# Avoiding Bad Programming Practices in Education and Profession — Initial Considerations

**Robert Logozar**
University North, Croatia, Dpt. of Electrical Engineering,
104. brigade 3, HR-42000 Varazdin, Croatia, EU
robert.logozar@unin.hr

**Matija Mikac**
University North, Croatia, Dpt. of Electrical Engineering,
104. brigade 3, HR-42000 Varazdin, Croatia, EU
matija.mikac@unin.hr

*Abstract: In contemporary computer programming and various presentations thereof in all sorts of media, one can witness the emergence of several bad programming practices such as undermining the abstractness and generality of programs, poor commenting and input/output messaging, bad identifiers, brute-force computations that ignore closed-form results from elementary mathematics, indolence toward computational optimality, and many more. Several of those are also found in the programs produced by the GenAI (Generative Artificial Intelligence) tools, such as the freely available ChatGPT that we used here for comparison. We analyze those bad practices and discuss how to avoid and correct them by providing parallel exemplary programs, which are based on the best algorithms and implemented in C/C++ in a textbook, scholarly way. Drawbacks of bad program code range from hard readability and reusability to significantly and even drastically lower efficiency. This last, very degrading downside of bad programming is shown by measuring the execution times of inferiorly conceived and realized C/C++ functions for a few common programming examples, and by comparing them to the corresponding well-written functions with proper algorithms. The main reasons for bad programming habits and inferior source code quality are low prerequisite knowledge and skills, a weak foundation in mathematics and computer science, and a lack of intellectual and working discipline in both teachers and learners of computer programming. With more and more bad source code examples available on the Web, the future AI-generated programs could comprise considerable amounts of programming code of bad quality and low efficiency, or even code that gives incomplete or wrong results. This will happen unless the AI tools' input sources are supervised by experts.*

*Keywords: computer programming; bad programming practices; program form deficiencies; program formal and functional correctness; program code readability; influence on performance*

# 1 Introduction

As computer programming is becoming ubiquitous today, there is more and more content on this subject in all types and forms: from short instructions, more elaborate tutorials, to full teaching materials. Many of those are free and easily accessible. The ease of distributing such content to a wider audience is especially enhanced by the possibility of publishing those materials on the Web, often with little or no reviewing. The authors of such content possess various levels of programming knowledge and skills, as well as different levels of formal education in computer science. In the same manner, their attitude toward the necessity for critical considerations of their work can vary and is often insufficient. A popular belief and even an educational tendency today that "everybody can and must program" also contributes to the notion that anybody can also teach programming or at least freely disseminate her or his programming ideas and solutions to others.

To object to such an easy-going approach, here we outline bad programming practices often found in unrevised texts, videos, posts, and comments, mostly published on the Web, but after some time, also appearing in student and even professional works. Certain malpractices can also be found in otherwise solid textbooks that are over-pretentious in "demystifying" programming and making it "very easy for everybody". Consequently, such bad examples and explanations can and do appear in the computer programs programmed by novices, students, and even professionals who uncritically adopt them just because they seem easier to grasp than those from a proper teaching textbook. Because of all the above, the overall awareness of the importance of good programming practices and exemplary programming style seems to be on the decline these days.

Adopting good programming principles does require some discipline and effort, and many programmers, from beginners up, might think that it is not worth the effort. However, that is not true, and showing that argumentatively is the main motivation for concocting this paper.

An appropriate innuendo for this agenda would be to correctly interpret the motto promoted in the title of the famous D. Knuth's textbook: *The **Art** of Computer Programming* [1]. Programming certainly isn't a rigidly proscribed technical discipline, but the mentioned artistry should be taken rather conservatively! Every reader of this book will—immediately after opening it—realize that the title certainly does not refer to any sort of "free art." Namely, D. Knuth wrote all the algorithms and programs in that book in an assembly language that he invented specially for that purpose. In that way, he was enforcing the idea that every true programmer should also be familiar with the basic level of programming to learn the importance of thinking about every single machine instruction that she or he write. Thus, *the art* in that title is not in having freedom in everything, but in finding possibly different and still elegant solutions in the given, strictly proscribed logical and technical form of *good* computer programming. While doing that, the

programmer should also follow standard technical and engineering principles of clarity, accuracy, and consistency.[1]

To address the aforementioned problems and expose the solutions to them, in Section 2, we start with an overview of the "basic" and very common bad programming practices that include the loss of the abstractness and generality of the written program code. In Section 3, we argue on the problem of teaching practical programming to learners without sufficient theoretical background and discuss how bad examples — emanating from the bad programming practices — deteriorate the programming discipline. Section 4 illustrates the abuse of brute-force to calculate tasks that could be mathematically solved much more elegantly, and warns about the constant need to analyze the complexity and efficiency of the written code. Finally, in Section 0, we provide the concluding remarks on the subject.

