Quick tutorial: Programming with the lab server

Daniel Schmidt

Summer 2021

1 Some notes on the problem input and output

1.1 Passing your solution to the server

The server expects your solution in the *standard output*, i.e., it expects that you print your solution to the console or terminal window. Essentially, the standard output behaves like a (virtual) file with a fixed name; and indeed, in Java/C++/Python, it may be accessed through a file-like object, namely as `std::cout` in `C++`, as `System.out` in `Java`, and `sys.out` in `Python`. The `print` statement in `Python` will by default print to the standard output. We will see examples for all three programming languages below.

1.2 Reading your solution from the server

The competitive programming lab server passes the problem input via another virtual file: The *standard input*. The standard input is called `std::cin` in `C++`, `System.in` in `Java`, and `sys.in` in `Python`. In particular, this means that we will not use the standard command line parameters (e.g., `sys.argv` in `Python`, or `argv` in `C++`) to access the problem input.

Here's what to expect from the problem input on the lab server.

1. line endings: It is recommended that your code can cope with both Unix (\n) and Windows (\r\n) file endings. However, as a general rule, the input in the lab has Unix file endings.

2. end-of-file: the problem input may or may *not* have a trailing `newline` character, i.e. the input may end with an empty line. Your code should always be able to cope with both cases.

3. You may always assume that the input is well-formed (i.e., that it matches the specification). You will not be given empty input. If you suspect malformed input, please let me know and I will have a look.

4. One more thing to be aware of: Some built-in functions stop when the end-of-file character `EOF` is the next to be read. Some built-in functions stop only after `EOF` has been read. There's no reason to prefer one over the other, but the calling code needs to match the built-in function. It is a common source of errors that it does not!

The following text briefly discusses how to read input on the server and shows how you may output your solution. Throughout, let's assume that we want to solve the following example problem:

input The input consists of lines. Each line $i = 1, \ldots, n$ contains exactly one integer number a_i.

output The sum $\sum_{i=1}^{n} a_i$ of the input numbers.

Let us also assume that the problem input (the testcase) is given in a file called `sample.in`.

```
daniel:~/lab> cat sample.in
8
2
4
6
9
```

The expected output for this sample is 29. Suppose the server has created an executable program `linesum` from your source code. The programm will then be called in the following way:

```
daniel:~/lab>./linesum < sample.in
29
```

Thus, the program produced the correct output – good! If the server has several test cases, it will make one independent call of your program for each testcase. In order to debug your program, you may call it on your local machine in the same way the server does.

Finishing remark: Passing the input file as a command line parameter will *not work*:

```
daniel:~/lab>./linesum sample.in
```

simply shows a cursor while the program waits for input. If we wanted to, we could now manually type in the desired input via the keyboard and finish our input with the `EOF` character by pressing `CTRL+D`. For your program, both methods will look identical (i.e., your program is not aware of the filename containing the input, nor does it know if the input came from a file in the first place).

2 Reading from standard input with C++

2.1 Using stream operators

Let's write a program that solves our example problem in `C++`. We can use an `istream` object called `std::cin` to read input from the standard input. The object supports stream operations and has overloaded operators for all basic types. One seemingly elegant way of reading from `stdin` is this:

Listing 1: stdin1.cpp

```
#include <iostream>

int main(int argn, char** argv){
  int sum = 0;
  while(std::cin){
    int number;
    std::cin >> number;
    sum += number;
  }

  std::cout << sum << std::endl;

  return 0;
}
```

Surprisingly, the output of the program is not correct:

```
daniel:~/lab> g++ -o linesum stdin1.cpp
daniel:~/lab> ./linesum < sample.in
38
```

This is because the stream `std::cin` only becomes invalid *after* `EOF` has been read and hence the while loop makes an additional iteration after the last number has been read from the input.

We may solve this problem by checking whether we have reached the end of the file:

Listing 2: stdin2.cpp

```
#include <iostream>

int main(int argn, char** argv){
  int sum = 0;
  while(!std::cin.eof()){
    int number;
    std::cin >> number;
    sum += number;
  }

  std::cout << sum << std::endl;

  return 0;
}
```

However, this code fails as well if the input contains a trailing new line (like in this case).

```
daniel:~/lab> g++ -o linesum stdin2.cpp
daniel:~/lab> ./linesum < sample.in
38
```

We can avoid both issues with the following code:

Listing 3: stdin3.cpp

```
#include <iostream>

int main(int argn, char** argv){
  int sum = 0;
  int number;
  while(std::cin >> number){
    sum += number;
  }

  std::cout << sum << std::endl;

  return 0;
}
```

Here, the while-loop stops once no more integer can be extraced from `std::cin`.

```
daniel:~/lab> g++ -o linesum stdin3.cpp
daniel:~/lab> ./linesum < sample.in
29
```

Final remark: The beauty of this method is that it will read any white-space separated input. For instance, if the numbers were separated by `space` instead of `newline`, our code would still work. The downside of the methode is precisely that it will read any white-space separated input; often, it will fail silently if the input is malformed.

