2024/2025

ITC313

Mastering Object-Oriented Programming in C++: From Fundamentals to Best Practices

Pr. Dominique Ginhac
dginhac@u-bourgogne.fr

Last updated on November, 6, 2024 by dG

STL Containers

http://ginhac.com/ITC313/04-stl.pdf

Photo by BAILEY MAHON on Unsplash

Introduce STL containers a generic collection of class and algorithms that allow programmers to easily implement common data structures

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types

Lecture #02
Inheritance

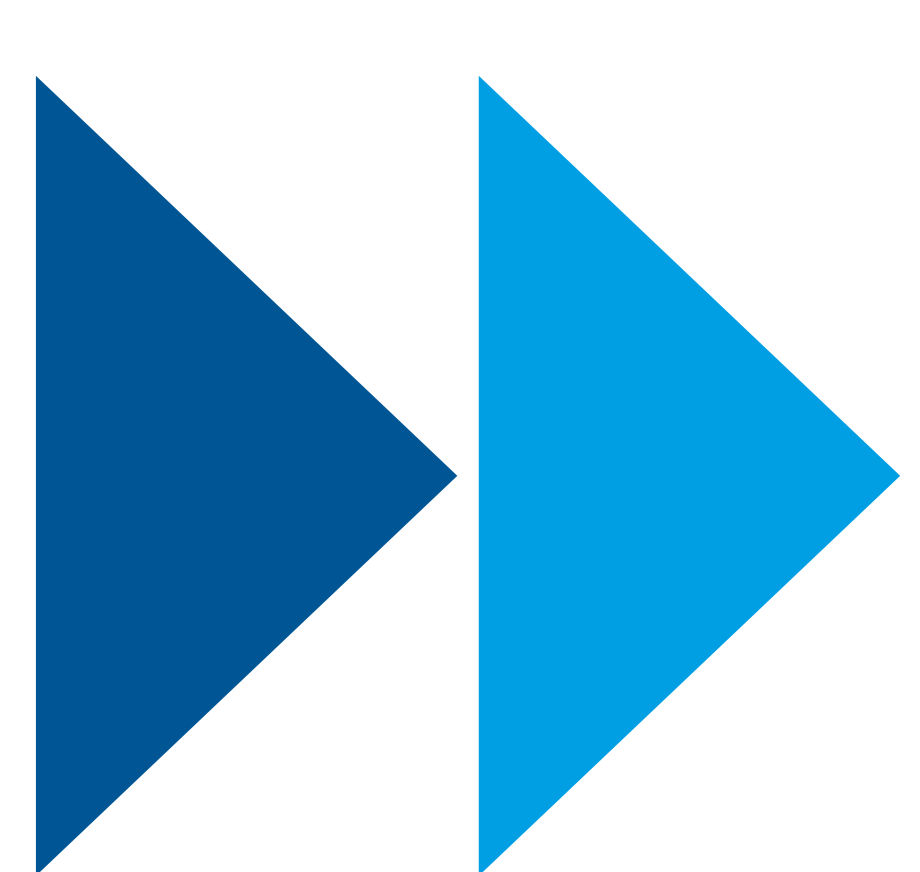
Lecture #03
Polymorphism

Today

Lecture #04
STL Containers

Lecture #05
Templates

Lecture #06
Exceptions



AGENDA

01 – What are STL containers ?

