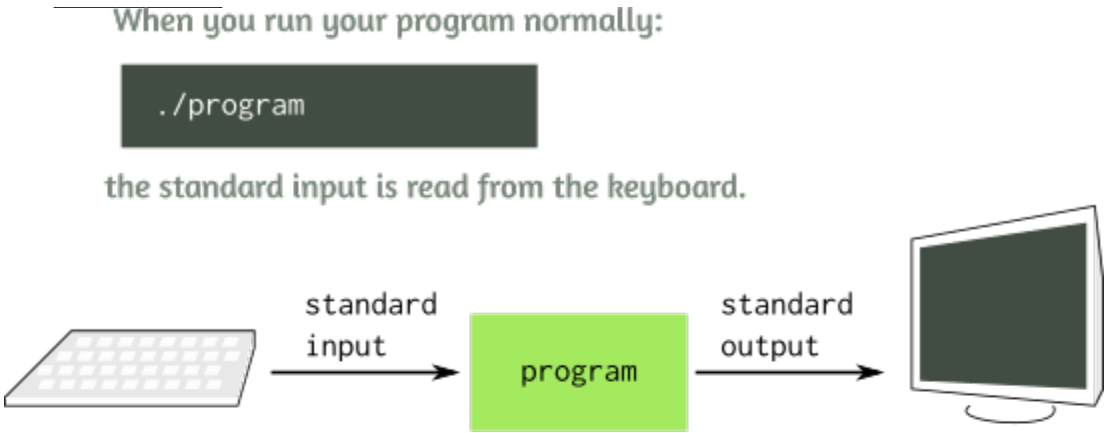
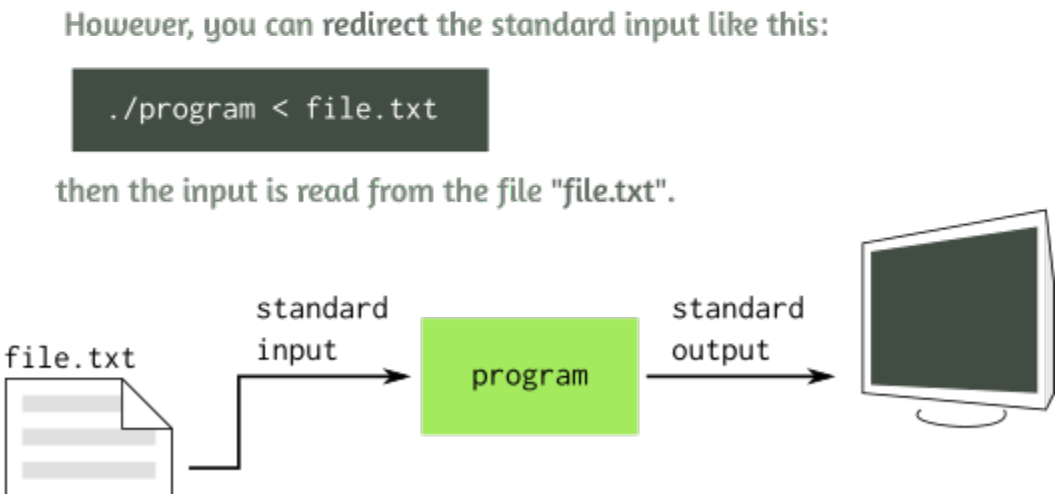
Project 1. Streams and calculators.

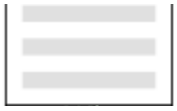
Introduction: Input redirection.

We have learned how to read user input from keyboard, and we will learn how to use file streams to read data from files.

However, there is another simpler way to read data from a file – using a feature of Unix shell called **standard input redirection**. We are going to use it in this assignment.







In Unix, if we run the program as follows:

```
./program < myfile.txt
```

Every time you read from `cin`, such as in this statement

```
cin >> x;
```

the value `x` is read not from the keyboard, but from the text file `myfile.txt` you specified. Isn't it neat? It is called file redirection. The text from the file is redirected character by character as the standard input for your program.

To read the full contents of the file *word by word*, you can write a program:

```
#include <iostream>
using namespace std;
int main() {
    string s;
    while(cin >> s) { // While the reading operation is a success
        cout << s << endl;  // print the read word
    }
}
```

It relies on the fact that the expression `cin >> s` does not only read a string into the variable `s`, but also itself evaluates to `true`, if the reading operation was a success, and to `false`, if it was a failure. Practically, it means that **when the program reaches the end of the file**, the operation `cin >> s` fails to read anything from the file, evaluating to `false` and indicating that the loop should stop.

Task A. Adding integers.

Write a program `sum.cpp` that reads a sequence of integers from `cin`, and reports their **sum**.

