Incredible Machines

First assignment - Tecniche di Programmazione Avanzata 2020-21

You will develop a C++ program to produce SVG (Scalar Vector Graphics) drawings of absurd parametric mechanical machines.

You will develop on your own devices and try integrating in your project devices made by others to produce the machines.

Teamwork: we will adopt a loose groupwork model, in which you can pick and choose the devices you want, even from colleagues you don't know.
Collaboration and communication via Github is encouraged, actually it would be best if you *only* communicated via Github issues (so please avoid email / chat!)

Evaluation:

- This work will earn you up to 3 points for the exam
- If you already have graded assignments from previous years, you can keep those and skip these assignments - please contact me explaining your case to receive a confirmation
- I'm much more interested *in the development process* than in the actual code. I'm sure to some of you all this issues / commits / documentation stuff will look a lot like bureacracy, and believe me, I do hate buracracy. BUT, this is basically the minimum required to become actually efficient when working in a team on long term projects.

Deadline: Tuesday 27 April 23:59
If you foresee problems with this schedule, please contact me ASAP.

IMPORTANT 1: remember we are doing this for learning how to cooperate, so should you find a problem in somebody else's code, you should definitely try communicating the problem via Github issues without any fear of ripercussions. In my judgement I will *not* consider issues which where discovered and reported: actually, the more I see you working in group and finding, reporting and fixing issues via Github, the happier I am (and grades go up...). So please use the tools I suggest and avoid communicating via email / chat, because that way I *cannot* know if you are actually interacting with others.

IMPORTANT 2: this is not a computational mechanics course, so no physics is required nor even wanted: just try to make something that makes some sense visually and be inventive!

DEFINITIONS

- Device: a single simple mechanical object
- Machine: a composition of devices

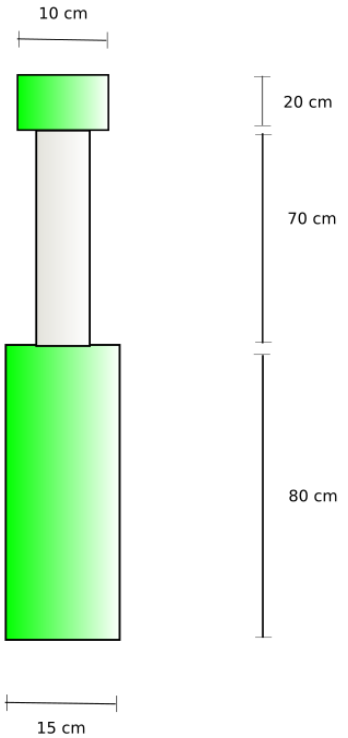
METHODOLOGY

We will follow the *Publish early, release often* development strategy, which means you should put online whatever you do as soon as you make it, *even if not working*: the important thing will be to put the code in the *right* place so people can understand its status, so use `fixXY / dev / main` branches and `vX.Y.Z` tags appopriately. Moreover, if you follow this strategy I will be able to give you suggestions (remember I'm there to help, not to control you!)

1 - Create one device

1. think about one simple device, like i.e. a piston:

   IMPORTANT: EACH student MUST create a DIFFERENT device!
   To avoid collisions, write what you want to make on the students spreadsheet



2. Create a new Github repo from min-cmake-prj , give it a meaningful name, like Piston. Please follow the existing folder structure: remember that the more you deviate from it, the harder will be for your colleagues to work with it.
3. Add a link to the repo on students spreadsheet in the appropriate column
4. create with any tool of your choice (like Inkscape) a sketch with relevant measures, push it to the repo and create an issue marked with @DavidLeoni to notify me for approval. You MUST have *at least* 3 parameters, which must constrain each other in some non-trivial way (i.e. the moving part of the piston shouldn't be longer than the

