array2.back() << std::endl;
    // Fill all the elements with a value
    array2.fill(4);
    // C-Style For loop
    for (auto i=0;i<array2.size();i++)
        std::cout << array2[i] << " ";
    std::cout << std::endl;
    // C++ for loop
    for (auto number : array1)
        std::cout << number << " ";
    std::cout << std::endl;
    return 0;
}
```

lesson05 — -zsh — 65×9

```
➜  lesson05 clang++ std-arrays.cpp -o std-arrays -std=c++11
➜  lesson05 ./std-arrays
Size of array1 = 6
array1 front = 1
array2 back = 2
4 4 4 4 4
1 1 2 3 5 8
➜  lesson05 hk3
➜  lesson05
```

std::array out of range

```cpp
int main() {
    std::array<int, 5> array1 = {14,30,-23,15,-13};
    array1[5]=42;
    std::cout << "After Last : " << array1[5] << "\n";
    std::cout << "Still alive" << std::endl;

    array1.at(5) = 666;
    std::cout << "After Last : " << array1.at(5) << "\n";
    std::cout << "Dead" << std::endl;
  return 0;
}
```

.at() has the same behavior as the operator [] function

except that .at() checks the array bounds and signals whether the index is out of range by throwing an exception

lesson05 — -zsh — 80×9

```
→  lesson05 clang++ -std=c++11 std-arrays-out-of-range.cpp -o std-arrays-out-of-
range
→  lesson05 ./std-arrays-out-of-range
After Last : 42
Still alive
libc++abi.dylib: terminating with uncaught exception of type std::out_of_range:
array::at
[1]    61860 abort      ./std-arrays-out-of-range
→  lesson05
```

Exceptions

Upcoming lesson

Comparing arrays

```cpp
double array_mean(double means[6], int size) {
    double m =0.0;
    for (int i = 0; i < size; i++)
        m+= means[i];
    return m/size;
}
double stdarray_mean(std::array<double, 6> means) {
    double m=0.0;
    for (double value: means)
        m+=value;
    return m/means.size();
}

int main() {
    double a1[6] = {10, 8.5, 12, 16.5, 13.5, 18 };
    std::array<double, 6> a2 = {10, 8.5, 12, 16.5, 13.5, 18};
    double m1 = array_mean(a1,6);
    double m2 = stdarray_mean(a2);
    std::cout << m1 << std::endl;
    std::cout << m2 << std::endl;
    return 0;
}
```

lesson05 — -zsh — 80×6

```
→  lesson05 clang++ -std=c++11 arrays-and-functions.cpp
→  lesson05 ./arrays-and-functions
13.0833
13.0833
→  lesson05
```

std::array operations

Member functions

Implicitly-defined member functions

(constructor) (implicitly declared)	initializes the array following the rules of aggregate initialization (note that default initialization may result in indeterminate values for non-class T) (public member function)
(destructor) (implicitly declared)	destroys every element of the array (public member function)
operator= (implicitly declared)	overwrites every element of the array with the corresponding element of another array (public member function)

Element access

`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying array (public member function)

Iterators

`begin` `cbegin`	returns an iterator to the beginning (public member function)
`end` `cend`	returns an iterator to the end (public member function)
`rbegin` `crbegin`	returns a reverse iterator to the beginning (public member function)
`rend` `crend`	returns a reverse iterator to the end (public member function)

Capacity

`empty`	checks whether the container is empty (public member function)
`size`	returns the number of elements (public member function)
`max_size`	returns the maximum possible number of elements (public member function)

Operations

`fill`	fill the container with specified value (public member function)
`swap`	swaps the contents (public member function)

Non-member functions

`operator==` `operator!=` (removed in C++20) `operator<` (removed in C++20) `operator<=` (removed in C++20) `operator>` (removed in C++20) `operator>=` (removed in C++20) `operator<=>` (C++20)	lexicographically compares the values in the array (function template)
`std::get`(std::array)	accesses an element of an `array` (function template)
`std::swap`(std::array) (C++11)	specializes the `std::swap` algorithm (function template)
`to_array` (C++20)	creates a `std::array` object from a built-in array (function template)

See https://en.cppreference.com/w/cpp/container/array

std::vector: a modern dynamic array

std::vector is a sequence container representing an array that can change in size.

std::vector provides DYNAMIC ARRAY functionality.

You can create arrays that have their length set at runtime, without having to explicitly allocate and deallocate memory.

```
#include <vector>

std::vector <type> variable_name = {initial values};
```

std::vector: a modern dynamic array

The std::vector class simplifies operations which are cumbersome with C-style dynamic arrays.

- As for the std::array, current array size is tracked
- Current amount of allocated memory is tracked (With C-style arrays, you would need to manually track this)
- Growing and shrinking the array is a function call
- Inserting elements into the middle of an array is simplified into a function call (C-style arrays require manually moving memory around when inserting elements into the middle of the array)

Using std::vector