Example

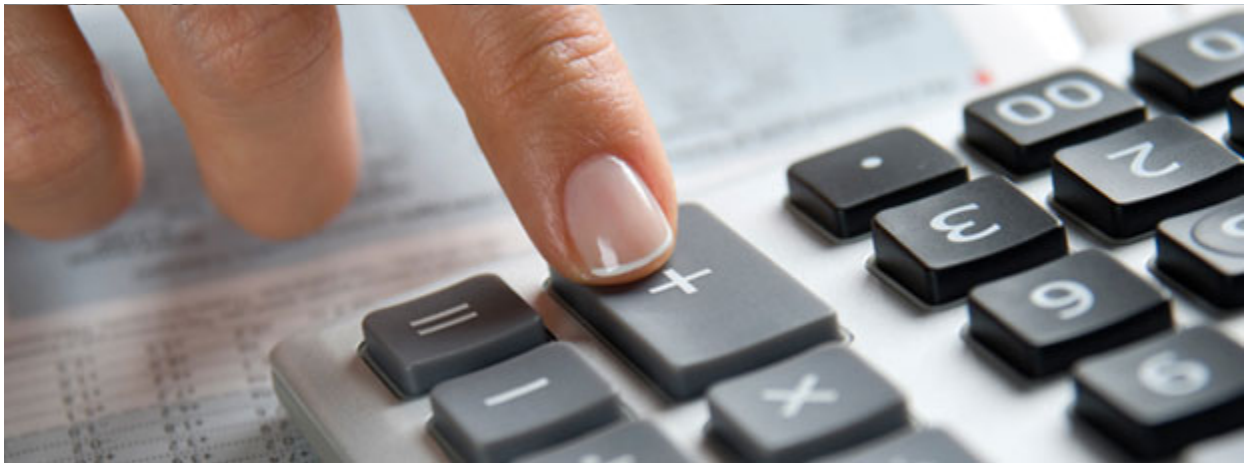
If you have a file `numbers.txt` that contains:

```
10 15 16 -7 102 345
```

then if you redirect it into the program, it should report:

```
$ ./sum < numbers.txt
481
```

Task B. Calc: A calculator program.



We want to make a **simple calculator that can add and subtract integers**, and will accept

arbitrarily long mathematical formulas composed of symbols `+`, `-`, and non-negative integer numbers.

Imagine you have a file `formula.txt` with the summation formula such as:

```
100 + 50 - 25 + 0 + 123 - 1
```

If you redirect the file into your program, it should compute and print the answer:

```
$ ./calc < formula.txt
247
```

It may sound tricky, but it is actually easy to write such a program, and you already know all the needed tools. Just think carefully how to put it all together.

Specifically, write a program `calc.cpp` that reads from the `cin` a sequence of **one or more non-negative integers** written to be **added or subtracted**. Space characters can be anywhere in the input. After the input ends (end-of-file is reached), the program should compute and print the result of the input summation.

Possible input for your program may look like this:

```
15
```

```
10 + 3 + 0 + 25
```

```
5+6- 7 -8 + 9  + 10 -   11
```

```
1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 +
1 + 1 + 1 + 1 +
1 + 1 + 1 + 1
```

(Each of the inputs above is a separate file containing one single formula, even if it spans multiple lines.)
The corresponding outputs should be: `15`, `38`, `4`, and `16`.

A hint on how to handle possible space characters in the input:

You can use `>>` operator to read the numbers and the `+` / `-` characters, because `>>` will be skipping all spaces between the input terms. It is also suggested to use the `char` type for reading the `+` / `-` operator characters, not `string`, because it will work well even when numbers and the operator symbol are adjacent and not separated by spaces (such as in `10+5+3`).

Task C. Calc2: Reading multiple formulas.

Write a better version of the calculator, `calc2.cpp`, that can evaluate multiple arithmetic expressions. Let's use the semicolon symbol that must be used at the end of each expression in the input.

Assuming that the input file `formulas.txt` looks as follows:

```
15 ;
10 + 3 + 0 + 25 ;
5 + 6 - 7 - 8 + 9 + 10 - 11 ;
```

When we run the program with that input, the output should evaluate all of the expressions and print them each on its own line:

```
$ ./calc2 < formulas.txt
15
38
4
```

Task D. Calc3: Squares.

Write an even better calculator program `calc3.cpp` that can understand squared numbers. We are going to use a simplified notation `X^` to mean X^2. For example, `10^ + 7 - 51^` should mean $10^2 + 7 - 51^2$.

Example:

When reading input file `formulas.txt`

```
5^;
1000 + 6^ - 5^ + 1;
```

the program should report:

```
$ ./calc3 < formulas.txt
25
1012
```

A hint:

To take into account `^`, don't add or subtract new numbers right away after reading them. Instead, remember the number, read the next operator and if it is a `^`, square the remembered number, then add or subtract it.

An additional note on how to test calculator programs

In addition to writing your formulas into files, remember that your program still accepts the input from the keyboard (Hey, do you see the benefit of input redirection? The program can work great on both keyboard and file inputs!)

When typing the input from the keyboard, the key combination `Ctrl+D` emulates the *End-of-file* situation, telling the program that the input has ended.

So, you can test your program like this:

```
$ ./calc
15 - 4 + 13 - 2 + 1  <Enter> <Ctrl+D>
23
```

(finalizing your input by pressing `Enter` and `Ctrl+D`).