The Role of Input-Output Management in Programming Education

HORVÁTH Győző

Abstract. In the introductory programming education, during coding input and output management often suppresses in proportion the essential parts of the code. A novice programmer is essentially given three tasks with the solution of the original problem: reading the input, processing the data and writing the output. This article attempts to explore the role of input and output in the code, and in some cases, how to make them easier to implement.

Keywords: programming education, methodology, input-output

1. Introduction

Programming is basically data processing: the final result must be produced from the initial values. Computers are called to help in the process of this calculation. This process can be examined from different aspects. Programming education focuses on how computer-assisted problem solving can be done systematically. After reading and understanding the text of the task, the first three of the steps of problem solving are important to us: 1) define and describe in a formal way *what* the task is (*specification*), 2) describe in an abstract language *how* to solve the problem (*algorithm*), and 3) to *implement* the former solution of the task in a specific programming language (*coding*). Breaking the problem into steps is an important methodological support in achieving the basic goal of introductory programming education, where we expect the student to

- understand the task;
- recognize the input and output data in the text;
- describe the structure of those data;
- formulate the relationship between input and output data (either textually or formally);
- break down the solution into elementary steps;
- implement the solution in a specific programming language;
- check the correctness of the solution.

Students in this process need to learn a lot of new knowledge: the formal language of the specification (although it can be replaced by textual or visual descriptions in some circumstances), the algorithm description language (Nassi–Shneiderman diagram, aka NSD, sentence-like description, pseudocode) and the specific programming language itself. The purpose of this article is not to consider how much this is necessary, how to simplify or remove it, but rather to consider the above process as given. It is very important not to burden students with unnecessary information in this learning process.

The other aspect of looking at programming as data processing is considering how the initial data gets into this process and how the results come out from the process to the user. The first operation is *reading the input values*, and the second one is *writing the output values*, and together they are called *IO operations*.

The IO operations are very important at the early stage of learning programming. It allows interaction with the computer, which brings the bare bone hardware closer to the student by giving it human attributes. With the help of those operations the student can start a conversation with the computer even if it is programmatically designed in advance. This is a big methodological advantage in programming education. In addition, without output operation the user would not know the result of the calculation. So, at least, outputting data is inevitable in programming, and fortunately is quite simple in most programming languages.

On the other hand, reading input can be a cumbersome process. The starting point of this article is the observation, that it is often found during coding – especially in the early stages –, that reading the input and writing the output (but mainly the former) require disproportionally more codes and programming language knowledge than the solution itself. Usually, the reading operation itself is simple, but the communication with the user, filtering out the user's accidental or intentional errors, and checking the sometimes complicated relations of preconditions, often leads to unnecessarily lot of codes, and requires the use of language elements that should not be introduced in the initial phase of programming. Many times, concepts such as input buffer, compiler switches, type conversions need to be explained to the students, for which they are not prepared enough, or these concepts unnecessarily take time and effort from the lesson. The primary goal in programming education is not how to communicate safely, understandably and politely with the user (those are important, but secondary goals), but how to represent the data in the problem and how to generate the output data from the input data, i.e. how to solve the problem. Beside this, the way of obtaining the data (standard input, file, database, network query) and writing the end result (standard output, file, etc.) can be considered as a separate task. In other words, when students work on the coding of a task, they suddenly have to solve three tasks: besides the original task, they have to carry out the input and output parts as well.

It is interesting that the literature has very few references regarding to this topic. Input and output come to the foreground when it is mentioned among best educational practices that input values should always be validated [1], but does not mention the complexity coming with it. In an other article, which defines its own educational programming language, a new abstract interface is defined to hide the original complexity of input operations [2].

In this article, we look at the role of input and output handling in computer-assisted problem solving, how we can make them easier to let the students focus on the real task and investigate the role of the whole input-output management. The problem is investigated in the context of introductory programming education in higher education, that is why the code examples are in C++, but the concepts of this article may be applied to lower levels of programming education as well.