```cpp
int main() {
    // Dynamic array allocated on the heap
    std::vector<int> vect1;
    vect1.assign(3, 666);
    std::cout << "Size = " << vect1.size() << std::endl;
    std::cout << vect1;
    vect1.push_back(42);
    vect1.push_back(1337);
    vect1.push_back(314);
    std::cout << "Size = " << vect1.size() << std::endl;
    std::cout << vect1;
    vect1[2]= 707; vect1.at(3) = 0;
    std::cout << vect1;
    vect1.pop_back(); vect1.pop_back();
    vect1.resize(8,101);
    std::cout << vect1;
    vect1.resize(2);
    std::cout << vect1;
    vect1.clear();
    return 0;
}
```

lesson05 — -zsh — 80×10

```
→  lesson05 clang++ -std=c++11 std-vector.cpp -o std-vector
→  lesson05 ./std-vector
Size = 3
666 666 666
Size = 6
666 666 666 42 1337 314
666 666 707 0 1337 314
666 666 707 0 101 101 101 101
666 666
→  lesson05
```

std::vector out of range

```cpp
int main() {
   // Dynamic array allocated on the heap
   std::vector<int> vect1 = {14,30,-23,15,-13};
   std::cout << "Using index" << std::endl;
    vect1[6] = 666;
   std::cout << "After Last : " << vect1[6] << std::endl ;
   std::cout << std::endl;
   std::cout << "Using method at" << std::endl;
   std::cout << "After Last : " << vect1.at(6) << std::endl ;
   return 0;
}
```

```
lesson05 — -zsh — 86×9
→  lesson05 clang++ -std=c++11 std-vector-out-of-range.cpp -o std-vector-out-of-range
→  lesson05 ./std-vector-out-of-range
Using index
After Last : 666

Using method at
libc++abi.dylib: terminating with uncaught exception of type std::out_of_range: vector
After Last : [1]    85327 abort      ./std-vector-out-of-range
→  lesson05 
```

same behavior
as the std:array
with the generation
of exceptions

std::vector: operations

Same operators as std::array + extra operators for managing the ability to have a dynamic storage size.

Modifiers	
clear	clears the contents (public member function)
insert	inserts elements (public member function)
emplace (C++11)	constructs element in-place (public member function)
erase	erases elements (public member function)
push_back	adds an element to the end (public member function)
emplace_back (C++11)	constructs an element in-place at the end (public member function)
pop_back	removes the last element (public member function)
resize	changes the number of elements stored (public member function)
swap	swaps the contents (public member function)

Capacity	
empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)
reserve	reserves storage (public member function)
capacity	returns the number of elements that can be held in currently allocated storage (public member function)
shrink_to_fit (C++11)	reduces memory usage by freeing unused memory (public member function)

See https://en.cppreference.com/w/cpp/container/array

A real-life case study

?

What type of arrays will be adequate for our todos?

C-style ? std::array ? std::vector ?

A real-life case study

```
define MAXSIZE 1000
class Todos {
 private:
     Todo _todos[MAXSIZE];
};
```

```
define MAXSIZE 1000
class Todos {
 private:
    std::array<Todo,MAXSIZE> _todos;
};
```

```
class Todos {
 private:
   std::vector<Todos> _todos;
};
```

What is the best solution

A real-life case study

```
define MAXSIZE 1000
class Todos {
 private:
     Todo _todos[MAXSIZE];
};
```

```
define MAXSIZE 1000
class Todos {
 private:
     std::array<Todo,MAXSIZE> _todos;
};
```

C-STYLE array is a VERY BAD choice because of the fixed size of the todo list and also because no operator is provided

Using STD::ARRAY is NOT OPTIMAL because of the fixed size of the todo list

```
class Todos {
 private:
    std::vector<Todos> _todos;
};
```

STD::VECTOR is the BEST CHOICE because of the automatic management of the todo list when adding/removing a todo

todos.h

The class Todos includes :

- A basic constructor for building empty objects (optional)
- A minimal set of public methods
- A vector of Todo as variable member

Todos objects use Date and Todo class

See todos.h, todos.cpp and todos-main.cpp for details.

```cpp
#include <vector>
#include "todo.h"

namespace todo {
    class Todos {
    public:
        Todos();
        void add(Todo todo);
        void del(int id);
        friend std::ostream& operator<<(std::ostream& os, const Todos& todos);
    private:
        std::vector<Todo> _todos;
    };
} // todo
```

todos.cpp

```
#include "todos.h"
namespace todo {
    Todos::Todos() {
    }
    void Todos::add(Todo todo) {
        _todos.push_back(todo);
    }
    void Todos::del(int id) {
        if (id < _todos.size()) {
            _todos.erase(_todos.begin()+id);
        }
    }
    std::ostream& operator<<(std::ostream& os, const Todos& todos) {
        for (const Todo todo: todos._todos)  {
            os << todo;
        }
        return os;
    }
} // todo
```

As a friend function, operator<< can access to private variables such as _todos

os << todo requires that operator << is also overloaded for Todo class

Using todos

```cpp
#include "todos.h"
int main() {
  todo::Todos todos;
  todo::Todo todo1("Buy beer", Category::Personal, NORMAL, date::Date(2020,11,5));
  todos.add(todo1);
  todo::Todo todo2("Play piano", Category::Personal, NORMAL, date::Date(2020,11,10));
  todos.add(todo2);
  todo::Todo todo3("Write report", Category::Work, HIGH, date::Date(2020,11,1));
  todos.add(todo3);
  todo::Todo todo4("Prepare keynote", Category::Work, LOW, date::Date(2020,12,20));
  todos.add(todo4);
  std::cout << todos << "\n";
  todos.del(2);
  std::cout << todos;
  return 0;
}
```