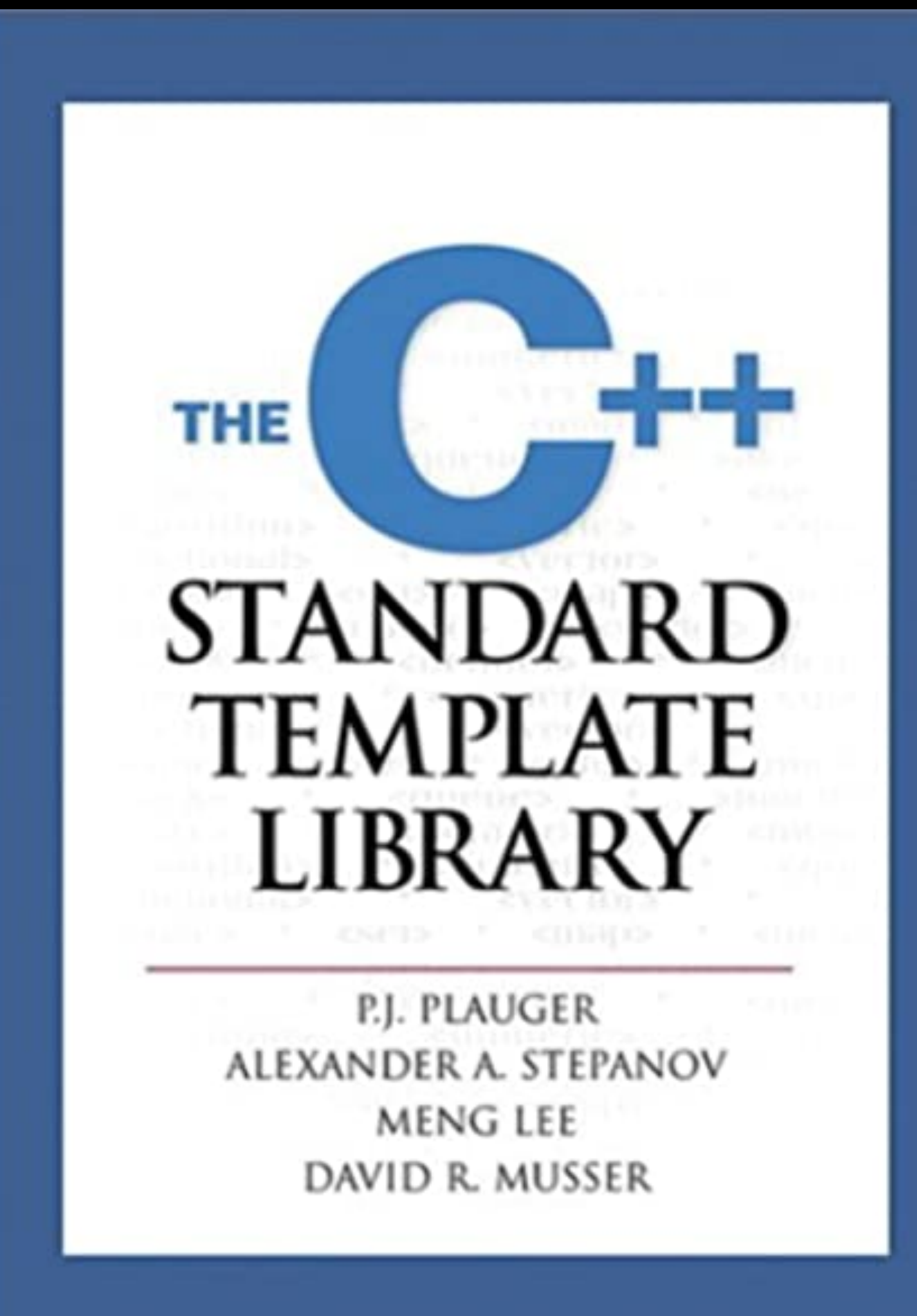
02 – Arrays

03 – The todos class with containers

04 – STL Algorithms

05 – Using algorithms with todos

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

AGENDA

01 – What are STL containers ?

02 – Arrays

03 – The todos class with containers

04 – STL Algorithms

05 – Using algorithms with todos

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,** std::deque
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

c-style-arrays.cpp

```
int main() {
    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;
}
```

```
-zsh — d0m
→  arrays clang++ c-style-arrays.cpp -o app
→  arrays ./app
The first prime number is: 2
The sum of the first 5 primes is: 28
→  arrays
```

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

Indexing out of range

c-style-arrays-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 (auto i=0; i<10; i++) {
        std::cout << prime[i] << " ";
    }
    std::cout  << std::endl;
    return 0;
}
```

```
-zsh — d0m                                                        ⌥⌘1
➜  arrays clang++ -std=c++11 c-style-arrays-out-of-range.cpp -o app
c-style-arrays-out-of-range.cpp:6:5: warning: array index 7 is past the end 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 here
    int prime[5] = {2,3,5,7,11};
    ^
c-style-arrays-out-of-range.cpp:7:18: warning: array index 7 is past the end 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 here
    int prime[5] = {2,3,5,7,11};
    ^
2 warnings generated.
➜  arrays ./app
13
2 3 5 7 11 32766 461308026 13 -278165376 32766
[1]    6078 abort      ./app
➜  arrays
```

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.

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

