The following are the compulsory assignments from the book (plus intopt):

- RPN : 1.5
- Poly: 1.7
- Freq: 1.9

1.5 Implement an **RPN calculator**, which reads one character at a time from `stdin` using `getchar`, see Section 13.20.7.6, page 519. Your program should handle integers and the *binary* operators `+`, `-`, `*`, and `/`. Your program should process one line at a time and print the (only) value on the stack when a new-line character is seen. Your stack should have space for exactly 10 numbers and your program must check for the following errors:

- invalid character (such as !),
- no space left on your stack when you need to push a number,
- zero or one number on the stack when you need to pop two numbers for a binary operator,
- divide by zero,
- an empty stack when you see a new-line character,
- and two or more numbers when you see a new-line character.

You do not have to consider overflow. When you encounter an error, print an error message and skip the rest of that line.

You should terminate the program by typing ctrl-D at the beginning of a new line. In UNIX ctrl-D means end of file, or EOF. If we first push x and then y and next see $-$, we should calculate $x - y$, i.e. the input `2 3 -` should result in the output `-1`. At a new-line character, print output such as: `line 1: 42`.

Hint 1: The function `getchar` returns an `int` and you should *not* store the value read in a `char` since a char may be an unsigned integer and the return value `EOF` normally is a negative number which typically will lead to an infinite loop. The source code that can be downloaded from the book's site has a makefile which specifies that `char` should be an unsigned number (in order to test your code for this error).

Hint 2: To print the two characters `\n` you can use a string `"\\n"`.

Hint 3: To check if a character is a number, use `isdigit`, see Section 13.3.1.5, page 433, and to check if it is e.g. a plus sign, use the constant `'+'` (using the ASCII number `43` in source code is not portable and generally a bad idea).

Hint 4: In this exercise, there is no need to use pointers.

See `rpn.zip` at the book's site. Implement you calculator in a file `rpn.c` and type `make`, This will test your program with the input below, which should result in the output found on the next page.

```
2 3 +
4 5 -
124 1000 * 36 +
6 7 8 9*+-
60 4 /
```

```
2 1000 * 10 5 + + 19 100 * 8 10 * 5 + + -
1 2 3 4 5 6 7 8 9 10 +++++++++
1 2

1 0 /
1 2 3 4 1 2 3 4 5 6 7 ++++++++++
1 +
!
```

Expected output:

```
line 1: 5
line 2: -1
line 3: 124036
line 4: -73
line 5: 15
line 6: 30
line 7: 55
line 8: error at \n
line 9: error at \n
line 10: error at /
line 11: error at 7
line 12: error at +
line 13: error at !
```

1.6 Write a program which takes an integer argument n from the command line and prints out the exact value of $n!$, for any $1 \leq n \leq 100$.

1.7 Write a program which can multiply polynomials. Your program should be able to multiply polynomials such as the following:

$$(x^2 - 7x + 1) \times (3x + 2)$$
$$(x^{10000000} + 2) \times (2x^2 + 3x + 4)$$

The following header file declares the **opaque** type `poly_t` as an incomplete struct type. You need to define it in a file `poly.c`. See `poly.zip` at the book's site.

```
#ifndef poly_h
#define poly_h

typedef struct poly_t poly_t;

poly_t*         new_poly_from_string(const char*);
void            free_poly(poly_t*);

poly_t*         mul(poly_t*, poly_t*);

void            print_poly(poly_t*);
```

```
#endif
```

Use a test program such as the following:

```
#include <stdio.h>

#include "poly.h"

static void poly_test(char* a, char* b)
{
        poly_t*         p;
        poly_t*         q;
        poly_t*         r;

        printf("Begin polynomial test of (%s) * (%s)\n", a, b);

        p = new_poly_from_string(a);
        q = new_poly_from_string(b);

        print_poly(p);
        print_poly(q);

        r = mul(p, q);

        print_poly(r);

        free_poly(p);
        free_poly(q);
        free_poly(r);

        printf("End polynomial test of (%s) * (%s)\n\n\n", a, b);
}

int main(void)
{
        poly_test("x^2 - 7x + 1", "3x + 2");
        poly_test("x^10000000 + 2", "2x^2 + 3x + 4");

        return 0;
}
```

The output from this program should be:

```
Begin polynomial test of (x^2 - 7x + 1) * (3x + 2)
x^2 - 7 x + 1
3 x + 2
3 x^3 - 19 x^2 - 11 x + 2
End polynomial test of (x^2 - 7x + 1) * (3x + 2)
```

```
Begin polynomial test of (x^10000000 + 2) * (2x^2 + 3x + 4)
x^10000000 + 2
2 x^2 + 3 x + 4
2 x^10000002 + 3 x^10000001 + 4 x^10000000 + 4 x^2 + 6 x + 8
End polynomial test of (x^10000000 + 2) * (2x^2 + 3x + 4)
```

1.8 Write a program which finds the longest word in its input (`stdin`). Here we define a word to be a sequence of characters for which `isalpha` returns a nonzero value. See `word.zip` at the book's site.

1.9 Write a program which reads input one line at a time from `stdin` and finds which word is most frequent. There is exactly one word per line, so you don't need to use any function like `isalpha`. For each line, the word should either be counted or deleted (or, equivalently, its count be set to zero),depending on if the current line number is a prime number or not. Input starts with line 1, so the first word is counted, and then at lines 2 and 3, these words are (tried to) be deleted. At the end the most frequent word should be printed as below. If there are multiple words with the same count, the lexicographically least is the one that should be printed (i.e., print `"a"` and not `"b"`).

In addition, the actions of the program should also be printed.

Given the input:

```
abc
abc
abc
abc
def
abc
xyz
xyz
xyz
```

The following should be printed:

```
added abc
trying to delete abc: deleted
trying to delete abc: not found
added abc
trying to delete def: not found
counted abc
trying to delete xyz: not found
added xyz
counted xyz
result: abc 2
```