```
lesson05 — -zsh — 84×11
→ lesson05 clang++ -std=c++11 date.cpp todo.cpp todos.cpp todos-main.cpp -o todos
→ lesson05 ./todos
TODO: Buy beer (2020/Nov/5)
TODO: Play piano (2020/Nov/10)
TODO: Write report (2020/Nov/1)
TODO: Prepare keynote (2020/Dec/20)

TODO: Buy beer (2020/Nov/5)
TODO: Play piano (2020/Nov/10)
TODO: Prepare keynote (2020/Dec/20)
→ lesson05
```

Containers: Iterator concept

Iterating through an array of data is quite a common thing to do. We've already covered index-based loops (for-loops and while loops) and range-based for-loops for std::arrays or std::vector

```
int main() {
    int data1[5] = {0,2,4,6,8};
    std::array<int,5> data2 = {1,3,5,7,9};
    std::vector<int> data3 = {10,11,12,13,14,15};
    for (int i=0; i<5; i++) {
        std::cout << data1[i] << " " << data2.at(i) << " ";
    }
    std::cout << std::endl;
    for (int d: data3) {
        std::cout << d << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```
lesson05 — -zsh — 84×6
→  lesson05 clang++ -std=c++11 basic-iterators.cpp -o basic-iterators
→  lesson05 ./basic-iterators
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15
→  lesson05 hk3
→  lesson05
```

C++ Iterators

C++ Iterators are used to step through the elements of collections of objects.

C++ iterators offer common interfaces for any container type (array, vector, list, queue, stack, …) including the following basic operations :

- operator= : assigns an iterator to a specific position
- operator* : returns the element of the current position of the iterator
- operator++ / operator --: step forward to the next element / step backward to the previous element
- operator== and != : check whether 2 iterators represent or not the same position

C++ Iterators syntax

```
Container-type<type>::iterator iterator_name = position;

  std::array<int,5> data = {1,3,5,7,9};
  std::array<int,5>::iterator it = data.begin();
```

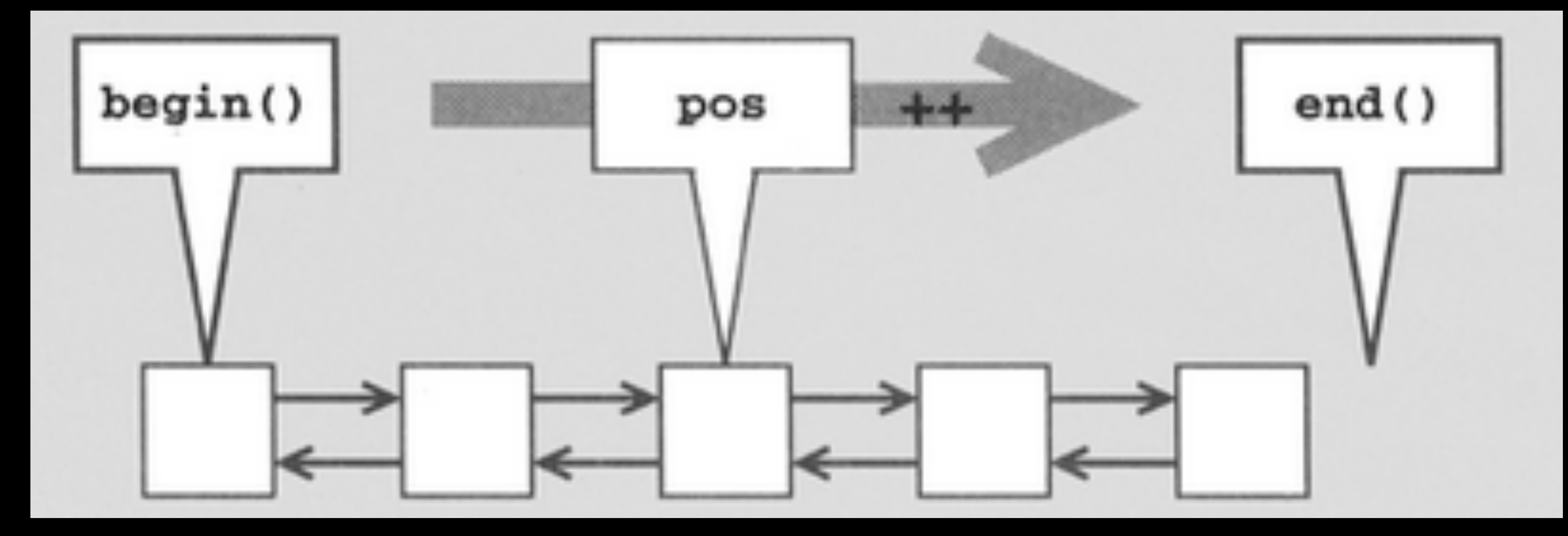
Specifying the data type for an iterator is a bit lengthy

So use the AUTO keyword to declare and initialize an iterator in one statement

```
  auto it1 = data.begin();
  auto it2 = data.end();
```

begin() is on the first element

end() is after the last element



First ex. of C++ iterators