```
#include <array>
std::array <int, 5> primes = {2,3,5,7,11};
```

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

std-arrays.cpp

```cpp
int main() {
    //Declares an array of 6 ints. Size is always required
    std::array<int, 6> primes;
    primes[0] = 2; primes[1] = 3; primes[2] = 5;
    primes.at(3) = 7; primes.at(4) = 11; primes.at(5) = 13;
    //Declare and initialize with an initializer list
    std::array<double, 5> numbers = {-2.1, -1.45, 0.1, 1.9, 2.3};
    // Access to elements
    std::cout << "Size of primes = " << primes.size() << std::endl;
    std::cout << "primes front = " << primes.front() << std::endl;
    std::cout << "numbers back = " << numbers.back() << std::endl;
    // C-Style For loop
    for (auto i=0;i< primes.size();i++)
        std::cout << primes[i] << " ";
    std::cout << std::endl;
    // C++ for loop
    for (auto number : numbers)
        std::cout << number << " ";
    std::cout << std::endl;
    return 0;
}
```

-zsh — d0m ⌥⌘1

```
→  arrays clang++ -std=c++11 std-arrays.cpp -o app
→  arrays ./app
Size of primes = 6
primes front = 2
numbers back = 2.3
2 3 5 7 11 13
-2.1 -1.45 0.1 1.9 2.3
→  arrays
```

std::array out of range

stl-arrays-out-of-range.cpp

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

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

```
-zsh — d0m                                                        ⌥⌘1
➜  arrays clang++ -std=c++11 std-arrays-out-of-range.cpp -o app
➜  arrays ./app
After Last : 42
Still alive
libc++abi: terminating with uncaught exception of type std::out_of_range: array::at
[1]    5693 abort      ./app
➜  arrays
```

.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

Exceptions

Upcoming lesson

Comparing arrays

arrays-and-functions.cpp

```cpp
double array_mean(double data[], int size) {
    auto mean =0.0;
    for (auto i = 0; i < size; i++)
        mean+= data[i];
    return mean/size;
}

double stdarray_mean(std::array<double, 6> data) {
    auto mean=0.0;
    for (auto value: data)
        mean+=value;
    return mean/data.size();
}

int main() {
    double data1[6] = {10, 8.5, 12, 16.5, 13.5, 18 };
    std::array<double, 6> data2 = {10, 8.5, 12, 16.5, 13.5, 18};
    auto mean1 = array_mean(data1,6);
    auto mean2 = stdarray_mean(data2);
    std::cout << mean1 << std::endl;
    std::cout << mean2 << std::endl;
    return 0;
}
```

-zsh — d0m ⌥⌘1

```
➜  arrays clang++ -std=c++11 arrays-and-functions.cpp -o app
➜  arrays ./app
13.0833
13.0833
➜  arrays
```

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.

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

```
#include <vector>
std::vector <int> primes = {2, 3, 5, 7, 11, 13};
```

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

std-vector.cpp

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

-zsh — d0m ⌥⌘1

```
➜  arrays clang++ -std=c++11 std-vector.cpp -o app
➜  arrays ./app
Size = 3
data: 666 666 666
Size = 6
data: 666 666 666 42 1337 314
data: 666 666 707 0 1337 314
data: 666 666 707 0
data: 666 666 707 0 101 101 101 101
data: 666 666
data:
➜  arrays
```

std::vector out of range

std-vector-out-of-range.cpp

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

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

```
-zsh — d0m                                                    ⌥⌘1
➜  arrays clang++ -std=c++11 std-vector-out-of-range.cpp -o app
➜  arrays ./app
Using index
After Last : 666
data: 14 30 -23 15 -13

Using method at
libc++abi: terminating with uncaught exception of type std::out_of_range: vector
After Last : [1]    7047 abort      ./app
➜  arrays
```

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

Questions

AGENDA

01 – What are STL containers ?

02 – Arrays

03 – The todos class with containers

04 – STL Algorithms

05 – Using algorithms with todos

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<Todo> _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;
};
```

```
class Todos {
 private:
    std::vector<Todos> _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

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

todos.h