2.2 Using `std::getline(...)`

The STL includes the built-in method `std::getline` for reading files line by line, as well as methods to convert strings to basic data types (e.g., `std::stoi` and `std::stod`).

Listing 4: stdin4.cpp

```
#include <iostream>
#include <string>

int main(int argn, char** argv){
  int sum = 0;
  std::string line;
  while(std::getline(std::cin, line)){
    sum += std::stoi(line);
  }

  std::cout << sum << std::endl;

  return 0;
}
```

Instead of using string conversion methods, each line can be made into a stream object to be parsed with stream operators.

Listing 5: stdin5.cpp

```
#include <iostream>
#include <string>
#include <sstream>

int main(int argn, char** argv){
  int sum = 0;
  std::string line;
  while(std::getline(std::cin, line)){
    std::stringstream line_stream;
    line_stream << line;
    int number;
    std::cin >> number
    sum += number;
  }

  std::cout << sum << std::endl;

  return 0;
}
```

Even though the name suggests otherwise, the `std::getline` method is not fixed to using `newline` characters as separators; in fact, any character may be used by passing it via the third, optional parameter of the method. For instance, if we were to use comma separated values

```
daniel:~/lab> cat sample.csv
8,2,4,6,9
```

we could use this code:

Listing 6: stdin6.cpp

```
1  #include <iostream>
2  #include <string>
3
4  int main(int argn, char** argv){
5    int sum = 0;
6    std::string line;
7    while(std::getline(std::cin, line, ','){
8      sum += std::stoi(line);
9    }
10
11   std::cout << sum << std::endl;
12
13   return 0;
14 }
```

The `std::getline` method expects a character as the third parameter and hence, single-quotes are needed.

```
daniel:~/lab> g++ -o linesumcsv stdin6.cpp
daniel:~/lab> ./linesumcsv < sample.csv
29
```

3 Reading from standard input in python

In Python, the standard input is called `sys.stdin` and is available once `sys` has been imported. You may pass file content to the standard input of a python program like so:

```
daniel:~/lab> python3 linesum.py < sample.in
29
```

A single line may be read with `sys.stdin.readline()` so that we may emulate the pattern from the C++ examples.

Listing 7: stdin1.py

```
1 import sys
2
3 sum = 0
4 while sys.stdin:
5   sum += int(sys.stdin.readline())
6
7 print(sum)
```

However, just as in the C++ example, this code fails if the file has a trailing `newline`. In this case, the code will read an extra empty line in the last iteration of the loop.

```
daniel:~/lab> python3 stdin1.py < sample.in
Traceback (most recent call last):
  File "~/lab/stdin1.py", line 5, in <module>
    sum += int(sys.stdin.readline())
ValueError: invalid literal for int() with base 10: ''
```

This may be avoided by extra error checking. Alternatively, we may loop over the lines of the file with a for-loop.

Listing 8: stdin2.py

```
1 import sys
2 print(sum([int(line) for line in sys.stdin]))
```

```
daniel:~/lab> python3 stdin2.py < sample.in
29
```

Both techniques may be combined.

Listing 9: stdin3.py

```
1 import sys
2
3 sum = int(sys.stdin.readline())
4 for line in sys.stdin:
5   sum += int(line)
6
7 print(sum)
```

```
daniel:~/lab> python3 stdin3.py < sample.in
29
```

4 Reading from standard input in Java

The standard input is called `System.in` in Java. File content may be passed to `System.in` as follows:

```
daniel:~/lab> java LineSum < sample.in
29
```

For easy parsing, the `System.in` object may be combined with the Scanner class.

Listing 10: LineSum.java

```
1  import java.util.Scanner;
2
3  public class LineSum {
4    public static void main(String[] args) {
5        Scanner scanner = new Scanner(System.in);
6
7        int sum = 0;
8        while(scanner.hasNextInt()){
9          sum += scanner.nextInt();
10       }
11
12       System.out.println(sum);
13   }
14 }
```

```
daniel:~/lab> javac LineSum.java
daniel:~/lab> java LineSum < sample.in
29
```

If performance is a concern, the `readLine()` method of a `BufferedReader` object may be used.