2. The problem

Reading the input in a program may be exciting, necessary, and on its own it is usually quite simple in most of the programming languages. However, many times the attention shifts towards the details of the reading process from the aim of this operation, i.e. getting the initial data for the problem-solving part. Tasks like *"check the fulfilment of the precondition"*, or *"check the type of the input"*, or *"exit the program if something fails"*, or *"repeat the reading until it is good"* should be considered different and separate tasks, which usually require advanced concepts of programming. Getting those tasks too early may cause confusions in beginner students.

The input may be given in a file which is read from the standard input. The structure of the file sometimes allows easy reading operations, sometimes it makes it harder. Typical examples for the latter are reading until the end of the file, or numbers and texts in the same line (Figure 1), or using an unusual separator in a line (Figure 2).

```
Some text here 1 2 3
```

Figure 1: Input line with mixed content

```
Some text;1;Other text;3
```

Figure 2: Special separator in input line

Handling input may take a lot of time, may result a lot of lines of code, and complex reading logic may introduce errors which makes the reading process unreliable for the main problem-solving part.

3. Example task

Consider the following task, which serves as a reference for the rest of the article!

Task: Two positive numbers are given, calculate the sum of those numbers!

```
Input:          a∈ℕ, b∈ℕ
Output:         c∈ℕ
Pre-condition:  a>0 and b>0
Post-condition: c=a+b
```

Figure 3: The specification of the example task

The algorithm in its simplest form (NSD or pseudocode):

```
c:=a+b
```

Figure 4: The short version of the algorithm

The pseudocode form is usually written as a separate subroutine:[1]

```
Type TInt=Integer
Procedure Task(Const a, b: TInt, Var c:TInt)
    c:=a+b
Procedure end
```

Figure 5: Longer version of the algorithm

The corresponding C++ code with the simplest reading of input is short as possible, but the reading part may result a lot of lines of code in more complex cases (see Appendix 1):

[1] The first line of the algorithm indicates that the simple and complex types in the specification can be defined before the procedure. Usually complex types are defined separately for later references, but there is nothing to prevent us from naming simple types. Producing the algorithm can thus become more schematic.

```cpp
#include <iostream>
using namespace std;
int main() {
    int a, b, c;          // declaration
    cin >> a >> b;        // reading input
    c = a + b;            // main process
    cout << c << endl;    // writing output
}
```

Figure 6: The C++ solution of the example task

4. Input and output in the specification and algorithm

The specification (Figure 3) does not have a *direct* impact on reading the input and writing the output. The input and output section describes the data and their structure required to solve the task. The pre-condition narrows the range of possible input data, and the post-condition specifies the connection between the input and output data. However, none of those parts deal with where and how the data comes from or goes. The specification deals with the conditions that must exist *before* and *after* solving a specific task (i.e. the main process).

As can be seen from the example algorithms (Figure 4 and Figure 5), the algorithm does not include, at least in the commonly used shorter form, reading and writing operations. At the same time, when constructing the algorithm, whether it is a NSD or a pseudocode, we have many implicit assumptions: 1) it is known from the specification which data are the inputs and outputs; 2) their types are also known from the specification; 3) the input data is correctly given at the beginning of the algorithm. Implicit assumptions become evident if we construct the algorithm of not only the essential task, but the whole program (Figure 7).[2] This is worth doing because it can show the general structure of the programs, where each program consists of three main parts: *reading input*, *processing* and *writing output*.

```
Const ...                    <-- see short algorithm
Type TInt=Integer            <-- see short algorithm
Var  a, b, c: TInt
Program
    In: a, b [a>0 and b>0]
    c:=a+b                   <-- see short algorithm
    Out: c
Program end
```

Figure 7: The algorithm of the whole program of the example task

In this version, our previous assumptions become explicit: it can be seen what was read and what was written, what the types of the variables are, and it is also clear that processing is started with correctly read input values, and the resulting data must be written out. However, usually it is not necessary to write such detailed algorithm, as 1) the name and type of variables can be "generated" from the specification, and 2) the structure of the program always task-independently builds up from the triple of input-processing-output, and finally 3) the input and output part also can be "generated" according to the specification. What remains is the essential part: the specification of the data structures (with type definitions, if necessary) and the sequence of elementary operations to solve the task.