```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
```

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.

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 (auto 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

main.cpp

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

-zsh — d0m ⌥⌘1

```
➜  todos make
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17  -o app todos.o todo.o date.o main.o
➜  todos ./app
Todo setCompleted
TODO: Buy beer (NORMAL) - 5/11
TODO: Play piano (HIGH) - 10/11
DONE: Write report (NORMAL) - 1/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11
DONE: Write report (NORMAL) - 1/11
TODO: Prepare keynote (LOW) - 20/12
➜  todos
```

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::array or std::vector

```cpp
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 (auto i=0; i<5; i++) {
        std::cout << data1[i] << " " << data2.at(i) << " ";
    }
    std::cout << std::endl;
    for (auto d: data3) {
        std::cout << d << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

basic-iterators.cpp

```
-zsh — d0m                                                      ⌥⌘1
→  iterators clang++ -std=c++11 basic-iterators.cpp -o app
→  iterators ./app
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15
→  iterators
```

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();
```

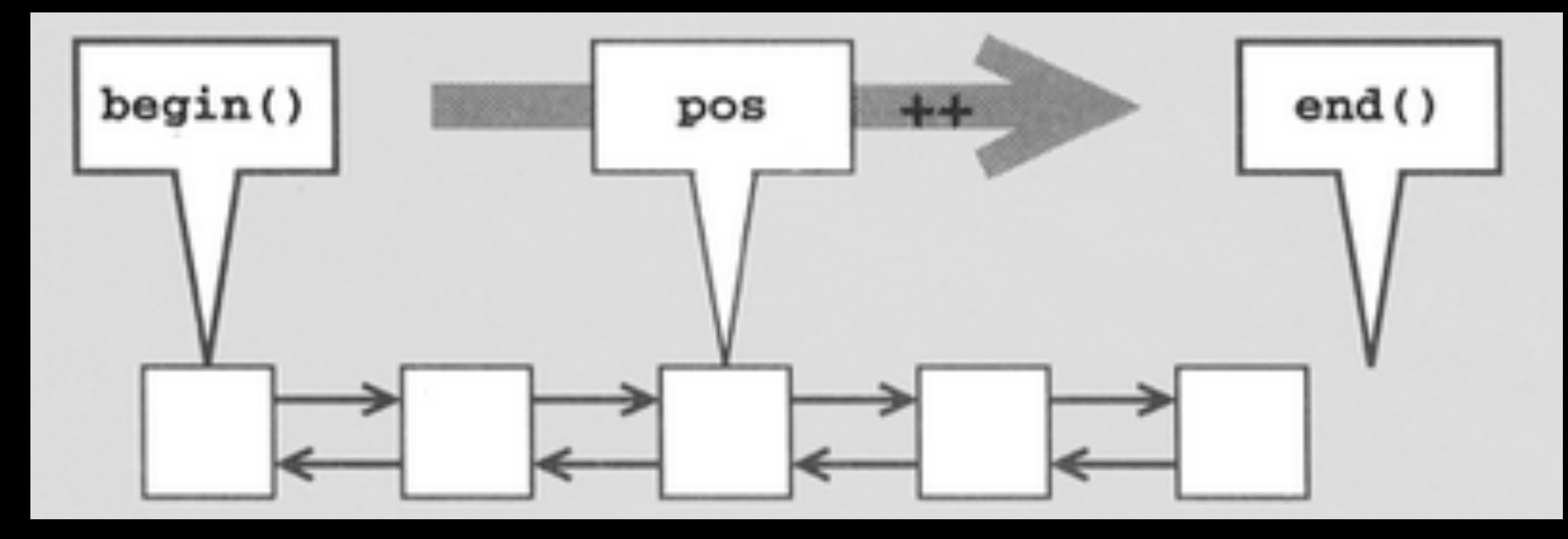
Specifying the data type for an iterator is a bit lengthy

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

basic-iterators2.cpp