```
int main() {
    std::vector<int> data = {1,3,5,7,9};
    for (int d: data)
        std::cout << d << " ";
    std::cout << std::endl;

    int i=0;
    while (i<data.size()) {
        std::cout << data.at(i) << " ";
        i++;
    }
    std::cout << std::endl;

    auto it = data.begin();
    while (it != data.end()) {
        std::cout << *it << " ";
        ++it;
    }
    std::cout << std::endl;
    return 0;
}
```

When using iterators, ++it or it++ do the same job but prefer using ++it because the postfix operation implies first a useless copy of the unincremented iterator (see Lesson 04 about overloading ++ and -- operators)

```
lesson05 — -zsh — 73×6
→ lesson05 clang++ -std=c++11 basic-iterators2.cpp -o basic-iterators2
→ lesson05 ./basic-iterators2
1 3 5 7 9
1 3 5 7 9
1 3 5 7 9
→ lesson05
```

Other ex. of C++ iterators

```
int main() {
    std::vector<int> data = {1,3,5,7,9};
    int i=data.size();
    while (i>0) {
        i--;
        std::cout << data.at(i) << " ";
    }
    std::cout << std::endl;
    auto it2 = data.end();
    while (it2 != data.begin()) {
        --it2;
        std::cout << *it2 << " ";
    }
    std::cout << std::endl;
    auto it3 = data.rbegin();
    while (it3 != data.rend()) {
        std::cout << *it3 << " ";
        ++it3;
    }
    std::cout << std::endl;
    return 0;
}
```

When iterating from the end to the begin, do not use standard iterators with decrementation (--) operator.

Prefer using reverse iterators rbegin() et rend() with ++ incrementation

```
lesson05 — -zsh — 73×6
→  lesson05 clang++ -std=c++11 basic-iterators3.cpp -o basic-iterators3
→  lesson05 ./basic-iterators3
9 7 5 3 1
9 7 5 3 1
9 7 5 3 1
→  lesson05
```

STL algorithms

Many programming tasks fall into basic actions such as sum, count, find, sort

These are all actions that are performed on sequences

The goal of the STL algorithms is to define these actions in a generic way, using small, reusable functions that avoid writing repetitive code and define a consistent, portable interface

STL include more than 150 algorithms for searching, counting, and manipulating ranges

See https://en.cppreference.com/w/cpp/algorithm

See https://www.youtube.com/watch?v=2olsGf6JIkU

See https://medium.com/logicalbee/c-stl-algorithms-cheat-sheet-d92f986abe14

The advantages of using algorithms

Algorithms bring EXPRESSIVENESS: by raising the level of abstraction of code. Algorithms show what they do, rather than how they are implemented

Algorithms avoid some COMMON MISTAKES: when using loops, you always need to make sure that you stop at the right step, and it behaves correctly when there is no element to iterate over. Algorithms deal with these for you

Algorithms provide a high level of QUALITY, the best algorithmic COMPLEXITY and the highest level of PERFORMANCE you can get

See https://www.fluentcpp.com/2017/01/05/the-importance-of-knowing-stl-algorithms/

How to copy container data

```
int main(int argc, char const *argv[]) {
    std::vector<int> primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    std::vector<int> primes_copy, primes_copy2, primes_copy3;

    // Only works for basic data
    // May need copy constructor
    primes_copy = primes;
    std::cout << primes_copy;

    // loop based copy
    for (auto it = primes.begin(); it!= primes.end(); ++it)
        primes_copy2.push_back(*it);
    std::cout << primes_copy2;

    // using copy algorithm
    std::copy(primes.begin(), primes.end(), std::back_inserter(primes_copy3));
    std::cout << primes_copy3;
    return 0;
}
```

```
lesson05 — -zsh — 49×6
→  lesson05 clang++ -std=c++11 copy.cpp -o copy
→  lesson05 ./copy
2 3 5 7 11 13 17 19 23 29
2 3 5 7 11 13 17 19 23 29
2 3 5 7 11 13 17 19 23 29
→  lesson05
```

How to subset container data

```
int main() {
    std::vector<int> data = {1,3,5,7,9,11,13,15,17,19};
    std::cout << data;
    // loop based
    std::vector<int> sub_data1;
    for (auto it=data.begin()+2; it!= data.begin()+8;++it)
        sub_data1.push_back(*it);
    std::cout <<sub_data1;
    // copy based
    std::vector<int> sub_data2;
    std::copy(data.begin()+2, data.begin()+8, std::back_inserter(sub_data2));
    std::cout << sub_data2;
    // range constructor based
    std::vector<int> sub_data3 = {data.begin()+2, data.begin()+8};
    std::cout << sub_data3;
    return 0;
}
```



```
→  lesson05 clang++ -std=c++11 range.cpp -o range
→  lesson05 ./range
1 3 5 7 9 11 13 15 17 19
5 7 9 11 13 15
5 7 9 11 13 15
5 7 9 11 13 15
→  lesson05
```

How to sort container data