With subroutines, the algorithm of the example task looks like this:

[2] The examples are given with pseudocode in the following.

```
Const ...                          <-- see short algorithm
Type TInt=Integer                  <-- see short algorithm
Var  a, b, c: TInt
Program
    Read(a,b)
    Process(a,b,c)
    Write(c)
Program end

Procedure Read(Var a,b:TInt)
    In: a [a>0]
    In: b [b>0]
Procedure end

Procedure Process(Const a,b:TInt, Vált c:TInt) <-- see short algorithm
    c:=a+b                                     <-- see short algorithm
Procedure end                                  <-- see short algorithm
Procedure Write(Const c:TInt)
    Out: c
Procedure end
```

Figure 8: The algorithm of the example task with subroutines

In this case, it is also obvious that, apart from the data structure descriptions and the processing algorithm, everything else can be derived mechanically from the specification, and therefore those parts are usually omitted. However, the advantage of this detailed description is that it gives a firm recommendation on the code structure.

The above algorithm can be developed further. If the procedures were replaced by functions, the main program would look like this:

```
Program
    (a,b) := Read()
    (c)   := Process(a,b)
    Write(c)
Program end
```

Figure 9: The algorithm of the example task with functions

The parenthesized data on the left side of the assignment indicates that in this case the reading function should return a complex data structure containing the read data, and the processing function should return all the output data.

Furthermore, our detailed algorithm can be generalized. If a program is considered as a function mapping the output data from the input data, this concept can be reflected in the data structures:

```
Const ...                          <-- task-specific
Type TInt = Integer                <-- task-specific
Type TInput  = Record(a,b:TInt)    <-- task-specific
Type TOutput = Record(c:TInt)      <-- task-specific
Var  in:  TInput
     out: TOutput
```

Figure 10: Input-output data structures of the example task

The algorithm of the main program will thus become task-independent:

```
Program                                    Program
    Read(in)                                   in:=Read()
    Process(in, out)          OR               out:=Process(in)
    Write(out)                                 Write(out)
Program end                                Program end
```

Figure 11: Task-independent structure of the main program

The functional solution can be simplified to a function composition:

```
Write(Process(Read()))    <===>    Write∘Process∘Read
```

Figure 12: The functional approach of the algorithm of the main program

For procedures, the three subroutines can look like this:

```
Procedure Read(Var in:TInput)
    In: in.a [a>0]
    In: in.b [b>0]
Procedure end

Procedure Process(Const in:TInput, Var out:TOutput)
    out.c:=in.a+in.b
Procedure end

Procedure Write(Const out:TOutput)
    Out: out.c
Procedure end
```

Figure 13: Subroutines for the generalized input-output data structures

To avoid the continuous `in/out` references in the `Process` procedure, we need to do the following:

```
Procedure Process(Const in:TInput, Var out:TOutput)
Var a,b,c:TInt
    a:=in.a             <-- "read"
    b:=in.b             <-- "read"
    c:=a+b              <-- "process"
    out.c:=c            <-- "write"
Procedure end
```

Figure 14: "Reading" and "writing" in the main process in the case of the generalized input-output data structures

In the end, the structure of the main program became task-independent, and the task-specificities were moved to each subroutine. It should be noted, that the `Process` subroutine repeats the structure of the previous main program, as it reflects the read-process-output triple. The main difference in this case is that the "read" and "write" operations have been decoupled from the outside world, and the work can be done with clean data structures. On algorithmic level there are not so many benefits coming from this approach, because there are not many differences between "`In: a`" and "`a:=in.a`", but the simplicity of the latter can be utilized during coding.

