Information Processing 2 — Week 7 exercises

Complete the exercises in-class on your own laptop. You do not need to submit the answers online. Instead, ask an instructor to check your work during the class as soon as you finish answering each question. If you do not finish all the questions in-class, try to finish them outside class and have them checked *at the start of the next exercises class*.

Note: Exercises checked **more than one week** after the class will have a **50% late penalty** applied.

Note: for each exercise you can download a 'template' from Teams that contains the test program shown in this document.

Mini project: a Reverse-Polish Notation calculator

This week's exercises guide you through the creation of a RPN calculator. You will develop a function to read floating-point values or operators, a stack data structure storing `double`s, and a program that uses them together to implement a calculator.

1 Reading floating-point numbers [2 points]

You have already written a function to convert a text representation of an integer into its value.

```
int getint(void)
{
    int value = 0, c;
    while (isdigit(c = getchar()))
        value = value * 10 + c - '0';
    return value;
}
```

Modify this function to create `int getnum(void)` which converts a text representation of a floating-point into its `double` value and stores the result in a global variable `double number`. (You can ignore negative sign, decimal point, and fractional part for now.) `getnum()` should return `-1` if it detects `EOF`, or any other value to indicate success.

Test it by typing a few numbers into this program (download '`01-getnum.c`' from Teams):

```
int main()
{
    while (getnum() != -1) printf("%g\n", number);
    return 0;
}
```

▷▷ Ask an instructor to check and record your work.

2 Reading numbers or operators [2 points]

Modify `getnum()` to make a new function `int getop(void)` that:

1. skips over any blank characters (according to `isblank()`) and then **either**

2a. reads `EOF` and returns `-1`, **or**

2b. reads a newline `'\n'` and returns 0, **or**

2c. reads a floating-point number beginning with a decimal digit and returns `'0'`, **or**

2d. reads a single non-digit character and returns that character.

In the case of a floating-point number, `getop()` will have to return `'0'` **and** report the value of the number that it read. The easiest way to do this is to store the `double` value in a global variable (perhaps called `number`). The `getop()` function should therefore behave as follows:

- read a character `c`
- while `c` is a blank (according to `isblank(c)`) read another character `c`
- if `c` is `EOF`, return −1
- if `c` is a newline, return 0
- if `c` is *not* a digit, return `c` as the result
- read a floating-point number into `number` and return `'0'`.

Test your function with the following program (download '`02-getop.c`' from Teams):

```
int main()
{
    for (;;) {
        int c = getop();
        switch (c) {
            case -1:    printf("EOF\n");                        return 0;
            case  0:    printf("newline\n");                    break;
            case '0':   printf("%g\n", number);                 break;
            default:    printf("unknown operator '%c'\n", c);   break;
        }
    }
    return 0;
}
```

and input consisting of empty lines, integers, and characters representing arithmetic operators. (It should be obvious whether or not the output is correct.)

▷▷ Ask an instructor to check and record your work.

3 LIFO stack of doubles [2 points]

Use an array `double stack[32]` to implement a stack of up to 32 `double`s. The interface to the stack should be a pair of functions:

`void push(double d)`	pushes the value `d` onto the stack, and
`double pop(void)`	pops the topmost value off the stack and returns it.

Store the current depth of the stack in a global variable `int depth` (initialised to 0).

Test your stack with the following program (download '`03-stack.c`' from Teams):

```
int main()
{
    push(1.0);
    push(2.0);
    push(3.0);
    printf("%g\n", pop());
```

```
    printf("%g\n", pop());
    push(4.0);
    push(5.0);
    printf("%g\n", pop());
    printf("%g\n", pop());
    printf("%g\n", pop());
    return 0;
}
```

Verify that the output is: `3 2 5 4 1`

▷▷ Ask an instructor to check and record your work.

4 Calculator for addition [2 points]

Add `push()` and `pop()` from question 3 to the program of question 2. Modify the `main()` function to finish the implementation of your calculator:

- in `case '0':` instead of printing the `number`, `push()` it onto the stack,
- in `case 0:` repeatedly `pop()` and print the items on the stack until the stack is empty, and
- add a new `case '+':` that pops the `rhs` operand, pops the `lhs` operand, and then pushes the sum of `lhs + rhs` to perform the operation.

Test your program by entering '`1 2 3 4 +++`' and verify that the output is `10`.

▷▷ Ask an instructor to check and record your work.

5 Calculator for basic arithmetic [1 point]

Add three more `case` labels to your program so that it can evaluate subtraction ('`-`'), multiplication ('`*`'), and division ('`/`').

Test your program by evaluating: `4 6 8 * + 20 2 / -`

▷▷ Ask an instructor to check and record your work.

6 Reading the fractional part of floating-point numbers [1 point]

Extend `getop()` so that it understands an optional decimal point and fractional part of a floating-point number.

Test your program with: `10000000000 0.0000000001 * 1 -`

(What happens if you add an extra zero in the middle of each of the two long numbers? Why?)

▷▷ Ask an instructor to check and record your work.

Challenge

7 Fix the limitations of the calculator [1 bonus point]

Fix the following problems:

- If you have not already done so, handle stack underflow (popping when `depth` is zero) and overflow (pushing when `depth` is 32).
- Handle negative numbers. (A '−' immediately before a digit can begin a floating-point number. A '−' in any other context is the subtraction operator.)
- Implement a floating-point modulus operator '%'.
- You have to press enter twice to print a number with no operators, which is a secondary effect of this problem…
- The character following a floating-point number is currently discarded. Write your own version of `getchar()` called `getch()` that reads input characters. At the end of reading a floating point number, when a non-digit is encountered, save that character (perhaps using another function that you write, `ungetch(int c)`) and return the saved character the next time `getch()` is called.

When you have fixed these problems, the following input should work correctly and safely:

`+`	prints 'stack underflow' twice, then leaves a 0 on the stack
`-42`	prints '−42'
`1 2+`	prints '3'
`1-2+`	prints '−1'
`22 7 /`	prints '3.14286'
`22 7 %`	prints '1'

▷▷ Ask an instructor to check and record your work.