```
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    std::sort(numbers.begin(),numbers.end());
    std::cout << numbers;
    std::reverse(numbers.begin(), numbers.end());
    std::cout << numbers;

    std::vector<std::string> words = {"Hello", "word", "The", "answer", "is",
"42"};
    std::cout << words;
    std::sort(words.begin(),words.end());
    std::cout << words;
    std::reverse(words.begin(), words.end());
    std::cout << words;
    return 0;
}
```

```
lesson05 — -zsh — 49×9
→  lesson05 clang++ -std=c++11 sort.cpp -o sort
→  lesson05 ./sort
0 -4 2 10 1 -5 6 -7 12 3
-7 -5 -4 0 1 2 3 6 10 12
12 10 6 3 2 1 0 -4 -5 -7
Hello word The answer is 42
42 Hello The answer is word
word is answer The Hello 42
→  lesson05
```

How to remove data

```cpp
int main() {
    std::vector<int> numbers = {0,10,20,10,10,30,40,10,-10,0};
    std::cout << numbers.size() << std::endl;
    std::cout << numbers;

    // remove all the numbers equal to 10
    // and returns an iterator on the last element
    auto it = std::remove(numbers.begin(),numbers.end(), 10);
    // numbers still have 10 elements
    std::cout << numbers;
    // Need to delete the last 4 elements using the member function erase
    numbers.erase(it, numbers.end());
    std::cout << numbers;
    return 0;
}
```

lesson05 — -zsh — 55×7

```
→  lesson05 clang++ -std=c++11 remove.cpp -o remove
→  lesson05 ./remove
10
0 10 20 10 10 30 40 10 -10 0
0 20 30 40 -10 0 40 10 -10 0
0 20 30 40 -10 0
→  lesson05
```

How to find an element

```
int main() {
    std::vector<int> numbers = {0,10,20,42,10,30,42,42,-10,42,0};
    std::cout << numbers;
    auto it = find(numbers.begin(), numbers.end(), 42);
    if (it != numbers.end()) {
        int index = std::distance(numbers.begin(), it);
        std::cout << "First Found at position " << index  << std::endl;
    }
    else
        std::cout << "Not found" << std::endl;

    int nb = std::count(numbers.begin(), numbers.end(), 42);
    if (nb) {
        std::cout << "Found " << nb << " times" << std::endl;
    }
    else
        std::cout << "Not found" << std::endl;
    return 0;
}
```

lesson05 — -zsh — 55×6

```
→  lesson05 clang++ -std=c++11 find.cpp -o find
→  lesson05 ./find
0 10 20 42 10 30 42 42 -10 42 0
First Found at position 3
Found 4 times
→  lesson05
```

How to find multiple values

```
int main() {
    std::vector<int> numbers = {0,10,20,42,10,30,42,42,-10,42,0};
    std::cout << numbers;
    auto it = numbers.begin();
    while (it != numbers.end()) {
        it = find(it, numbers.end(), 42);
        if (it != numbers.end()) {
            int index = std::distance(numbers.begin(), it);
            std::cout << "Found at position " << index  << std::endl;
            ++it;
         }
         else
            std::cout << "Not found" << std::endl;
    }
    return 0;
}
```

```
lesson05 — -zsh — 69×9
➜  lesson05 clang++ -std=c++11 multiple-find.cpp -o multiple-find
➜  lesson05 ./multiple-find
0 10 20 42 10 30 42 42 -10 42 0
Found at position 3
Found at position 6
Found at position 7
Found at position 9
Not found
➜  lesson05
```

How to sort data

It's easy to sort numbers or strings

```
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::sort(numbers.begin(),numbers.end());
    std::vector<std::string> words = {"Hello", "word", "The", "answer", "is",
"42"};
    std::sort(words.begin(),words.end());
    return 0;
}
```

The comparison used is operator< by default

But you can provide a custom comparator to sort on a different order

Sort using a custom function

std::sort can take a custom comparison function as 3rd parameter

The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1 &a, const Type2 &b);
```

```
bool less(const int a, const int b) {
    return a < b;
}
bool greater(const int a, const int b) {
    return a > b;
}
```

Sort using a custom function

```
bool less(const int a, const int b) {
    return a < b;
}
bool greater(const int a, const int b) {
    return a > b;
}

int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    // sort using a custom function object
    std::sort(numbers.begin(),numbers.end(), less);
    std::cout << numbers;
    std::sort(numbers.begin(),numbers.end(), greater);
    std::cout << numbers;
    return 0;
}
```

lesson05 — -zsh — 90×6

```
→ lesson05 clang++ -std=c++11 sort-with-custom-function.cpp -o sort-with-custom-function
→ lesson05 ./sort-with-custom-function
0 -4 2 10 1 -5 6 -7 12 3
-7 -5 -4 0 1 2 3 6 10 12
12 10 6 3 2 1 0 -4 -5 -7
→ lesson05
```

Sort using functors

A functor (or function object) is a user-defined C++ class or struct

A functor must overload the operator()

A functor is called like an ordinary function

A functor has advantages over standard function because, being objects, they can have properties and maintain state