Listing 11: LineSumReader.java

```
1  import java.io.IOException;
2  import java.io.BufferedReader;
3  import java.io.InputStreamReader;
4
5  public class LineSumReader {
6    public static void main(String[] args) throws IOException {
7        BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
8        int sum = 0;
9        while(bf.ready()) {
10         sum += Integer.parseInt(bf.readLine());
11       }
12
13       System.out.println(sum);
14   }
15 }
```

```
daniel:~/lab> javac LineSumReader.java
daniel:~/lab> java LineSumReader < sample.in
29
```

5 Performance considerations

5.1 Controlling buffering in C++

The C++ environment flushes the output buffer when `std::endl` is called. Hence, if you want to avoid forcing a flush, output `\n` instead (although this will still flush the buffer when it is full). In any case, the output buffer may be manually flushed by calling `std::cout.flush()`. The performance of the input stream may be further improved by calling

Listing 12: iosync.cpp

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Without these calls, the C-style input/output via `printf` and `scanf` is faster than using `std::cout` and `std::cin`. Buffering of `printf` and `scanf` may be disabled in the following way.

Listing 13: iosync.cpp

```
setvbuf(stdin, NULL, _IONBF, 0);
setvbuf(stdout, NULL, _IONBF, 0);
```

5.2 Controlling buffering in Java

Buffer size and auto-flushing may be controlled as follows in Java.

Listing 14: Unbuf.java

```
import java.io.BufferedOutputStream;
import java.io.FileDescriptor;
import java.io.FileOutputStream;
import java.io.PrintStream;

class Unbuf {
  private static void main(String[] args) {
    FileOutputStream fdout = new FileOutputStream(FileDescriptor.out);
    // Set buffer size to 1024 bytes
    BufferedOutputStream outBuf = new BufferedOutputStream(fdout, 1024);
    // false -> disable automatic flushing on endline
    PrintStream outStream = new PrintStream(outBuf, false);
    System.setOut(outStream);
  }
}
```

Furthermore, the following pattern improves output performance significantly.

Listing 15: FastOut.java

```
import java.io.BufferedOutputStream;

class FastOut {
  private static final BufferedOutputStream out
      = new BufferedOutputStream(System.out);

  public static void main(String[] args) {
    // notice manual insertion of newline and conversion of string to bytes.
    out.write("my output\n".getBytes());
    ...
    // no output is produced without these lines.
    out.flush();
    out.close();
  }
}
```

6 Profiling

A code profiler may be used to analyze the performance and the memory use of your code.

6.1 Profiling `Java` code

Profiling of Java code may be done with integrated profiler of `Netbeans`. In order to read the input in the profiled program, replace `Scanner scanner = new Scanner(System.in)` by `Scanner scanner = new Scanner(new File("myfile.in"))`.

6.2 Profiling `C++` code

Code in `C++` may be profiled with the `perf` tool from `linux-tools`. First, compile your executable with debugging symbols. Then, run `perf` as follows. This creates an output file `perf.data` that may be visualized with `hotspot` (see caller/callee tab for quick timing analysis).

```
daniel:~/lab> g++ -g -o linesum stdin5.cpp
daniel:~/lab> perf record ./linesum < sample.in
daniel:~/lab> ls -l perf.data
-rw------- 1 daniel daniel 21k 21. Mai 14:12 perf.data
daniel:~/lab> hotspot perf.data
```

As an alternative, the code may be profiled with `callgrind`. This again requires compiling with debugging symbols.

```
daniel:~/lab> g++ -g -o linesum stdin5.cpp
daniel:~/lab> ./linesum < sample.in
```

Then, the code is run with the `callgrind` tool.

```
daniel:~/lab> valgrind --tool=callgrind ./linesum < sample.in
```

This in turn produces a file `callgrind.out.<pid>` which is human-readable, but not very informative.

```
daniel:~/lab> ll callgrind.out.*
-rw------- 1 daniel daniel 98k 21. Mai 13:57 callgrind.out.426071
```

The command `callgrind_annotate` may now be used to extract more readable information.

```
daniel:~/lab> callgrind_annotate --show-percs=yes callgrind.out.426071 > profile.txt
```

The `profile.txt` file now contains summarized information at the top and detailed annotated source code below. Search for the name of your source file to quickly find the relevant information. The graphical tool `kcachegrind` makes browsing the information much easier.

A classical alternative is the gnu profiler `gprof`. Here, the executable has to be compiled *and* linked with the `-pg` flag of `gcc`. The executable is then run normally and due to the `-pg` flag, it will now dump runtime information into an output file called `gmon.out`. The output file is not human readable, but the tool `gprof` may be used to extract performance information.

```
daniel:~/lab> g++ -pg -o linesum stdin5.cpp
daniel:~/lab> ./linesum < sample.in
daniel:~/lab> ls -l gmon.out
-rw-r--r-- 1 daniel daniel 8,9k 21. Mai 13:07 gmon.out
daniel:~/lab> gprof median gmon.out > profile.txt
```

6.3 Profiling `python` code

To profile `Python` code, you may use the inbuilt profiler as:

```
daniel:~/lab> python -m cProfile -s tottime stdin3.py < sample.in
```

This will print profiling information in the console window.