Finally note that, regardless of a particular implementation, the program is executed in the following steps: 1) reading the data so that it is available in a predefined structure; 2) the main process results in a predefined structured output data, which are 3) written out. An interesting feature of this process is that each step is determined by the structure of the data. Defining the *data structure* is a very important part of the problem solution, it requires a number of conscious, algorithmic and code decisions, so this should be considered as the fourth (more precisely the zero) task of a problem solution. While reading, processing, and writing data can be solved independently of each other, each of them depends on the chosen data structure (Figure 15). Thus, special attention should be paid to specification and thus to the description of the data structure.

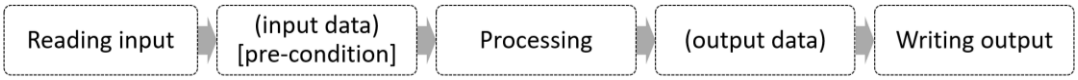

Figure 15: The role of the data structure on the read-process-write triple

5. Input and output in the code

At the beginning of programming education, there are mainly smaller tasks that make students practice the essential elements of problem solving and programming language. As it was already mentioned in the introduction, during coding, disproportionally amount of code needs to be written for reading the input compared to solving the main task. In this chapter, we will discuss what opportunities we have to simplify inputting. These techniques may be useful not only at the beginning of programming, but also later, when we want to deal with as many tasks as possible and not with their inputs.

5.1. Omitting input

If reading the input takes unacceptable effort, *the simplest if it is omitted*. In this case the defined input data structure is pre-filled with baked-in values. The solution leaves freedom to define the data structure that fits the task, only the input part is skipped immediately focusing on the task solution.

```
int main() {
    int a, b, c;        // declaration
    a = 3;              // reading input
    b = 5;
    c = a + b;          // processing
    cout << c << endl;  // writing putput
}
```

Figure 16: The C++ solution of the example task with baked-in input values

The disadvantage of this solution is that the program is not general: the program needs to be changed every time for specifying new input data. However, at this stage, the goal is not necessarily to write a general program, but to ensure that the student can solve the task with the provided data. They can learn the generalization later, e.g. with independent input reading tasks. The testing process is not much different in the omitted and general case:[3]

- omitted: *new data* → (*compile* → *run*) → *check the result*
- general: (*run*) → *new data* → *check the result*

A further disadvantage of this solution is that it cannot be tested automatically in this form.

5.2. Providing helper functions

Reading the input is often complicated, because it can be done with very low-level operations, and beginners need to use complex language elements to manage and control the standard input-output, and to transform incoming data. However, these low-level operations can be hidden behind *higher-level functions*. Of course, this helper library needs to be provided to the students somehow, for example a properly configured programming environment [3,4] can take care of it. The helper functions can be freely analysed, expanded, and thus contribute to the learning of input and output management. A great advantage of this approach, furthermore, that the

[3] The steps in parentheses can usually be done in one action in programming environments.

definition of data structures remains in the student's hands. Here are some examples for using such functions:

```
bool positive(int p) {
    return p > 0;
}

read(a);            // reading input
read(a, "a = ", "Positive number must be given!", positive);
read_line(s);
read_line(s, ';');
write(a);           // writing output
write_line(s);
```

Figure 17: Reading with helper functions

where, for example, verified reading can look like this:

```
template<typename T>
void read(T& p, string message, string errormsg, bool f(T)) {
    bool good;
    do {
        clog << message;
        cin >> p;
        good = f(p);
        if (!good) { clog << errormsg << endl; }
    } while(!good);
}
```

Figure 18: An example for a higher-level helper function for reading input

5.3. Pre-written input reading

Another option for making input reading easier is that the student receives a *prepared template* in which the input and output logic has already been implemented and should only complete the processing part correctly. As the reading part already fills up a given data structure, the freedom of data description is missing in these solutions. On the one hand, this can be a disadvantage, as it does not exercise this ability, but it can also be an advantage, as it can be used to show good patterns for data description of given problems, and also to practice adaptability, where you need to understand and work with existing decisions (soft skill).