Using a functor

```
int my_add_function (int x, int y) { return x + y; }

class MyAddFunctor {
public:
    MyAddFunctor(int x): _x(x) {}
    int operator() (const int value) {
        return _x + value ;
    }
private:
    int _x;
};
int main() {
    std::cout << my_add_function(10,5) << std::endl;
    MyAddFunctor add5(5); // create object
    std::cout << add5(10) << std::endl; // call operator()
    MyAddFunctor add42(42);
    std::cout << add42(10) << std::endl; // call operator()
    return 0;
}
```

lesson05 — -zsh — 59×6

```
→  lesson05 clang++ -std=c++11 functor.cpp -o functor
→  lesson05 ./functor
15
15
52
→  lesson05
```

Using a functor on a vector

```
class MyAddFunctor {
public:
    MyAddFunctor(int x): _x(x) {}
    int operator() (const int value) {
        return _x + value ; }
private:
    int _x;
};

int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    std::transform(numbers.begin(), numbers.end(),
                   numbers.begin(), MyAddFunctor(5));
    std::cout << numbers;
    return 0;
}
```

```
lesson05 — -zsh — 78×5
→  lesson05 clang++ -std=c++11 functor-with-vector.cpp -o functor-with-vector
→  lesson05 ./functor-with-vector
0 -4 2 10 1 -5 6 -7 12 3
5 1 7 15 6 0 11 -2 17 8
→  lesson05
```

Sort using a functor

```
struct Less {
    bool operator() (const int a, const int b) { return a < b;}
};
class Greater {
public:
    bool operator() (const int a, const int b) { return a > b;}
};
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    std::sort(numbers.begin(),numbers.end(), Less());
    std::cout << numbers;
    std::sort(numbers.begin(),numbers.end(), Greater());
    std::cout << numbers;
    return 0;
}
```

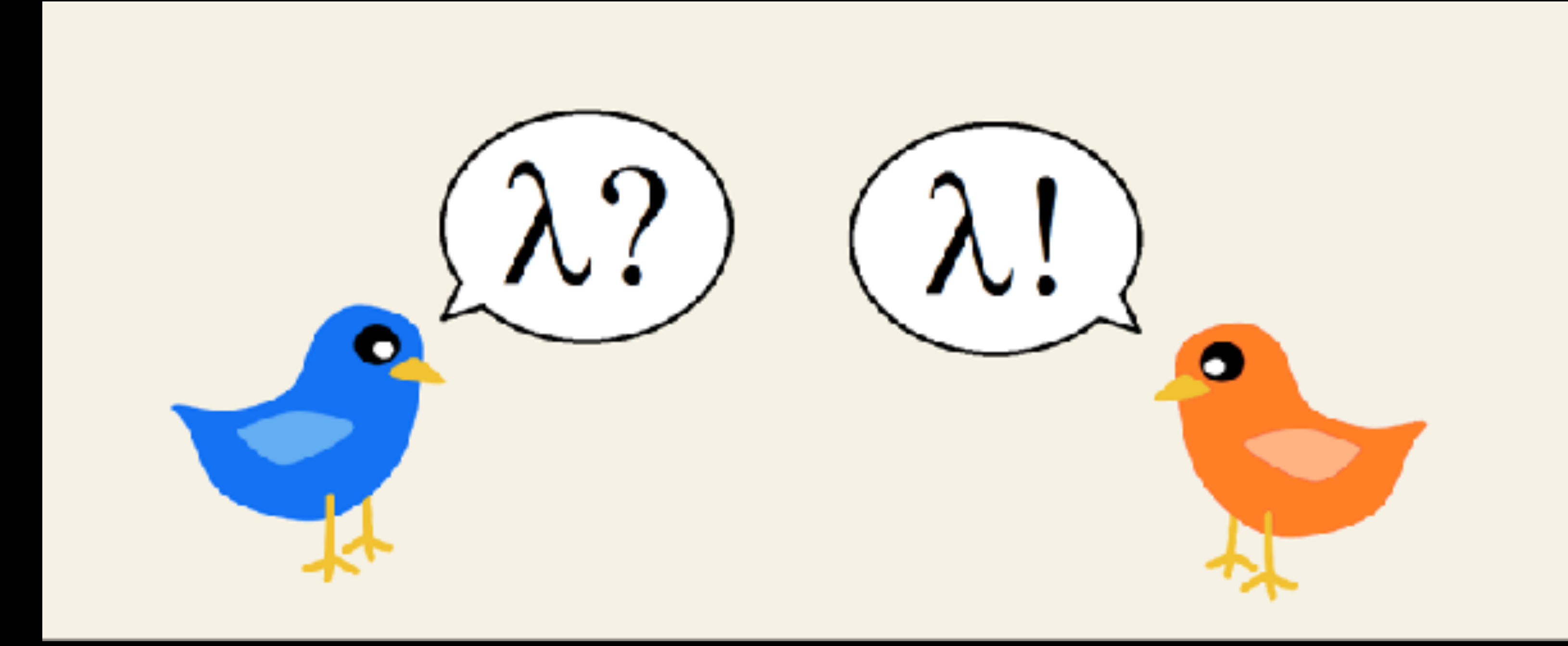
lesson05 — -zsh — 78×6

```
→  lesson05 clang++ -std=c++11 sort-with-functor.cpp -o sort-with-functor
→  lesson05 ./sort-with-functor
0 -4 2 10 1 -5 6 -7 12 3
-7 -5 -4 0 1 2 3 6 10 12
12 10 6 3 2 1 0 -4 -5 -7
→  lesson05
```