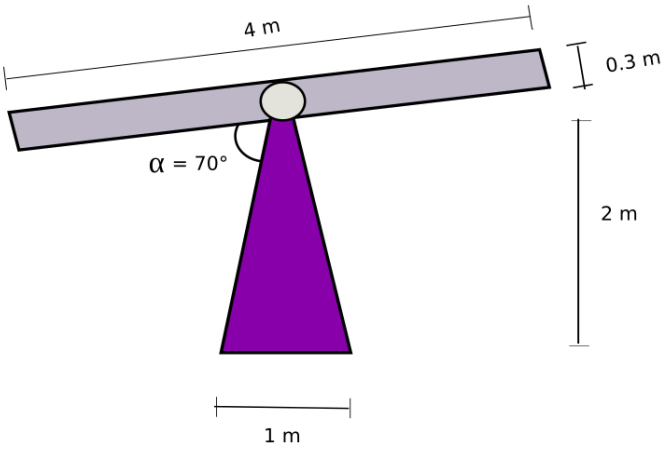
cylinder and should have a minimal height proportional to the cylinder in order to be useful)
5. define struct and constraints (i.e. min / max height, max run of the piston, …)
6. create `my_init` function which checks constraints
7. create `string my_to_svg(MyDevice*)` function which produces a C++ a `string` with the SVG code (see SVG section). (can it fail? if so, what should be the expected output?)
8. create `my_set_` functions to modify dimensions, which must check for violated constraints and report error codes, which must be properly described

```
/**
      Sets a new width in the structure
   -  if the new width is incompabile with other measures,
      RETURN 1, otherwise zero
*/
int my_set_width(MyDevice* device, new_width)
```

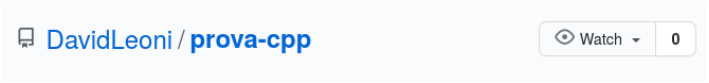
9. implement save to file function
10. implement `MyDevice* my_parse(string svg)` function, which creates a struct from a SVG textual representation
11. implement load from file function
12. create simple command line program to specify parameters and file to save to / load from
13. add tests (i.e.: can we load what we saved? Is the resulting structure the same?)
14. merge code on branch `main` and tag it with `v1.0.0`
15. add `with_measures` boolean parameter to to_svg function to include measurements in the drawing
    add `with_measures` to save function to include measurements
    NOTE: modifying the function signature means breaking code which depends on yours! So you would have to tag the change with a bump in MAJOR versioning number, like going from 1.0.0 to 2.0.0
16. merge code on branch `main` and tag it appropriately

2. Integrate a device

1. Look for *at least* one device made by some your colleague which might make some sense (remember there's no need to make realistic stuff!) if used in combination with your device. For example, let's say you find this swing:

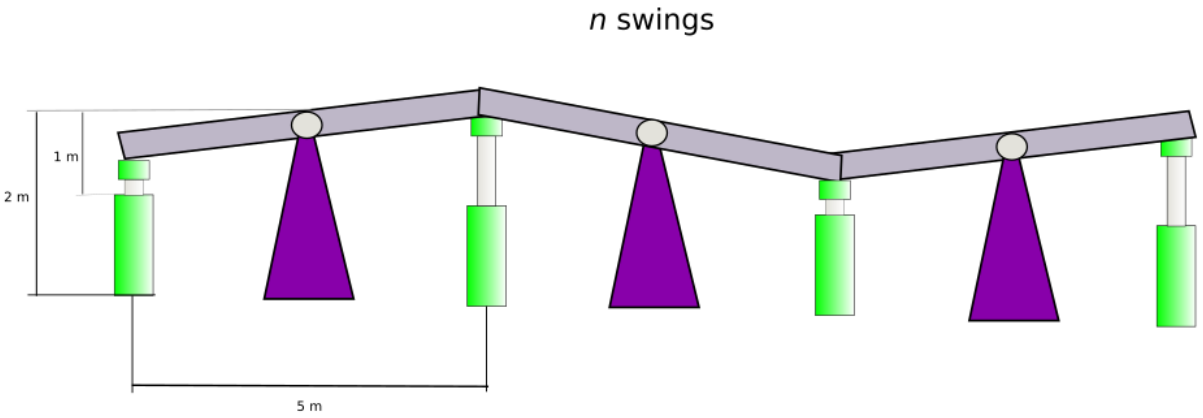


2. Click *Watch* button on your colleague repo to be notified when somebody opens issues in that repository



3. Open an issue on your repo to describe you are integrating the swing in your code. Add a @USERNAME to signal the guy / lady you are taking his/her code.
4. Copy your colleague code taken from `main` branch (because we suppose that code is stable) to your repository, writing in your commit text the link to the exact github commit (or even better, tag) from which it was taken
5. Make sure your project compiles and your colleague tests pass
6. Study the code of your colleague, and write some tests for it. Do your *worst* to break it: try passing negative numbers, exceedingly big parameters, NULL, whatever you can possibly think of. Try also to see if it follows the written documentation. If you spot a problem, clone your colleague repository, add tests in his/her code which show the problem and open a pull request in his/her repo, explaining what was wrong. Remember you can also paste a screenshot directly in the issues. If you also found a fix, include it.
7. Even if your colleague code works, you can also look for:
    a. missing / unclear documentation
    b. improper / missing prefixes
    c. also additions, i.e. comments in generated SVG

3. Combine devices



Try creating a new Machine from the devices you collected so far:

IMPORTANT: EACH student MUST create a DIFFERENT machine!
To avoid collisions, write what you want to make on the [students spreadsheet]

- do not care about creating a realistic machine: the only requirement is that it SHOULD visually make some sense.
- machine SHOULD have a number of parameters lesser than the parameters of the single devices, which should be derived according to the input parameters (they can even be constant)
- machine MUST have at least a device you created
- machine MUST have at least a device created by somebody else
- each device MUST be dinamically allocated
- the machine should be defined in such a way that it MUST be possible to specify *n* instances of a device, which MUST be connected in some way to the others.
- define new .h, .cpp files and test file for the machine
- define init, set, to_svg, parse (can you reuse devices parse functions?), save, load functions as before.
- Add the possibility to specify in main program which device / machine to output
- implement a function `bool are_equal(MyMachine*, MyMachine*)` which checks if the two machines are the same. NOTE they will have different pointers, so a deep scan will be needed!
- add tests
- merge in `main` branch and tag it
- create an issue with @DavidLeoni to notify me for final approval

4. [EXTRA] try integrating complex machines from your colleagues

5. [EXTRA] include SVG animations

[Example animation]

This would definitely be fun, but it is not required and may make things too complicated

Other examples can be found at the bottom of W3 examples page

Non functional requirements

PLEASE FOLLOW DEVELOPMENT GUIDELINES IN GIT TUTORIAL LAST EXERCISE

Furthermore:

- DO NOT care about physics (mass, density, ..), just dimensions
- DO NOT care about making realistic machines. Be inventive!
- development MUST be divided in features / bugs, examples:
    - feature 0: generate string of SVG
    - feature 1: saving to file
    - feature 2: loading from file
    - feature 3: flag for displaying measurements, …
    - bug: wrong constraint check

  Bug issues will become especially important when people will start using your device, you should signal them problems *only* via issues (if they enabled watch for the repo they should get automatic notification)
- BEFORE implementing a feature / fixing a bug, you MUST open an issue on Github and mark it as *enhancement* / *bug*
- EACH commit you do MUST contain a reference to the issue it is about
- DO NOT make cargo commits: while a commit MAY reference more than one issue, if it is about 10 features probably it's not a good commit
    - take particular care when reformatting, which could potentially mark thousands of changes
    - ALWAYS review what you are about to commit wit a `git status`
    - try limiting usage of `git add .` with the dot, often brings in too much stuff
- Code to produce MUST be C-style:
    - choose a world unique prefix (here we use `my` which would be silly)
    - each function MUST be prepended with the prefix, like `my_init`
      each struct MUST be prepended with Camelcase namespace, like `MyShape`
    - using C++ `string` instead of `char*` is allowed (it's much easier to work with)
    - all error signaling MUST be done C-style, with functions returning error codes in case of failures, which MUST be properly documented.
    - DO NOT use C++ exceptions
    - DO NOT use C++ namespacing, only prefixes
- Each function MUST be documented, Doxygen style, for example:

```
/**
 * Sum numbers in an array.
 *
```

```
 * @param values array whose values are summed.
 * @param n number of elements in the array
 * @return sum of `values`, or 0.0 if `values` is empty.
 */
 double my_sum(double* values, int n);
```

for other documentation examples you can follow [these guidelines]

- Each written function MUST have *at least* two tests. Some functions like `my_init` which can fail in many cases will actually need a lot of them !
- DO NOT use an SVG / XML library to write/load files

SVG

Some example: https://github.com/DavidLeoni/dii-tpa-svg-tests

Basic Transformations (translation, rotation, scaling …)

You can sketch ideas with inkscape program which unfortunately generates complex SVGs, and then start from scratch by looking at W3 schools examples and examples without `-inkscape` in the repo linked above

Visual Studio Code: To ease working you can install this SVG Preview Extension , which also allows to live preview changes you do while editing SVG files.