The examples in the text are given in C++, as a common language of choice for introductory programming courses in the studies of computer science and electrical and electronic engineering. We hope that they are readable and comprehensible, as well as that the readers who program in other programming languages will be able to implement the presented general ideas in their work, too.

# 2 Basic and Common Bad Programming Practices

In this section, we start by addressing the problem of a possible violation of basic programming principles under the pretext of realizing "simpler" and "easier to understand" programming examples or even professional solutions. Hidden behind such an approach is most often the path of least resistance that originated from indolence and ignorance.

## 2.1 Undermining the Basic Principles of Programming — Abstractness and Generality

The fundamentals of programming are algorithms and data structures, and both of them are abstract notions. Once an algorithm and the corresponding data structures are correctly programmed in a desired programming language, the obtained program should correctly solve *all* imaginable concrete problems of that type, including special cases and exceptions. It should do it efficiently and in a transparent manner that is easy to understand for programming professionals. The novices must be introduced to this fact right from the beginning of their study of computer programming. There should be no excuses for not doing so. Children adopt their

[1] We discussed the importance of knowing the basics of programming in low-level languages in our educational paper [2].

first abstract mathematical concepts very early, when still in elementary school. They learn to describe geometric objects and their properties by using mathematical symbols and then calculate different quantities by general formulas that use those symbols. Teachers of programming must continue using fully abstract notions also in programming, showing their students that this discipline builds on the mathematical foundations they have learned earlier.

Those who ignore the need to keep everything exact, abstract, and formally correct might have their program code filled with several bad forms. One of them is the use of the so-called *magic numbers*. These are the explicit number constants written in the program expressions and statements, most often completely unexplained [3], [4]. Of course, the "starting" integers, i.e., those with the least absolute values, `0` and `1`, perhaps even `2` or `3`, whose (mathematical) meaning is obvious from the context, are not considered to be magic numbers. Although writing magic numbers in the program code is syntactically allowed by the standard computer languages, one of the basic rules of good programming is that they must be replaced with variables of the appropriate type and declared as *constants* to which their explicit numerical values are immediately assigned. After such a declaration — done in one place — those variables can be used as many times as needed, while the change of their value or even type is done only in the line of their declaration.

The even worse example of the use of magic numbers that can, unfortunately, also be found in many tutorials and teaching materials, is putting them in the condition or loop expressions, especially of the ever-present `for`-loop. The alleged excuse that this is done "for the sake of simplicity" should not be justified at any level of programming education. The number of iterations in a general program must not be expressed by a fixed numerical value of `5` or `10`, just because we might love those numbers! Similarly, in professional programming, no magic numbers should appear anywhere in the source code, as in the array declarations or the loop heads.

The loss of program generality and abstractness caused by magical numbers is illustrated in Listing I.A. A program code that performs the same task but is formally correct, clear, and easily maintainable is in Listing I.B. In those and in all the following pieces of the program code presented here in which it is needed, we assume the prior use of the statement: `using namespace std;`. It was omitted for the sake of brevity.

## 2.2 Incomplete and Unclear Source Code Comments and Descriptions of the Input and Output Data

The lack of comments and descriptions of the data to be input or output, or the imprecise formulations thereof, are notorious problems of badly written programs. Many of our students find excuses such as, "I intended to add the comments when the program is finished." Of course, it's not a big surprise if such students never finish their programs in the first place. If they cared to divide the given task into

precisely defined and described sections and then explain to themselves what they were doing and with what variables and functions by properly commenting on all of them (*clarity* as a *conditio sine qua non*), they might be more successful. Or, in the opposite case, students might have a program that works more or less well, but they cannot explain its parts because they do not understand them. That may also be — at least partly — caused by a nonchalant, genuine author of the program who didn't bother to write the necessary comments. Besides the general indolence of students and programmers, the omission of proper comments may as well be a consequence of yet another and quite prosaic reason: their bad typing skills. This problem shouldn't be ignored, especially in countries in which keyboard typing is not part of the regular school curriculum.

Regarding the program comments used in *teaching examples*, it is good to stress that at the beginning of the programming education, those can contain explanations of the language syntax and the meanings of statements. However, once the programming language or some of its parts are mastered and when students are given to solve concrete programming tasks, they must be explicitly warned that the comments in those programs should generally not contain reinterpretations of the written source code or explanations of other things that are obvious to language connoisseurs. They must — in the same way as it is done in "professional" programming — explain the specifics of a particular problem and, if needed, elaborate on the language tools used to solve that problem.

Concerning the position in the source code, there are two general types of comments:

i. Those written at the beginning of the separate parts of the code, announcing and perhaps shortly explaining their functionality and performance. Such comments are usually preceded by an empty line and are written starting from the same column as the statements below them. To emphasize that they refer to several lines below them, we usually end them with a semicolon.
ii. Those explaining a single statement or a line in a statement, which are written after the statement or in an appropriate position close to that.