5.3.1. Without subroutines

In the first case, the code is not divided into subroutines, the processing part is empty from the quadruple of the declaration-reading-processing-writing. Such tasks require proper preparation and can be used well in automatic assessment environments. Depending on the environment, additional option may be to hide certain parts of the prepared code or make it read-only. If the whole code remains editable, any language element (e.g. function) can be used. If the access is restricted, it depends on the programming language what elements can be used or not, that is why it is important to show the entire context of the code:

```
#include <iostream>
using namespace std;
int main() {
    int a, b, c;                // declaration
    cin >> a >> b;              // reading input
    // USER CODE BEGINS
                                // processing
    // USER CODE ENDS
    cout << c << endl;          // writing output
}
```

Figure 19: Pre-written code template without subroutines for the example task

5.3.2. With subroutines

With subroutines, the basic concept of the previous sub-section remains valid, only the body of the processing function is not filled (Figure 20).

```
// ...
int process(int a, int b);
int main() {
    // ...
    c = process(a, b);          // processing
    // ...
}
int process(int a, int b) {
    // USER CODE BEGINS

    // USER CODE ENDS
}
```

Figure 20: Pre-written code template with subroutines for the example task

Since subroutines are functionally well separable units, with proper task description it is possible to solve the problem by implementing *only one* function:

```
int process(int a, int b) {
    // USER CODE BEGINS

    // USER CODE ENDS
}
```

Figure 21: Pre-written code showing only the main processing function

This pattern has many advantages methodologically. On the one hand, it shows very well that the solution of the problem is sharply separated from input and output management. Functions, as interfaces, communicate with the outside world through their parameters and return values, not being interested in where the data comes from and where it goes. They practice modularity, separation of responsibilities, and they can be tested well as the calling environment determines the input data and evaluates the outgoing. It is up to the programming language how this kind of interface should be implemented, what language elements are allowed. The most flexible solution can be achieved by utilizing the modular options of a particular programming language (modules, classes, functions). In C++, for example, the user code can be implemented in a header file that is loaded and used by the runtime or testing environment:

```cpp
// task.h
#include <vector>            // arbitrary includes
typedef vector<int> Numbers; // arbitrary types and constants
int process(int a, int b) {
    // USER CODE
}

// program.cpp
#include "task.h"
int main() {
    int a, b, c;             // declaration
    // ...
    c = process(a, b);       // processing
    // ...
}
```

Figure 22: C++ user solution in a separate module for the example task

5.3.3. Input-output data structure

Investigating the input-output possibilities in algorithms (Section 4), we saw the option of input and output as a complex data structure. For prepared inputs, there may be a variant which works with such data structures. It can work with and without subroutines, of course, but the following examples utilize subroutines. For C++, such processing may look like this:

```cpp
typedef int Integer;
struct Input { Integer a, b; };
struct Output { Integer c; };
Output process(Input input) {
    int a = input.a;         // "reading input"
    int b = input.b;
    int c = a + b;           // "processing"
    Output output;           // "writing output"
    output.c = c;
    return output;
}
```

Figure 23: C++ implementation of an input-output data structure for the example task

With modern language elements, "reading" and "writing" can be simplified as follows:

```cpp
Output process(Input input) {
    auto [a, b] = input;     // "reading input"
    int c = a + b;           // "processing"
    return { c };            // "writing output"
}
```

Figure 24: Reading and writing using complex data structures in C++ with modern language elements

For example, in TypeScript, it looks like this:

```
type Input = {a: number, b: number};
type Output = {c: number};
export function process({a, b}: Input): Output {  // "reading input"
    const c: number = a + b;                      // "processing"
    return { c };                                 // "writing output"
}
```

Figure 25: Reading and writing using complex data structures in TypeScript with modern language elements

5.4. Structured input

In the case of console programs, the input is practically the sequence of the requested data. Its structure is therefore determined by the logic of the requesting program. If this data is saved into a file, then it will contain a set of numbers and texts, which can be understood in the light of the requesting logic, but without it, it remains just a meaningless unstructured file. In addition, reading the input is done in a reverse order: a sequence of data is given, and this must be read into the appropriate data structures. One of the advantages of reading unstructured input is to leave the freedom of data representation. The disadvantage is the complicated logic for reading, and the lack of transparency when it comes to edit the input file.

