326.006 - Algorithms and Data Structures Exercises **Introduction**

Cleopatra Pau
Ioana.Pau@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Organizational Issues 1/4

- The course is meant as a practical addition for the theoretical part.
- Every week you are supposed to implement the programs from the theoretical course for the next week's exercises session.
- When time allows, you will do some additional problems.
- Homeworks are not graded per se, but their absence will lead to worsening your grade (0.3-0.4 addition to your final grade).
- Non-working homework submissions will not affect your grade.
- Not submitting any of the homeworks will result in failing the class.

Organizational Issues 2/4

- Presence is not mandatory, but it is recommended.
- The activity during class, as well as the extra problems might get you up to one point to the final grade.
- In the first week of February there will be an exam.
- Finding (serious) bugs in the proposed solutions may exempt you from one subject of the exam.
- The problems at the exam will need the studied algorithms, but will be completely new. Subject from the last year will be provided.

Organizational Issues 3/4

- The students coming to the class will be graded every week for their activity.
- Depending on how well the homework assignments were performed, the activity in class will comprise either
  - discussing the interesting issues;
  - (re)implementing the homeworks (without helping material);
  - or discussing applications of the studied algorithms.
- You should use your own laptops for this purpose.
- Homework submissions will be done via RISC's Moodle page, which you can access from KUSSS.

Organizational Issues 4/4

- Course language: English.
- Programming language: `c++`.
- Working environment:
  - text editor with highlighted syntax (your choice);
  - `c++` compiler (`g++` recommended)
  - `make`
  - In Linux both `g++` and `make` should be present by default. In Windows, they should be installed, and are included Cygwin and MinGW (help to be found here). Choose one and install it.
- Other choices for the language and the environment, are possible, but should be discussed before using them.

Compiling and Running 1/2

Use the command line window.

- Move to the source folder: `cd PATH`
- Windows:
  - Compile source code: `g++ FILENAME.cpp -o FILENAME.exe`

    Creates the executable `FILENAME.exe`.
  - or Make your sources: `make FILENAME.exe`

    Same effect.
  - Run the program: `FILENAME`

Compiling and Running 1/2

- Linux:
  - Compile source code: `g++ FILENAME.cpp -o FILENAME`

    Creates the executable `FILENAME`.
  - or Make your sources: `make FILENAME`

    Same effect.
  - Run the program: `./FILENAME`
- In the examples above all capital letters words should be replaced by the actual paths or names.

Reading integers from a file

```
int number;
ifstream inputfile;
inputfile.open("Input.txt");


while (inputfile >> number)
{
  ... // process the number
}

inputfile.close();
```

Timing an algorithm

```
#include <iostream>
#include <cstdio>
#include <ctime>
using namespace std;

int main()
{
clock_t start;
double duration;

start = clock();
/* algorithm to be timed */
duration = (clock() - start) / (double)
    CLOCKS_PER_SEC;

cout<< "duration: " << duration << endl;
}
```

Naming conventions

- Constants in ALL_CAPS with underscore separating the words.
- camelCase for all other names.
- Beginning with a capital if it's a class.
- Lowercase if not.
- (Functions are named like variables: camelCase (you can recognize it's a function since it requires ()).

Operations with Files

Returning to the beginning of a file:

```
inputfile.clear();
inputfile.seekg(0, ios::beg);
```

Finding out the length of the file:

```
inputfile.seekg (0, inputfile.end);
int length = inputfile.tellg();
inputfile.seekg (0, inputfile.beg);
```

makefile

A simple `makefile` contains a sequence of rules of the form:

```
target: dependencies...
command
command
command
...
```

Any rule can be called with `make target`.

`make` without any argument will call by default the first rule.

For each dependency the time stamp is checked before running it. Only if a dependency is not up-to-date, the corresponding dependencies and commands will be run.

makefile Example

```
# This is the makefile of the arithmetic expression
   evalution
# (Lines starting with # are comments)
# expression.cpp contains the main function

all: expression.o stack.o
g++ -o stack expression.o stack.o

expression.o: expression.cpp stack.h
g++ -c expression.cpp

stack.o: stack.cpp stack.h
g++ -c stack.cpp

clean:
rm *.o stack
```

makefile Variables

We want to run

```
g++ -std=c++11 contains3.cpp -o contains3
```

with make.

```
CC=g++
CXXFLAGS=-std=c++11
# additional useful variable
# folders src and include should be found in the
   current directory
VPATH = src include

all: contains3.cpp
$(CC): $(CXXFLAGS) contains3.cpp -o contains3

clean:
rm contains3
```