Constructing the SVG

To build the SVG, once you have a template you can keep increasing a string just like this: WARNING: *very* rought example!

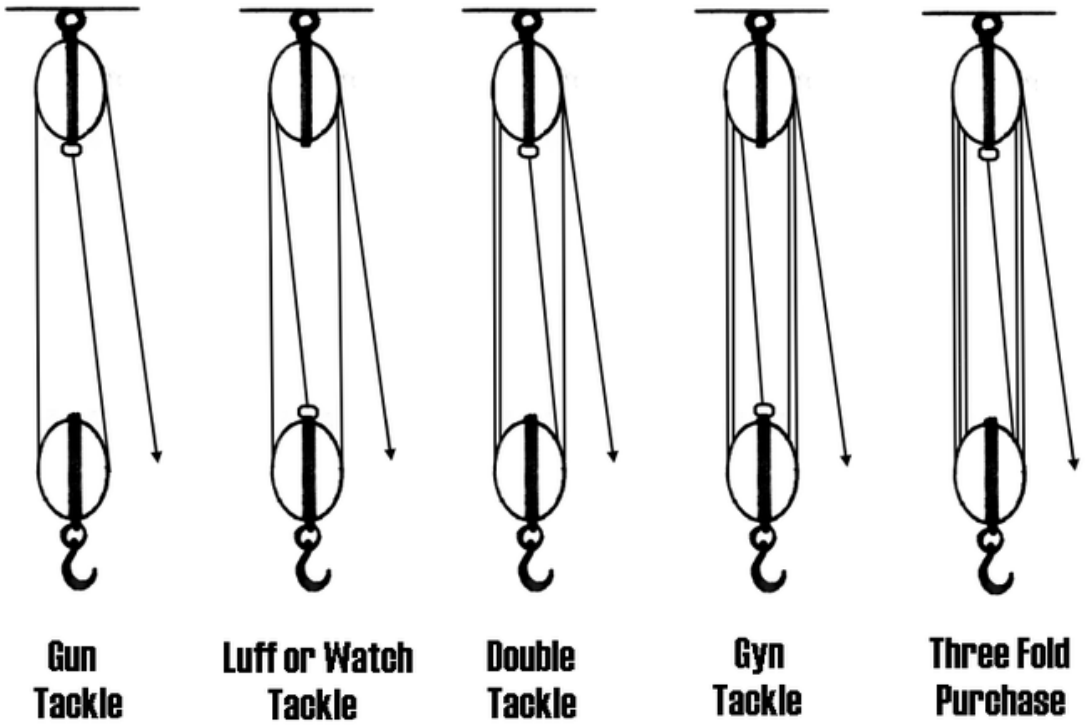
```
string s = "";
s += "<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
    width="800" height="600" >
"
int width  = 50
s += "<polygon points=\" "
s+ = 400
s+ = “,”
s+= width

etc …..
```

Read / Write string to file

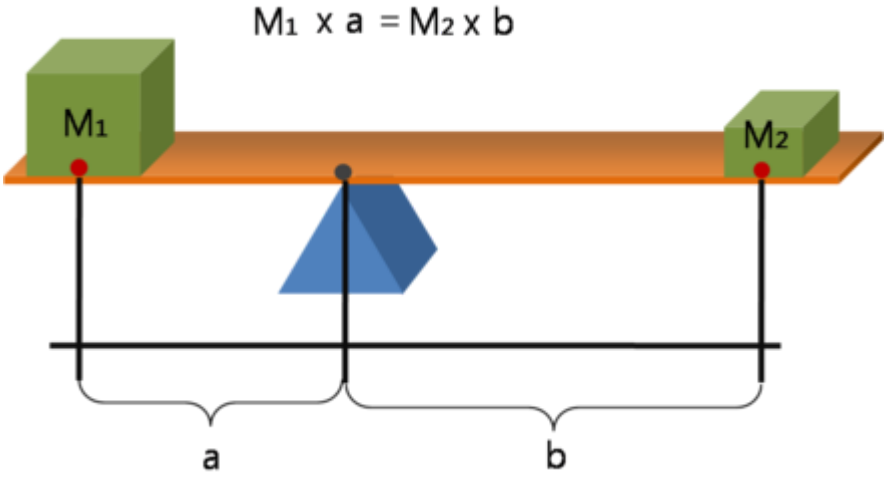
```cpp
#include <iostream>

#include "my_fraction.h"
#include "cxx_examples.h"

#include <iostream>
#include <fstream>
#include <streambuf>
#include <string>
#include <sstream>

using namespace std;

int main() {

 // Create and open a text file
 ofstream MyFile("filename.svg");

 string string_to_write = "Files can be tricky, but it is fun enough!";

 // Write to the file
 MyFile << string_to_write;

 // Close the file
 MyFile.close();


 // Read from file
 std::ifstream t("filename.svg");
 std::stringstream buffer;
 buffer << t.rdbuf();
 string s = buffer.str();

 cout << "I read this" << endl;
 cout << s << endl;


 return 0;
}
```

Some ideas

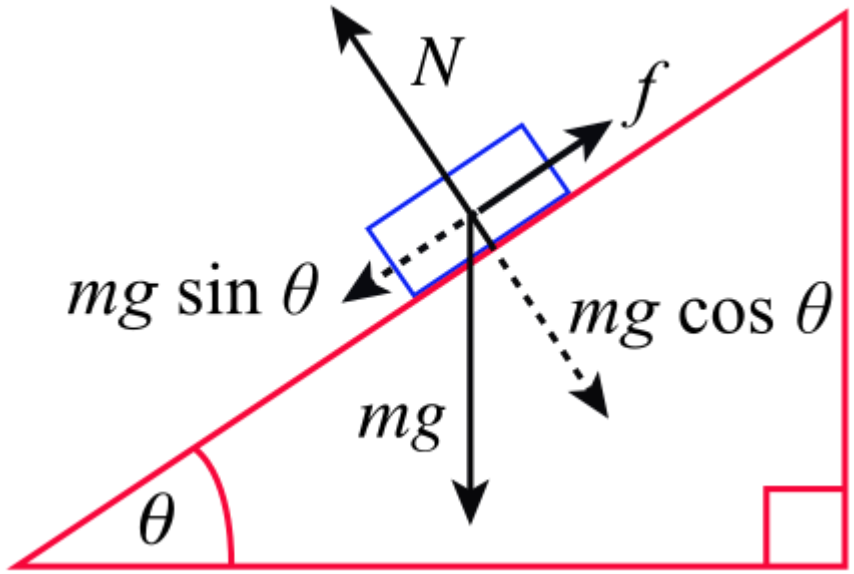
Pulley [Wikipedia]



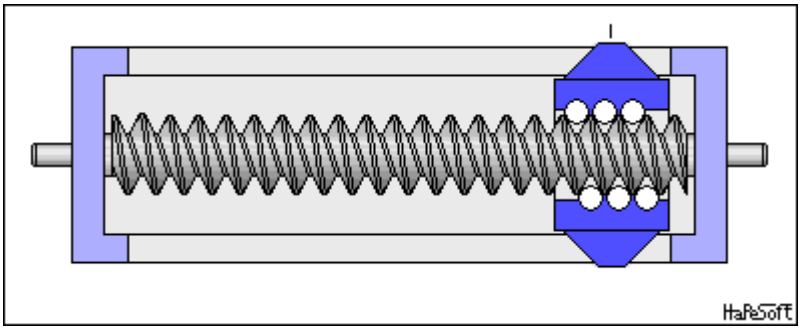
Lever (wikipedia)



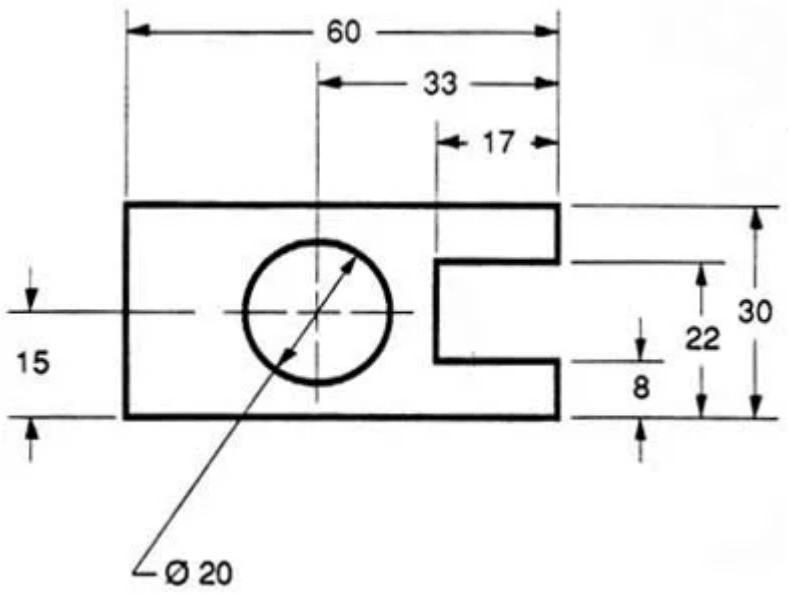
The inclined plane [Wikipedia]



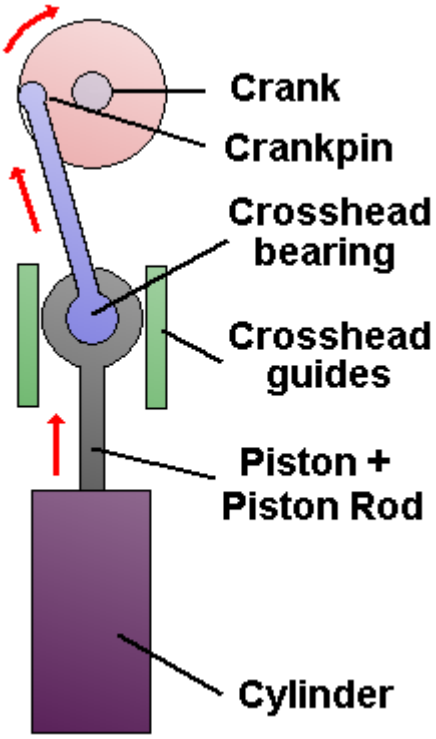
Screw [Wikipedia]



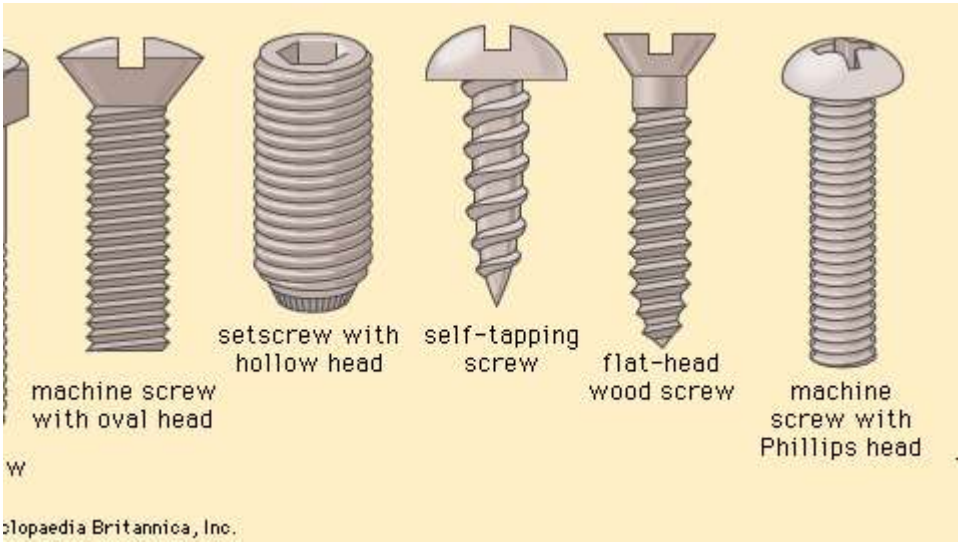
[MIT Engineering drawing]



Crosshead [Wikipedia]



The screw [britannica.com]



Adjustable Wrench