Using structured input can help a lot on those disadvantages. This practically means using a human-readable format for describing the input structure. Among today's popular text-based data-serialization formats it is worth mentioning JSON and YAML.[4] Table 1 shows a comparison between these two format in the case of our example and for a more complex data structure.[5]

YAML	JSON

`a: 3  # first number`
`b: 5  # second number`
	`{`
	`  "a": 3,`
	`  "b": 5`
	`}`

`teachers:`
`  -`
`  name: Zsakó László`
`  code: zslzsl`
`  -`
`  name: Szlávi Péter`
`  code: szpszp`
	`{ "teachers": [`
	`  { "name": "Zsakó László",`
	`  "code": "zslzsl" },`
	`  { "name": "Szlávi Péter",`
	`  "code": "szpszp" }`
	`  ]`
	`}`

Table 1: Comparing YAML and JSON format for the same examples

How to read structured data depends on the programming language. For these popular formats, there is usually a library that makes their management easy. For C++, there are external libraries for both formats[6,7], and for TypeScript, JSON is a legal citizen of the language, because it is a serialized form of JavaScript (and thus TypeScript) language structures, and YAML→JSON

[4] The XML format is used rather for machine processing than for human editing. Other formats such as HOCON (https://github.com/lightbend/config/blob/master/HOCON.md) is exciting, but their support and familiarity is low.

[5] It is worth noting that from version 1.2 the JSON description is part of the YAML language.

[6] JSON for C++, https://github.com/nlohmann/json (utoljára megtekintve: 2017.11.11.)

[7] YAML for C++, https://github.com/jbeder/yaml-cpp (utoljára megtekintve: 2017.11.11.)

converter can be found as well.[8] Generally speaking, it is good if the structured format can be easily mapped to the data structures of the language.

Appendix 2 shows how to read JSON data in C++. The applied JSON library automatically recognizes simple types and vector types, but for complex structures, the developer (teacher or student) must write the conversion code. For this, the knowledge of the library is required. The example in the appendix contains an intentionally more complex input reading.

There is no need for extra libraries for TypeScript:

```
const input: Input = JSON.parse(textarea.value);
```

Figure 26: Reading JSON input in TypeScript

The structured input, as its name suggests, already contains the decisions needed to represent the data. If the input data is given in this form, then the student does not practice the data representation. Structured inputs can also be used in automatic evaluation environments with proper preparation [3,4].

5.5. Input and output as a separate task

At the beginning of this article, we mentioned that during coding, the student actually meets three tasks: reading the data, calculating the result, and writing it out. So far, we have been focusing on how we can concentrate more on problem solving instead of input management. However, reading the input can also be a separate task: new language elements, concepts and techniques can be taught with it, and in more complex cases special programming theorems can be used during solution. For example, reading an array is a mapping programming theorem, while in precondition checking, any programming theorem (most often a decision) can occur. It can be given as task to student to read a given input file into a specific data structure, or to find out the data structure himself or herself. Thus, the input reading itself becomes processing, and the result of the reading must be printed to screen. Otherwise, it may have a positive benefit to force the students check the read data.

There has been little talk about the output part, because its complexity is usually much less than reading's. Less language items are required also. Thus, these kinds of tasks can come up relatively early, even at the very beginning. In these tasks, a data structure s given baked-in and this should be printed to the screen. Initially, these are simple types and can be written with a single command, later a selection is needed for writing logical values out, and an iteration for arrays. The applied programming theorems, if any, is almost always a mapping.

