EPFL Fall 2022
Master level
MATH-458
September 11, 2024

Programming Concepts in Scientific Computing

Introduction to Standard Library (STL)

The goal of the present exercises is to discover and manipulate the STL structures in C++11 fashion.

Exercise 1: *std::vector*

- In a main function, create a vector of double.

```
std::vector<double> array;
```

- In order to append items to the vector you can use the method 'push_back'. For example.

```
array.push_back(1.);
array.push_back(2.);
array.push_back(10.);
```

- Append 100 values ranging from 0 to 2π (π is defined under the standard name M_PI)
- Print the size of the vector to screen

```
std::cout << array.size() << std::endl;
```

- If you want to resize the vector you can use the 'resize' method. If you enlarge the vector, what values will be contained in the extra part of the vector ? What is the 'assign' method doing ?
  As example C++ Reference website is of great help to explore methods of STL classes.
- In order to iterate over the values contained in the array a standard loop can be written like:

```
std::vector<double>::iterator it = array.begin();
std::vector<double>::iterator end = array.end();
for (; it != end ; ++it){
  double & val = *it;
  ... what you want ...
}
```

- Write such a loop with C++11 syntax, using a **range-loop** and the **auto** keyword in order to construct another array (named sin_array) that contains the sinus values of these other values.
- Re-Write it with a foreach.
- Iterate once more to generate a file containing two columns (CSV format) with the values of array and sin_array. Scientific notation, with 20 digits of precision is expected. You can apply a lambda functor to factor this loop to the maximum.

Exercise 2: *Memory Management*

In C++, the programmer has to manage two kinds of memory: the **stack** and the **heap**. Variables allocated on the stack are short-lived: they die at the end of the block in which they were created. However the stack is very fast, which is good for performance. If we want data to live longer than a block, we allocate on the heap, which is much larger than the stack, but also much slower.

In this exercise, we will manipulate objects on the stack and the heap, and exploit dynamic allocation for persistent data (Use file `memory_first.cc`).

- Fill in the blanks in the starting point code so that `values` is an array in the stack of size 10.
- Change the previous code so that `values` is on the heap. Why is there a compile error? Correct it.
- Is the memory allocated on the heap freed at the end of the program? What should you change?

We now look at data persistence. You may have to make minor changes to the rest of the program depending on how you define the function and what return type you use (Use file `memory_second.cc`):

- Implement the function `stack_allocation` to return the address of a `std::vector` of `n` `int`s on the stack.
- Implement the function `heap_allocation` to return the address of a `std::vector` of `n` `int`s on the heap.
- Why is the program failing?
- After removing the stack allocation to make the program run successfully, is the memory correctly managed?

Manipulating raw pointers as we have done until now is to be avoided in modern C++: we use the following alternatives when we can:

- `int values[10]` → `std::array<int, 10> values`
- `int * values = new int[10]` → `std::vector<int> values(10)`
- `std::vector<int> * heap_allocation(int n)`
  → `std::unique_ptr<std::vector<int> > heap_allocation(int n)`

Now you can redo the first two questions with `std::array` and `std::vector`, and replace the `new` in the `heap_allocation` function by `std::make_unique`. Is the memory correctly managed?

Finally, what do you need to change to the file `memory_third.cc` to make it compile, so that the memory managed by the pointer `unique` is correctly transferred to `owner`?

Exercise 3: *std::map and std::set*

- In the `main.cc` file, define a data type `Point` which is a alias for `std::array<double, 3>` (please find on the cppreference website what is the usage of `std::array`).

```
using Point = std::array<double, 3>;
```

- In the main function, create a map from `std::string` to `Point`.

```
std::map<std::string, Point> map;
```

- Let us use this map to store relative coordinates of planets. In order to append items to the map you can use the '[ ]' operator:

```
map["sun"] = Point{0., 0., 0.};
```

- Add the earth, with the associated coordinates Triplet being (1.,0.,0.) in astronomical unit.
- What happens if you try to append an item with an already-existing entry ?
- Prevent this issue using the *map::find* method:

```
auto it = map.find("earth");
```

- Loop over the entries of the map and output the entire content to screen
- Create another map from `std::string` to `Point`. This map will contain coordinates of 4 planets

```
mercury : 0.25, 0., 0.
earth   : 1.0, 0., 0.
jupiter : 5.0, 0., 0.
sun     : 0., 0., 0.
```

- Create two sets (`std::set`) which contains only keys of the above created two maps. C++ Reference - Set
- Use function `std::set_intersection` to get the keys which are common in both sets. Store these keys in an another set. C++ Reference - Set Intersection
- Now, based on the above set of intersected keys create a new map which contains the data corresponding to those keys only.

Exercise 4: ***Standard Algorithms***

In this exercise, we will use algorithms of the Standard Template Library. They are generic implementation of useful, general-purpose algorithms (e.g. `std::sort`, `std::find`, etc.) that work with the STL containers, but also with user-defined types. Here, we will review some algorithms that are commonly used in scientific computing.

Algorithms are often combined with lambda-functions: they are anonymous functions, i.e. they do not have a name. They can easily be passed as arguments to algorithms to modify their behavior (e.g. a function which gives the order relationship of two elements for `std::sort`, a custom reduction operation for `std::reduce`).

For this exercise, look at the reference page `https://en.cppreference.com/w/cpp/algorithm`. Most algorithms take as parameters a `begin` and `end` iterators.

- Create a `std::vector<double>` of size `n`. Using the algorithm `std::fill` (not the member function `vector::fill` !), fill the vector with zeros. Check the expected result with a range-for loop.
- Look at the documentation for the `std::iota` algorithm. Apply it to the vector with a starting value of 1 (include `<numeric>`).
- Create a lambda function that takes a reference to `double` as an argument and squares it. Verify the expected result on a simple `double` variable. You can look up the syntax in the course notes.
- Using a range-for loop, apply this lambda to the previously initialized vector. Check the expected result.
- Replace the range-for loop with the `std::for_each` algorithm.
- We now want to compute the sum of all the elements in the vector. We can accomplish this by capturing a variable in a lambda function:
  - Create the variable that will hold the result
  - Create a lambda-function capturing that variable and summing its argument into it.
  - Using `std::for_each`, apply the lambda and check the sum result against $s = n(n+1)(2n+1)/6$.
  - If you get the wrong answer, check how you capture the result variable. Is it by-value or by-reference?
- Look up the documentation for `std::accumulate`. Do the previous question without a lambda function (be careful about the type of the `init` value).
- Finally, with the same vector containing squares of natural numbers, use the answer to the questions above to compute an approximate value of π using the sum of inverse squares, in reverse order (see the documentation of `std::vector`), without any loop.