Sort using Lambda expressions

A lambda expression is a small snippet of code that provide a better readability than a custom function or a functor

A lambda expression is mainly used when you just want to define a short code once and use it on the fly




```
[captures] (params) -> returnType { function body }
```

With captures: the outside variables accessible from within the lambda body

params: the list of parameters as in functions

returnType: the return type of the lambda (optional, often omitted)

function body: the lambda body

Lambda expressions with no capture

```
class IsBetweenZeroAndTen {
  public:
  bool operator()(const int value) {
    return 0 < value && value < 10;
  }
};

int main() {
    IsBetweenZeroAndTen functor;
    std::cout << functor(4) << " " << functor(-5) << std::endl;

    //auto lambda = [] (int value) -> bool { return 0 < value && value < 10;};
    auto lambda = [] (int value) { return 0 < value && value < 10;};
    std::cout << lambda(4) << " " << lambda(-5) << std::endl;
    return 0;
}
```

Using a lambda expression is like using a functor, but without the need to create a named object.

```
lesson05 — -zsh — 61×5
→  lesson05 clang++ -std=c++11 lambda.cpp -o lambda
→  lesson05 ./lambda
1 0
1 0
→  lesson05
```

Lambda expressions with capture

A Lambda expression is considered as a closure when there is something in the capture clause ([])

```
int main() {
    int upper = 42;
    //Compilation error because upper is not in the lambda scope
    //auto no_capture = [] (int value) { return 0 < value && value < upper;};
    auto no_capture = [] (int value, int max) { return 0 < value && value <
max;};
    auto capture = [upper] (int value) { return 0 < value && value < upper;};

    std::cout << no_capture(4, 42) << std::endl;
    std::cout << capture(43) << std::endl;
    return 0;
}
```

```
lesson05 — -zsh — 69×5
→ lesson05 clang++ -std=c++11 lambda-capture.cpp -o lambda-capture
→ lesson05 ./lambda-capture
1
0
→ lesson05
```

Using a lambda on a vector

```
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    std::transform(numbers.begin(), numbers.end(), numbers.begin(),
        [](const int number) {return number*2;});
    std::cout << numbers;

    int limit=4;
    std::transform(numbers.begin(), numbers.end(), numbers.begin(),
        [limit](const int number) { return number<limit ?  number*2 : number
*3;});
    std::cout << numbers;
    return 0;
}
```

lesson05 — -zsh — 76×6

```
→  lesson05 clang++ -std=c++11 lambda-with-vector.cpp -o lambda-with-vector
→  lesson05 ./lambda-with-vector
0 -4 2 10 1 -5 6 -7 12 3
0 -8 4 20 2 -10 12 -14 24 6
0 -16 12 60 4 -20 36 -28 72 18
→  lesson05
```

Sort using Lambda expressions

Sort can be done using a custom function, a functor but also a lambda

```
class Greater {
public:
    bool operator() (const int a, const int b) { return a > b;}
};
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << numbers;
    // sort using a functor
    std::sort(numbers.begin(),numbers.end(), Greater());
    std::cout << numbers;
    std::sort(numbers.begin(),numbers.end(), [](int a, int b) {return a<b;});
    std::cout << numbers;
    return 0;
}
```

```
lesson05 — -zsh — 73×6
→  lesson05 clang++ -std=c++11 sort-with-lambda.cpp -o sort-with-lambda
→  lesson05 ./sort-with-lambda
0 -4 2 10 1 -5 6 -7 12 3
12 10 6 3 2 1 0 -4 -5 -7
-7 -5 -4 0 1 2 3 6 10 12
→  lesson05
```

A real-life case study

?

How to sort the todos in our todo list ?

How to find specific todos ?

todos.h

```
#include <vector>
#include "todo.h"

namespace todo {
    class Todos {
    public:
        Todos();
        void addTodo(Todo todo);
        void delTodo(int id);
        void sort();
        Todo next() const;
        Todos find(int priority) const;
        friend std::ostream& operator<<(std::ostream& os, const Todos& todos);
    private:
        std::vector<Todo> _todos;
    };
} // todo
```

Adds three new member functions to Todos class

todos.cpp

```cpp
#include "todos.h"
namespace todo {
    void Todos::sort() {
        std::sort(_todos.begin(),_todos.end(),
            [](Todo const& t1, Todo const& t2) {
                return t1.dueDate() < t2.dueDate();
            }
        );
    }

    Todo Todos::next() const {
        assert (_todos.empty() == false && "No todo");
        auto it = _todos.begin();
        return *it;
    }
} // todo
```

The sort function uses std::sort with a lambda expression to compare the due dates of the todos

The next function returns the next todo if it exists

todos.cpp

```
#include "todos.h"
namespace todo {
    Todos Todos::find(int priority) const {
        auto it = _todos.begin();
        Todos results;
        while (it != _todos.end()) {
            it = find_if(it, _todos.end(),
                        [priority](const Todo& obj) {
                            return obj.priority()==priority;
                        });
            if (it!= _todos.end()) {
                results._todos.push_back(*it);
                ++it;
            }
        }
        return results;
    }
} // todo
```

The find function uses std::find_if with a lambda expression to compare the priority of each todo with the priority passed as parameter