6. Conclusion

Nowadays, in programming education, we often find it natural that reading and writing is an integral part of problem solution, together with their complexities. Although this is a legitimate expectation looking at the end result, this article tried to show that this is not always necessarily the case. The three tasks can be separated from each other and gradually introduced. As we gradually progress towards the more difficult examples, it is also advisable to familiarize students with the more complex cases of input reading, when the students are prepared for its complexity. The three major subtasks of the solution (reading-processing-writing) do not have to go hand in hand in each task, their separation also allows us to deal with more tasks from the curriculum. For most tasks, omit the input or make it very simple, and only sometimes go into details, for

[8] YAML for JavaScript, https://github.com/nodeca/js-yaml (utoljára megtekintve: 2017.11.11.)

input-specific tasks or home assignments. This article showed a number of ways how to make input management easier. Each method also allows the input handling to be progressively introduced, for example, at first with baked-in data, then using prepared code templates, and finally with input-only tasks. For most tasks, however, it is advisable to completely ignore the input-output part. Most methods can work in traditional development environments, but learning can also be supported in special environments that provide automatic evaluation or editing capabilities.

References

1. Linda Mannila, Mia Peltomäki & Tapio Salakoski: *What about a simple language? Analyzing the difficulties in learning to program*, Computer Science Education, Vol. 16, No. 3, September 2006, pp. 211 – 227. DOI: 10.1080/08993400600912384
2. Eric Roberts: *An Overview of MiniJava*, Conference Paper in ACM SIGCSE Bulletin, March 2001. DOI: 10.1145/366413.364525
3. Horváth Győző, Menyhárt László: *Webböngészőben futó programozási környezet megvalósíthatósági vizsgálata*, INFODIDACT 2016, Zamárdi, 2016.
4. Győző Horváth: *A web-based programming environment for introductory programming courses in higher education*, Annales Mathematicae et Informaticae, 48 (2018) pp. 23–32

Appendix

1. An example for a more complex input reading in C++

```
#include <iostream>
using namespace std;
int main(){
  // declaration
  int a, b;
  int c;

  // reading input
  bool good;
  string sv;
  do {
    clog << "First number: ";
    cin >> a;
    good = cin.good() && (a>0);
    if (!good) {
      clog << "The number should be positive!" << endl;
      cin.clear();
      getline(cin, sv);
    }
  } while(!good);

  do {
    clog << "Second number: ";
    cin >> b;
    good = cin.good() && (b>=0);
    if (!good) {
      clog << "The number should be non-negative!" << endl;
      cin.clear();
      getline(cin, sv);
    }
  } while(!good);

  // processing
  c = a + b;

  // writing output
  cout << c << endl;

  return 0;
}
```

2. A complex example for reading JSON in C++

```
#include <iostream>
#include <vector>
#include "json.hpp"
using json = nlohmann::json;
using namespace std;

typedef vector< vector<int> > matrix;
struct Point { int x, y; };
```

```
struct Input {
    int a, b;
    vector<int> numbers;
    vector<Pont> arrayOfPoints;
    matrix m;
};
void from_json(const json& j, Pont& p) {
    p.x = j.at("x").get<int>();
    p.y = j.at("y").get<int>();
}
void from_json(const json& j, Input &input) {
    input.a            = j.at("a");
    input.b            = j.at("b");
    input.numbers      = j.at("numbers")      .get< vector<int> >();
    input.arrayOfPoints= j.at("arrayOfPoints").get< vector<Pont> >();
    input.m            = j.at("m")            .get<matrix>();
}
void process(Input input) {
    auto [a, b, numbers, arrayOfPoints, m] = input;
}
int main() {
    json j;                 // reading input
    cin >> j;
    Input input = j;
    process(input);         // processing
    // ...                  // writing output
}
```

Authors

HORVÁTH Győző

Eötvös Loránd University, Faculty of Informatics, Department of Media and Educational Informatics, Budapest, Hungary, e-mail: gyozo.horvath@inf.elte.hu

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 1, Number 1. 2019.

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.1.1.382

License