```
int main() {
    std::vector<int> data = {1,3,5,7,9};
    std::cout << “for loop: ";
    for (auto d: data)
        std::cout << d << " ";
    std::cout << std::endl << “for loop with iterators: ";
    for (auto it=data.begin();it != data.end();++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl << “while loop: ";
    int i=0;
    while (i<data.size()) {
        std::cout << data.at(i) << " ";
        i++;
    }
    std::cout << std::endl << “while loop with iterators: ";
    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 Lecture about overloading ++ and -- operators)

```
-zsh — d0m                                    ⌥⌘1
➜  iterators clang++ -std=c++11 basic-iterators2.cpp -o app
➜  iterators ./app
for loop: 1 3 5 7 9
for loop with iterators: 1 3 5 7 9
while loop: 1 3 5 7 9
while loop with iterators: 1 3 5 7 9
➜  iterators
```

Other ex. of C++ iterators

basic-iterators3.cpp

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

```
-zsh — d0m                                              ⌥⌘1
→  iterators clang++ -std=c++11 basic-iterators3.cpp -o app
→  iterators ./app
while loop: 9 7 5 3 1
while loop with iterators: 9 7 5 3 1
while loop with iterators: 9 7 5 3 1
→  iterators
```

Questions

AGENDA

01 – What are STL containers ?

02 – Arrays

03 – The todos class with containers

04 – STL Algorithms

05 – Using algorithms with todos

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

copy.cpp

```
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 << "Basic: " << primes_copy;

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

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

-zsh — d0m ⌥⌘1

```
➜  algorithms clang++ -std=c++11 copy.cpp -o app
➜  algorithms ./app
Basic: 2 3 5 7 11 13 17 19 23 29
Push_back: 2 3 5 7 11 13 17 19 23 29
Algorithm: 2 3 5 7 11 13 17 19 23 29
➜  algorithms
```

How to subset container data

range.cpp

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

```
-zsh — d0m                                        ⌥⌘1
➜  algorithms clang++ -std=c++11 range.cpp -o app
➜  algorithms ./app
Initial: 1 3 5 7 9 11 13 15 17 19
Subset (2-7) - for: 5 7 9 11 13 15
Subset (2-7): - copy:5 7 9 11 13 15
Subset (2-7): - range: 5 7 9 11 13 15
➜  algorithms
```

How to sort container data

sort.cpp

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

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

-zsh — d0m ⌥⌘1

```
➜  algorithms clang++ -std=c++11 sort.cpp -o app
➜  algorithms ./app
Initial: 0 -4 2 10 1 -5 6 -7 12 3
Sort: -7 -5 -4 0 1 2 3 6 10 12
Reverse: 12 10 6 3 2 1 0 -4 -5 -7
Initial: Hello word The answer is 42
Sort: 42 Hello The answer is word
Reverse: word is answer The Hello 42
➜  algorithms
```

How to remove data

remove.cpp

```
int main() {
    std::vector<int> numbers = {0,10,20,10,10,30,40,10,-10,0};
    std::cout << "Initial: ";
    std::cout << "Size: " << numbers.size() << " - Data: " << numbers;
    // remove all the numbers equal to 10
    // and returns an iterator on the last element
    std::cout << "Remove all 10 ";
    auto it_end = std::remove(numbers.begin(),numbers.end(), 10);
       // numbers still have 10 elements
    std::cout << "Size: " << numbers.size() << " - Data: " << numbers;
    // Need to delete the last 4 elements using the member function erase
    std::cout << "Erase last nb ";
    numbers.erase(it_end, numbers.end());
    std::cout << "Size: " << numbers.size()
    << " - Data: " << numbers;
    return 0;
}
```

-zsh ⌥⌘2

```
➜  algorithms clang++ -std=c++11 remove.cpp -o app
➜  algorithms ./app
Initial
Size:10 - Data: 0 10 20 10 10 30 40 10 -10 0
Remove all 10
Size:10 - Data: 0 20 30 40 -10 0 40 10 -10 0
Erase last nb
Size:6  - Data: 0 20 30 40 -10 0
➜  algorithms
```

How to find an element

find.cpp

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

```
-zsh                                                        ⌥⌘2
→  algorithms clang++ -std=c++11 find.cpp -o app
→  algorithms ./app
Initial: 0 10 20 42 10 30 42 42 -10 42 0
First Found at position 3
Found 4 times
→  algorithms
```

How to find multiple values

multiple_find.cpp

```
int main() {
    std::vector<int> numbers = {0,10,20,42,10,30,42,42,-10,42,0};
    std::cout << “Initial: “ << numbers;
    auto it = numbers.begin();
    while (it != numbers.end()) {
        it = std::find(it, numbers.end(), 42);
        if (it != numbers.end()) {
            auto index = std::distance(numbers.begin(), it);
            std::cout << "Found at position " << index  << std::endl;
            ++it;
        }
    }
    return 0;
}
```

-zsh ⌥⌘2

```
→  algorithms clang++ -std=c++11 multiple-find.cpp -o app
→  algorithms ./app
Initial: 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
→  algorithms
```

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_int(const int a, const int b) {
    return a < b;
}
bool greater_int(const int a, const int b) {
    return a > b;
}
bool less_str(const std::string& a, const std::string& b) {
    return a < b;
}
bool greater_str(const std::string& a, const std::string& b) {
    return a > b;
}
```

Sort using a custom function

sort-with-custom-function.cpp

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

Only provide the name of the function.

No need () for function calls.

```
-zsh — d0m                                        ⌥⌘3
➜  algorithms clang++ -std=c++11 -o app
sort-with-custom-function.cpp
➜  algorithms ./app
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
➜  algorithms
```

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

functor.cpp

```
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;
}
```

-zsh — d0m ⌥⌘3

```
➜  algorithms clang++ -std=c++11 -o app functor.cpp
➜  algorithms ./app
15
15
52
➜  algorithms
```

Using a functor on a vector

functor-with-vector.cpp

```
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;
}
```

```
-zsh                                                      ⌥⌘3
➜  algorithms clang++ -std=c++11 -o app functor-with-vector.cpp
➜  algorithms ./app
0 -4 2 10 1 -5 6 -7 12 3
5 1 7 15 6 0 11 -2 17 8
➜  algorithms
```

Sort using a functor

sort-with-functor.cpp

```
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;
}
```

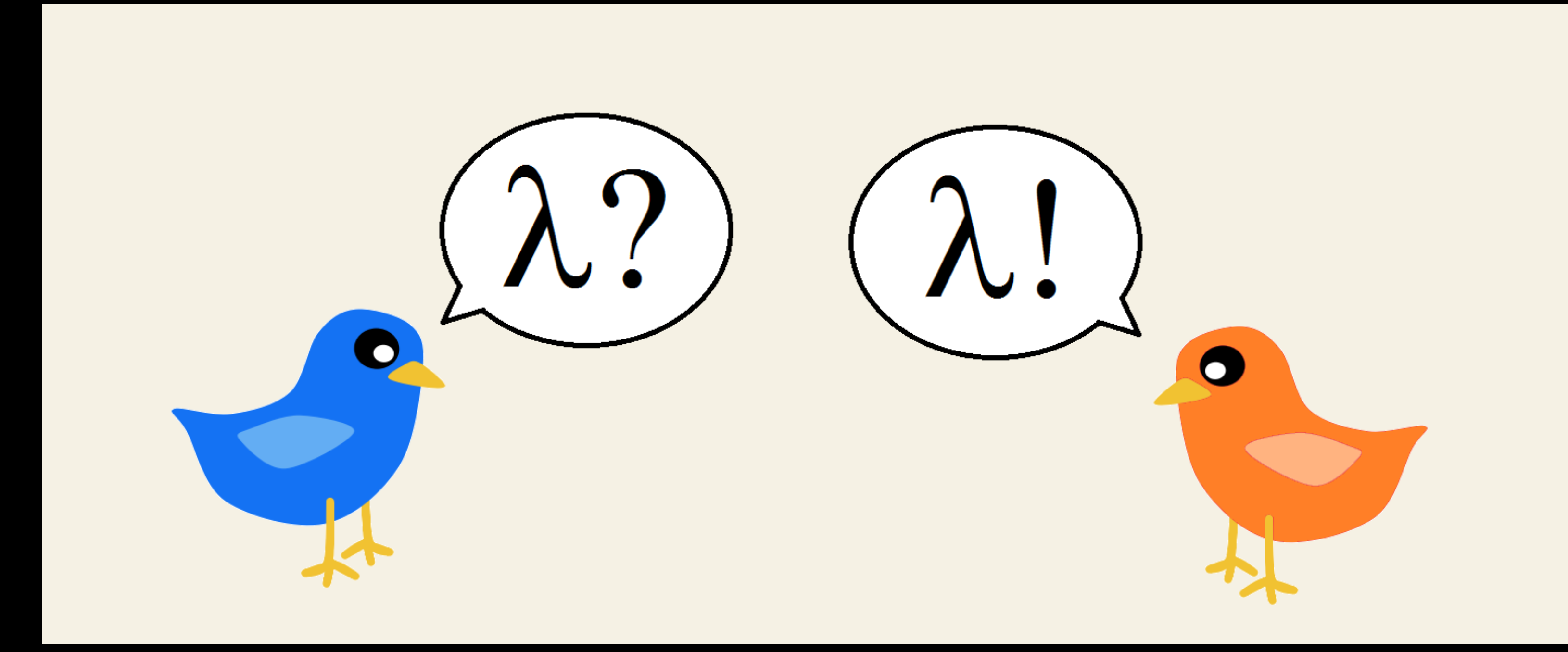
Need () for functor calls

```
-zsh                                                    ⌥⌘3
→  algorithms clang++ -std=c++11 -o app sort-with-functor.cpp
→  algorithms ./app
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
→  algorithms
```

Sort using Lambda expressions

A lambda expression is a small snippet of code that provides 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 my_functor;
    std::cout << my_functor(4) << " " << my_functor(-5) << std::endl;

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

lambda.cpp

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

```
-zsh                                                ⌥⌘3
→  algorithms clang++ -std=c++11 -o app lambda.cpp
→  algorithms ./app
1 0
1 0
→  algorithms
```

Lambda expressions with capture

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

lambda-capture.cpp

```cpp
int main() {
    int upper = 42;
    auto no_capture = [] (int value, int max) { return 0 < value && value < max;};
    std::cout << no_capture(4, upper) << std::endl;

   //Compilation error because upper is not in the lambda scope
    //auto no_capture2 = [] (int value) { return 0 < value && value < upper;};

    auto capture = [upper] (int value) { return 0 < value && value < upper;};
    std::cout << capture(43) << std::endl;
    return 0;
}
```

```
-zsh                                                    ⌥⌘3
→  algorithms clang++ -std=c++11 -o app lambda-capture.cpp
→  algorithms ./app
1
0
→  algorithms
```

Using a lambda on a vector

lambda-with-vector.cpp

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

-zsh ⌥⌘3

```
→  algorithms clang++ -std=c++11 -o app lambda-with-vector.cpp
→  algorithms ./app
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
→  algorithms
```

Sort using Lambda expressions

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

```
int main() {
    std::vector<int> numbers = {0,-4,2,10,1,-5,6,-7,12,3};
    std::cout << “Initial: “ << numbers;
    // sort using a custom function
    std::sort(numbers.begin(),numbers.end(), less);
    std::cout << “Sort function <: “ << numbers;
    // sort using a functor
    std::sort(numbers.begin(),numbers.end(), Greater());
    std::cout << “Sort functor >: “ << numbers;
    // sort using a lambda
    std::sort(numbers.begin(),numbers.end(), [](int a, int b) {return a<b;});
    std::cout << “Sort lambda <: “
              << numbers;
    return 0;
}
```

sort-with-lambda.cpp

```
-zsh                                                    ⌥⌘3
➜  algorithms clang++ -std=c++11 -o app sort-with-lambda.cpp
➜  algorithms ./app
Initial: 0 -4 2 10 1 -5 6 -7 12 3
Sort function <: -7 -5 -4 0 1 2 3 6 10 12
Sort functor >: 12 10 6 3 2 1 0 -4 -5 -7
Sort lambda <: -7 -5 -4 0 1 2 3 6 10 12
➜  algorithms
```

Questions

AGENDA

01 – What are STL containers ?

02 – Arrays

03 – The todos class with containers

04 – STL Algorithms

05 – Using algorithms with todos

A real-life case study

How to sort the todos in our todo list ?