7 Debugging

7.1 Finding good test cases

A good approach is to come up with test cases where the answer may be trivially verified. In the case of our summation example, good initial testcases are inputs that solely consists of zeroes or ones. Likewise, inputs containing all integers from 1 to n in random order make for outputs that are easily verified with Gauss' summation formula $\sum_{i=1}^{n} i = n(n-1)/2$. Such an input could for instance reveal that the sum of all input numbers easily exceeds the maximum representable `int` and that our variable `sum` should be of type `long` or similar. Another possibility would be to generate a random sequence of numbers and to include it twice: Once with a positive, once with a negative sign.

7.2 Avoiding mistakes

7.3 Using `C++` (more) safely

Catching errors early is particularly tricky in `C++` as the language is very forgiving by default. It is possible, however, to enable a tighter security net. To that aim, consider enabling additional compiler warnings, for instance by using the following alias:

```
alias g++-safe='g++ -Wall -Wextra -Wnon-virtual-dtor -Wconversion
  -Winline -Wshadow -Wunreachable-code -Wuninitialized -Wswitch-default -Wundef
  -Wfloat-equal -Wreturn-type -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC'
```

Then, when compiling with `g++-safe`, the compiler will point to possible problems. Additionally, the flags `-D_GLIBCXX_DEBUG` and `-D_GLIBCXX_DEBUG_PEDANTIC` enable error checks in the `g++` implementation of the STL (in particular, the flags make `g++` STL containers throw exceptions if accessed out of bounds).

For instance, the following code looks innocent:

Listing 16: abs.cpp

```
1 #include <iostream>
2 #include <cmath>
3
4 int main() {
5   std::cout << abs(23 / 2.0) << std::endl;
6   return 0;
7 }
```

Even though the code compiles without issues, it produces an unexpected output.

```
daniel:~/lab> g++ -o abstest abs.cpp
daniel:~/lab> ./abstest
11
```

Compiling with the above additional warnings alerts us to the problem.

```
daniel:~/lab> g++-safe -o abstest abs.cpp
abs.cpp: In function 'int main()':
abs.cpp:5:24: warning: conversion from 'double' to 'int' changes value from '1.15e
    +1' to '11' [-Wfloat-conversion]
```

Further investigation then reveals that `abs` takes and returns an integer value[1].

The following example demonstrates why additional exceptions are useful.

Listing 17: exception.cpp

```
1 #include <iostream>
2 #include <vector>
3
4 int main(){
5   std::vector<int> vec = {1,2,3,4,5};
6   vec[5] = 42;
7
8   std::cout << vec[5] << std::endl;
9
10  return 0;
11 }
```

The code leads to undefined behaviour as we access `vec[5]` in lines 6 and 8 even though only `vec[0],...,vec[4]` have been initialized. The code compiles and runs without errors.

```
daniel:~/lab> g++ -o exctest exception.cpp
daniel:~/lab> ./exctest
42
```

[1] We would need to use `fabs` or `std::abs` instead

Compiling with `-D_GLIBCXX_DEBUG` and `-D_GLIBCXX_DEBUG_PEDANTIC` reveals the error once the code is executed:

```
daniel:~/lab> g++-safe -g -o exctest exception.cpp
daniel:~/lab> ./exctest
/usr/include/c++/10.2.0/debug/vector:427:
In function:
...

Error: attempt to subscript container with out-of-bounds index 5, but
container only holds 5 elements.

Objects involved in the operation:
    sequence "this" @ 0x0x7ffd1c159610 {
      type = std::__debug::vector<int, std::allocator<int> >;
    }
Aborted (core dumped)
```

Here, we have compiled with debugging symbols (the `-g` flag) to get human readable method and type names. Another option to catch illegal accesses to memory is the `valgrind` tool.

```
daniel:~/lab> g++ -g -o exctest exception.cpp
daniel:~/lab> valgrind ./exctest
==489401== Memcheck, a memory error detector
==489401== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==489401== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==489401== Command: ./exctest
==489401==
==489401== Invalid write of size 4
==489401== at 0x109273: main (exception.cpp:6)
...
==489401==
==489401== Invalid read of size 4
==489401== at 0x10928A: main (exception.cpp:8)
...
42
==489401==
==489401== HEAP SUMMARY:
==489401== in use at exit: 0 bytes in 0 blocks
==489401== total heap usage: 3 allocs, 3 frees, 73,748 bytes allocated
==489401==
==489401== All heap blocks were freed -- no leaks are possible
==489401==
==489401== For lists of detected and suppressed errors, rerun with: -s
==489401== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Again, we compiled with `-g` for human readable output. We observe that `valgrind` detects both the illegal write access in line 6 *and* the illegal read access in line 8. Furthermore, it tells us that the data type involved in the illegal access has a size of 4 bytes. In the same way, `valgrind` will detect uninitialized variables and memory leaks.