Foreword

In order to do this lab you must have access to the school's virtual machine (see wiki on Moodle), your code from the previous lab.

In this exercise, you will cover the following topics:

- Use an STL container;
- Call an STL Algorithm;
- Look at STL and Polymorphism;
- Create a totally random forest. ***[BONUS]***.

Advice

- Use `man` an especially `man 3` for the development pages ;
- The website `https://en.cppreference.com` is your most valuable friend for the Teaching Unit;

Part I
Using a simple container

1 Represent a forest with an `std::vector`

At the moment, if we want to draw a forest containing an arbitrary number of trees we need to allocate the memory ourselves by using the `new` keyword and check that the memory is properly handled (aka that the objects are deleted). Thanks to the STL, we can use various containers to represent the forest and manage objects more easily[1].

In this section we will see what are the constraints of using a simple `std::vector` container.

Instruction

Write down the following code to create an `std::vector` of `Pine` and add two pines to it.

```
int main(int argc, char** argv) {
  std::cout << "Launching the main program" << std::endl;

  Pine p1;
  Pine p2;

  p1.info();
  p2.info();

  //create a vector of pines to represent the forest
  std::vector<Pine> forest;

  //add pines to the forest

  forest.push_back(p1);
  forest.push_back(p2);

  std::cout << "End of main program - destroying heap objects" << std::endl;

  std::cout << "End of main program - destroying stack objects" << std::endl;
  return 0;
}
```

Question 1

- Why do you see calls to the `Pine` copy constructor ?
- What happens if you remove your implementation of the copy constructor[a]?

[a] just comment it, it will be needed later

[1] Or so it seems...

Instruction

If we want to call the `draw()` method on each `Pine` inside the vector we will need to use an `iterator` of type `std::vector<Pine>::iterator`. Write a simple loop that uses this iterator to call the `draw()` method.

You can also call the `info()` method to verify that the objects inside the vectors are indeed copies.

Question 2

- Since our call is not modifying the `Pine` what other iterator could be used ?
- If we want to use this second iterator what are the needed modifications : to the `Pine` class and the `Tree` class ?

2 Removing elements from the vector

In this section, we will remove an element from the vector and see what are the implications for the elements inside it.

Instruction

First, comment or remove the overloading of the "=" operator in your `Pine` class.

Then add the following lines after your loop :

```
std::cout << "Removing the first pine of the vector" << std::endl;

forest.erase(forest.begin());
```

Question 3

- Does your program compile ?
- Does it run properly ? Explain what happened.

Instruction

Uncomment the "=" operator and check that everything is working properly.

Warning

Before using the STL, always check the documentation, for example the `std::vector` page mentions the following requirements for the class T (See Fig 2).

Template parameters

T - The type of the elements.

T must meet the requirements of *CopyAssignable* and *CopyConstructible*.	(until C++11)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of *Erasable*, but many member functions impose stricter requirements.	(since C++11) (until C++17)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type meets the requirements of *Erasable*, but many member functions impose stricter requirements. This container (but not its members) can be instantiated with an incomplete element type if the allocator satisfies the allocator completeness requirements.	(since C++17)

Feature-test macro	Value	Std	Comment
`__cpp_lib_incomplete_container_elements`	`201505L`	(C++17)	Minimal incomplete type support

Part II
Using an algorithm

Now that we are able to safely store `Pine` objects inside an `std::vector` we are going to apply a simple sort algorithm provided by the STL to the vector.

3 Create a bigger vector

Instruction

In your main program, create a loop that can add a fixed number of `Pine` inside an `std::vector`.

Question 4

- Why do you see many calls to either the copy/move constructors and destructors ?

4 Using a sorting algorithm

In the previous lab, we have defined that a `Tree` class has a `size` attribute and that comparing two trees is equivalent to comparing their size. Thanks to this mechanism we can directly apply the sorting algorithm provided in the STL. In fact the documentation states :

" A sequence is sorted with respect to a comparator comp if for any iterator it pointing to the sequence and any non-negative integer n such that it + n is a valid iterator pointing to an element of the sequence, comp(*(it + n), *it) (or *(it + n) < *it) evaluates to false. "

Instruction

In your main program, use `std::sort` to sort all the elements in your pine vector.

Check that the sorting is working by printing the size of each tree after that.

Question 5

- What iterators do you need to use ?
- What happens if you try to use const_iterators ?

Part III
STL and Inheritance

5 Using an `std::vector<Tree>`

It would be interesting to store trees of different nature and use polymorphism to call their adequate method. An example would be drawing a forest with various types of trees all inside a vector.

Instruction

Replace the `std::vector<Pine>` by `std::vector<Tree>`.

Question 7

- Why is this impossible ?

6 Create a new class derived from `Pine`

In this section we want to create a specific kind of `Pine` which we will call `ChristmasTree`. As this tree inherits from a `Pine` and not an abstract class `Tree`, our previous idea could work.

Instruction

Create a derived class `ChristmasTree` that only has one `std::string` attribute `decoration` in addition to the original `Pine`.

```
...

class Christmas : public Pine {
private:
  //Christmas decoration
  std::string Decoration;

...
```

In the `.cpp` file a specific constructor to initialize a Christmas decoration as follows[a][b] :

```
Decoration = "<0x1b>[38;2;0;0;0m▒<0x1b>[38;2;0;0;0m▒<0x1b>[38;2;0;0;0m▒<0x1b>[38;2;255;255;0m@";
```

The last addition to this class code will be the `Draw()` method :

```
void Christmas::draw() const{
  std::cout << std::endl << Decoration;
  Pine::draw();
}
```

Test that your code works by creating a `ChristmasTree` and calling the `Draw()` method.

[a] Don't forget to override the copy/move constructors as well as the "=" operator
[b] The specific code snippet will be provided on Moodle

Instruction

In your main program, create an `std::vector<Pine>` and add two classic pines to it. Then, add two Christmas trees as well.

Next, call the `draw()` method on each element of the vector.

Question 8

- Can you add a `ChristmasTree` to an `std::vector<Pine>` ?
- What happens when you call the `draw()` method ?
- What is the name of this phenomenon ?

Part IV
BONUS - Create a forest at random

One of the way to solve the previous problem is to use a vector of pointers or a vector of references to the parent class.

Instruction

Modify your main program so that your forest can contain a random number of random trees (oak, pines or christmas trees) that are correctly drawn on the console.

Question 6

- What is the main risk of doing this manually ?

Appendices : Useful commands

```
$man COMMAND              #display the manual page for the given COMMAND
$man 3 FUNCTION           #display the developer manual for the given FUNCTION
$g++                      #GNU project C and C++ compiler
$make                     #GNU make utility to maintain groups of programs
$ldd                      #print shared object dependencies
$strace                   #trace system calls and signals
$strings                  #print the sequences of printable characters in files
$gdb                      #The GNU Debugger
```