Using todos

```
int main() {
    todo::Todos todos;
    todo::Todo todo1("Buy beer", Category::Personal, NORMAL, date::Date(2020,11,5));
    todos.add(todo1);
    todo::Todo todo2("Play piano", Category::Personal, HIGH, date::Date(2020,11,10));
    todos.add(todo2);
    todo::Todo todo3("Write report", Category::Work, NORMAL, date::Date(2020,11,1));
    todos.add(todo3);
    todo::Todo todo4("Prepare keynote", Category::Work, LOW, date::Date(2020,12,20));
    todos.add(todo4);
    std::cout << todos << "\n";
    todo::Todos normal_todos = todos.find(NORMAL);
    normal_todos.sort();
    std::cout << normal_todos << "\n";
    std::cout << "Next NORMAL todo " << "\n";
    std::cout << normal_todos.next() << std::endl;
    todos.sort();
    std::cout << todos << "\n";
    return 0;
}
```

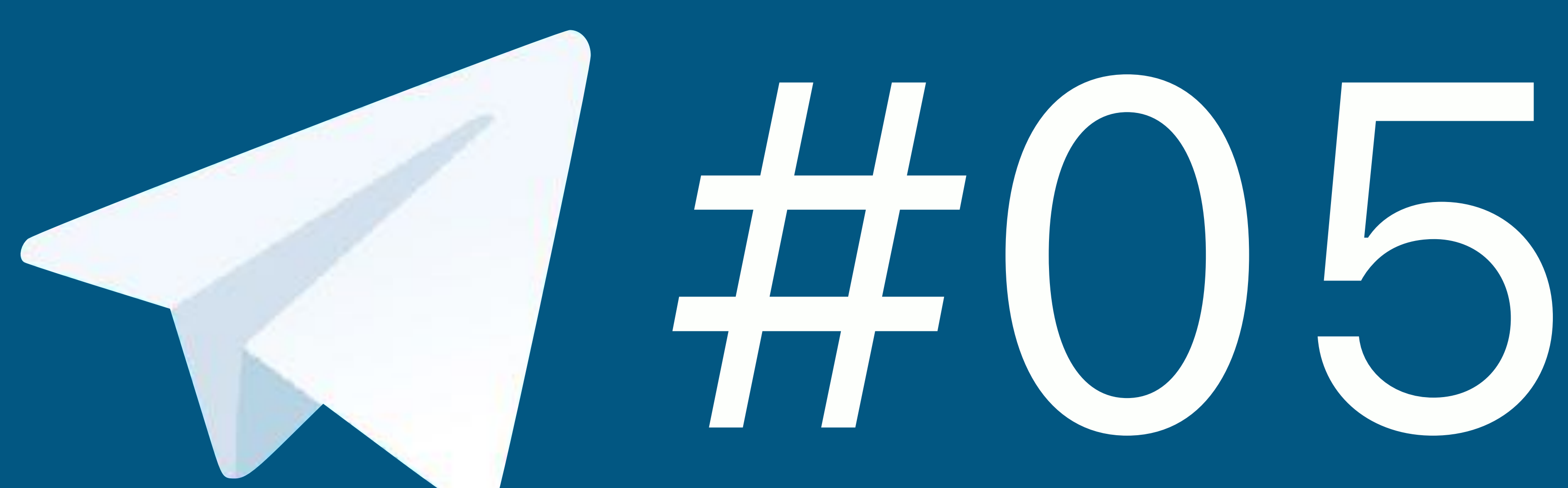
```
lesson05 — -zsh — 83×19
→ lesson05 clang++ -std=c++11 todos.cpp todos-main.cpp todo.cpp date.cpp -o todos
→ lesson05 ./todos
TODO: Buy beer (NORMAL) - 2020/Nov/5
TODO: Play piano (HIGH) - 2020/Nov/10
TODO: Write report (NORMAL) - 2020/Nov/1
TODO: Prepare keynote (LOW) - 2020/Dec/20

TODO: Write report (NORMAL) - 2020/Nov/1
TODO: Buy beer (NORMAL) - 2020/Nov/5

Next NORMAL todo
TODO: Write report (NORMAL) - 2020/Nov/1

TODO: Write report (NORMAL) - 2020/Nov/1
TODO: Buy beer (NORMAL) - 2020/Nov/5
TODO: Play piano (HIGH) - 2020/Nov/10
TODO: Prepare keynote (LOW) - 2020/Dec/20

→ lesson05
```



Take Home Message

The C++ Standard Template Library is a powerful set of general-purpose C++ classes that implement many popular and commonly used algorithms and data structures

C++11 provides the ability to create anonymous functions, called lambda functions, that can be used as parameters of STL algorithms

STL algorithms combined with iterators and lambda expressions offer unlimited ways of writing modern and powerful C++

~~Lesson 00: Introduction~~

~~Lesson 01: Hello, world!~~

~~Lesson 02: User-defined types~~

~~Lesson 03: Inheritance~~

~~Lesson 04: Polymorphism~~

~~Lesson 05: STL Containers~~

Lesson 06: Indirection

Lesson 07: Templates

Lesson 08: Exceptions

So let's go on C++ training with ❤️

(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac