Computer Programming 1

Laboratory

(command line)

Argc and Argv – exercise 1a

Implement *multiply*, which takes as argument a list of numbers and outputs their products

Argc and Argv – exercise 1b

Implement *calculate*, which takes as first argument an option specifying the operation and then a list of numbers and performs the operation on the numbers

```
calculate --max 10 30 40 5
```

Argc and Argv – exercise 1c

Implement *calculate*, which takes as first argument an option specifying the operation and then a list of numbers and performs the operation on the numbers

```
calculate --max 10 30 40 5
```

Variant: extend the implementation by controlling the execution of different functions: --sum, --product, --max, --average

```
calculate --max 10 30 40 5
calculate --sum 10 30 40 5
```

Argc and Argv – exercise 1d

Implement *calculate*, which takes as first argument an option specifying the operation and then a list of numbers and performs the operation on the numbers

```
calculate --max 10 30 40 5
```

Variant: re-implement calculate, so that it takes as command line arguments the option, but it now reads the number from cin (one per line or separated by spaces, if you prefer).

Note1: remember to add checks to avoid crashes of the program in case of wrong inputs are interested by the user

Argc and Argv – exercise 1e

Implement *calculate*, which takes as first argument an option specifying the operation and then a list of numbers and performs the operation on the numbers

```
calculate --max 10 30 40 5
```

Variant: re-implement calculate, so that it takes as command line arguments the option, but it now reads the number from cin (one per line or separated by spaces, if you prefer).

Note1: remember to add checks to avoid crashes of the program in case of wrong inputs are interested by the user

Note2 (opt.): experiment with redirection, for instance by:

1. preparing a file "input.txt" in which each line contains input for calculate, e.g., Input.txt

```
sum 3 5
sum 8 2 2
```

2. ./a.out $(awk 'NR==1' input.txt) or ./a.out $(sed '1!d' input.txt)

Linux command:

- **awk 'NR==1' input.txt:** it returns the line number 1 of the input.txt file
- **sed '1!d' input.txt :** it returns the line number 1 of the input.txt file
- **cmd1 $(cmd2):** the command cmd2 is executed and the output is passed to the command cmd1 as parameter

Stack– exercise 2

A stack is a data structure which allows to access data in a LIFO fashion (Last-In First-Out)

A stack can be used using two functions:

- *push(number),* which pushes a number (i.e., element) in the stack
- *pop(),* which pops a number from the stack (or return an error if the stack is empty)

Implement a program that presents to the user a menu of the possible functions that can be performed in the stack, and a loop to control the continuous execution of such functions on the stack

Example of an execution:

```
push(3)
push(4)
pop()
4
pop()
3
```

RPN Calculator – exercise 3

Implement a program, rpn, which takes as arguments an expression in RPN (Reverse polish notation) and outputs its result.

RPN: operands first and then operators.

For instance: *2 3 +*

Examples

- rpn 2 3 + 5 +
  - result → 10
- rpn 10 10 10 + +
  - result → 30
- rpn 2 3 + 5 *
  - result → 25
- rpn 10 + 10
  - result → ERROR

RPN Calculator – exercise 3

Implement a program, rpn, which takes as arguments an expression in RPN (Reverse polish notation) and outputs its result.

RPN: operands first and then operators.

For instance: *2 3 +*

Examples

- rpn 2 3 + 5 +
  - result → 10
- rpn 10 10 10 + +
  - result → 30
- rpn 2 3 + 5 *
  - result → 25
- rpn 10 + 10
  - result → ERROR

Note (opt.): experiment with redirection, for instance by:

1. preparing a file "input.txt" in which each line contains input for calculate, e.g., Input.txt

```
2 3 + 5 +
10 10 10 + +
2 3 + 5 *
10 + 10
```

2. `./a.out $(awk 'NR==1' input.txt)` or `./a.out $(sed '1!d' input.txt)`

RPN: Hint – exercise 4

Use a stack and the following strategy:

- When you find a number, push the number in the stack
- When you find an operator (e.g., +):
  - Pop the appropriate number of operands from the stack
  - Perform the operation
  - Push the result in the stack

Data Analysis (1)

We have a file with data about the population of big US cities.

The file is structured in fields, each field separated by a tab and the fourth field and fifth fields are the population in 2016 and 2010

Constraint: as input file use: us_cities.csv

Data Analysis (2) - Exercises

- Ex.5.1 We want to sum the column of the population in 2016
- Ex.5.2 We want to sum the columns of the population in 2016, in 2010 and calculate the growth percentage
- Ex.5.3 We want to read population in 2016, the size in km^2 and compute the density (inhabitants per km^2)

Suggestions

- First, try to use >> or getline
- Then, try to use the find function to find the separators and the fields (more complex, more general)

Find

- str.find(const char* *str2*, size_t *pos* = 0): it searches the first occurrence of the string *str2* in the string str. When *pos* is specified, the search only includes characters at or after position *pos*, ignoring any possible occurrences that include characters before *pos*.

Return value

- The position of the first character of the first match.
- If no matches were found, the function returns *string::npos*.

https://cplusplus.com/reference/string/string/find/

Data Analysis (2) - Exercises

- Ex.5.1 We want to sum the column of the population in 2016
- Ex.5.2 We want to sum the columns of the population in 2016, in 2010 and calculate the growth percentage
- Ex.5.3 We want to read population in 2016, the size in km^2 and compute the density (inhabitants per km^2)

Suggestions

- First, try to use >> or getline
- Then, try to use the find function to find the separators and the fields (more complex, more general)

Base: Write the output to cout
Variant 1: Write the output to three files, one for exercise
Variant 2: Write the output to one file and append the output, of each exercise, as a new column to the file

https://cplusplus.com/reference/string/string/find/

Data Analysis (2) - Exercises

- Ex.5.1 We want to sum the column of the population in 2016
- Ex.5.2 We want to sum the columns of the population in 2016, in 2010 and calculate the growth percentage
- Ex.5.3 We want to read population in 2016, the size in km^2 and compute the density (inhabitants per km^2)

Suggestions
- First, try to use >> or getline
- Then, try to use the find function to find the separators and the fields (more complex, more general)

Variant 3: Implement it as a pipe (i.e., sequence), using the cut command (available in Linux) and the sum command (your program)

Linux cut: it extract the content of some specific field form a stream/file, e.g.,
- cut -d " " -f 2
- -d “char” :defines the character to be used to delimit the fields
- -f <num> :defines the field to get, e.g., -f 2 means get the second field

https://man7.org/linux/man-pages/man1/cut.1.html
https://cplusplus.com/reference/string/string/find/

Data Analysis (2) - Exercises

- Ex.5.1 We want to sum the column of the population in 2016
- Ex.5.2 We want to sum the columns of the population in 2016, in 2010 and calculate the growth percentage
- Ex.5.3 We want to read population in 2016, the size in km^2 and compute the density (inhabitants per km^2)

Suggestions
- First, try to use >> or getline
- Then, try to use the find function to find the separators and the fields (more complex, more general)

Variant 4: Change the cut command form Linux with a simplified program implemented to extract one column from a file in which each line has fields delimited by a character.

```
cut –f<number> -d<character>
```

https://cplusplus.com/reference/string/string/find/