How to find specific todos ?

The Todos class

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
```

Add three new member functions (next, sort, find) to Todos class

Implementation of sort and next

todos.cpp

```
#include "todos.h"
namespace todo {
    void Todos::sort() {
        std::sort(_todos.begin(),_todos.end(),
            [](const Todo& t1, const Todo & 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 (the first element of the vector) if it exists

Implementation of find

todos.cpp

```
#include "todos.h"
namespace todo {
    Todos Todos::find(int priority) const {
        auto it = _todos.begin();
        Todos results;
        while (it != _todos.end()) {
            it = std::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

main.cpp

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

-zsh ⌥⌘3

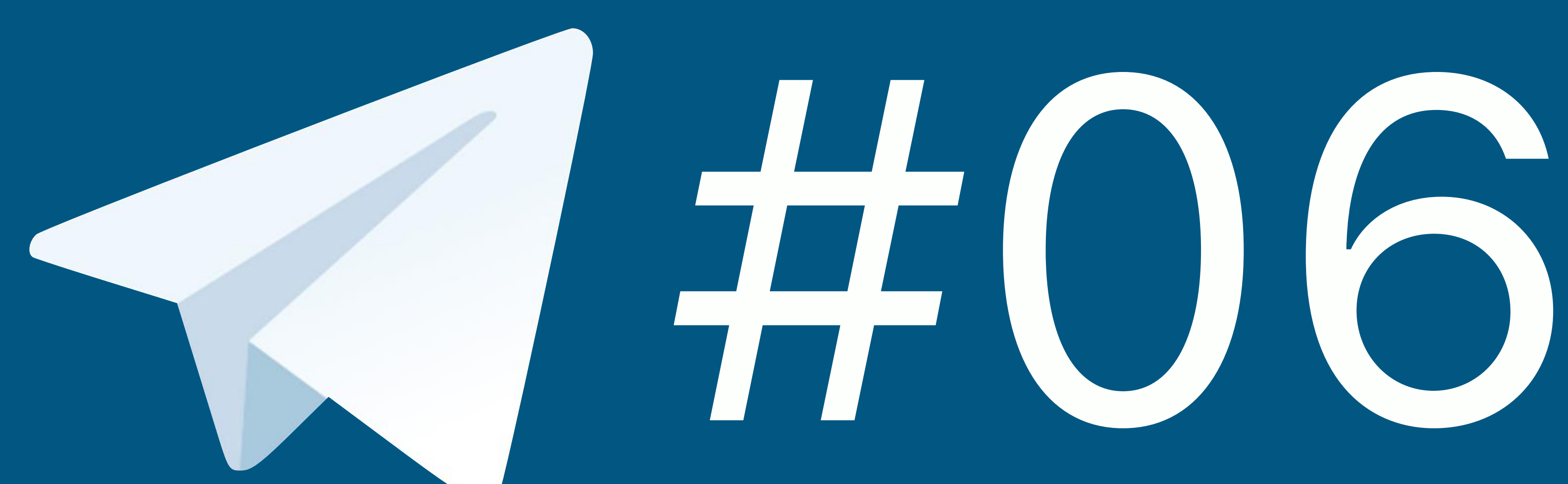
```
➜  todos-stl make
clang++ -std=c++17 -MMD -c  todos.cpp
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17  -o app todos.o todo.o date.o main.o
➜  todos-stl ./app
TODO: Buy beer (NORMAL) - 5/11
TODO: Play piano (HIGH) - 10/11
TODO: Write report (NORMAL) - 1/11
TODO: Prepare keynote (LOW) - 20/12

TODO: Buy beer (NORMAL) - 5/11
TODO: Write report (NORMAL) - 1/11

TODO: Write report (NORMAL) - 1/11
TODO: Buy beer (NORMAL) - 5/11

Next NORMAL todo
TODO: Write report (NORMAL) - 1/11

➜  todos-stl
```



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++.

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Come visit us at

https://github.com/dginhac/esirem-itc313

This work is **licensed** under a
Creative Commons Attribution-NonCommercial 4.0 International License.

https://creativecommons.org/licenses/by-nc/4.0/