The examples of comments — those written badly and those written correctly for the same programming task — are also presented in parts A and B of Listing I. In Listing I.A, we have probably exaggerated in writing only the obvious and thus completely unnecessary descriptions, but the intention was to illustrate what shouldn't be done. In Listing I.B, all that was remedied: the comments are informative and complete, and because of that, indispensable.

In bad programming, we also find unclear or imprecise descriptions of the input and output data from the user's standpoint. Starting with the former, we stress to our students that every input of the data in procedural programming must be preceded by a clear, precise, and concise description of what kind of data, in what form and range, is expected to be entered by the user in that input. Of course, that same information must be presented also in the programs with GUIs (*Graphical User*

Listing I.A An example of a source code with magic numbers, inappropriate variable names, accompanied by useless or imprecise code comments and output descriptions.

```
// AN ILLICIT PROGRAMMING STYLE!
// ...   ...
float a[1000]; // For storing decimal numbers.
int b;        // An integer number.
// The use of a do-while loop:
do
{
     cout << "Enter integer: "; cin >> b;
} while (b > 1000);
// Input with a for loop:
cout << "\nInput of decimal values:\n"
     <<  "=========================\n";
for(int c = 0; c < b; c++)
{
     cout << "Enter decimal num.: ";  cin >> a[c];
}
// ...   ...
int d[6]; // For storing integers.
// Input with a for loop:
cout << "\nData for the 6 grades:\n"
     << "\n===========================\n";
for(int e = 0; e < 6; e++)
{
     cout << "Number of grades " << e << ": ";  cin >> d[e];
}  // ...   ...
```

*Interfaces*), but presented in a different, suitable manner. Here, to make everything clear, the programmer must graphically organize the input and output boxes into logical groups, described by some additional text. If needed, this should be additionally supported by pop-up textboxes.

In Listing I.A, we again depict how it shouldn't be, and in Listing I.B, how it should be done in procedural or functional programming. When entering the array data, it is obligatory to specify the ordinal number of each entry. For a standard user, the counting should start from number 1, though in C/C++ and C-like languages, this first entry will be stored in the array element with index 0. Furthermore, bigger inputs (outputs) of data, usually made into (from) the arrays or other data structures, should be preceded by an appropriate common message, informing the user what kind of data and how much of it she or he will be prompted to enter.

## 2.3 Bad Identifiers

Laziness in forming proper identifiers is a common bad practice in computer programming. Many of the programmers who indulge in this coding sin — starting from beginners and students, and ending with experienced professionals — might

Listing I.B Source code with the same functionality as in Listing I.A, but written in a correct manner, which assures easy readability, updating, and reusability.

```
// ...   ...
// Declarations and initializations:
const int ciNEl = 1000; // Number of elements in fX.
float fX[ciNEl];  // Array fX for storing values "X", where X = ...
                  // (a concrete specification needed).
int iN; // The number of numbers to be entered in fX.
const int ciNGrds = 6;  // Number of grades.
int iGrdHist[ciNGrds]; // Array for the exam grade histogram, which
                   // stores the number of achieved grades:
                  // not attended = 0, exam grades = 1, 2, ... , 5.
// Input filter for iN:
do
{
     cout << "How many values X do you wish to enter? N <= " << ciNEl
          << ", N = ";  cin >> iN;
} while(iN > ciNEl);
// Inputting iN decimal X-values:
cout << "\nInput " << iN << " decimal X-values:\n"
     << "====================================\n";
for(int i = 0; i < iN; i++)
{
     cout << "fX(" << i + 1 << ") = ";  cin >> fX[i];
}
// ...   ...
// Inputting the grade histogram values:
cout << "\nNumber of appearances of the " << ciNGrds << " grades:\n"
     << "=========================================\n";
for(int i = 0; i < ciNGrds; i++)
{
     cout << "Grade " << i << ", num. of apprncs. = ";
     cin >> iGrdHist[i];
} // ...   ...
```

argue that identifiers will be obfuscated after compilation and that the only thing that matters here is the correct performance of the program! However, by saying that, they ignore the fact that dealing with unintuitive and even misleading names presents an unnecessary mental burden. In our educational practice, we have noted that after bad names are replaced with good ones, many students manage to correct the logical mistakes they have made in their programs. After the first code-writing sessions—during which everything is still fresh—poor names become an even worse problem. In the checking, improving, and upgrading phases, as well as in the possible reusing of the program code in other projects, well-chosen names of all programming entities are of utmost importance for the clarity and readability of the source code. A good practical rule is that if a programmer cannot figure out the purpose of a variable or a function from their name alone, by using only common mathematical and programming knowledge supported by some intuition, then those names are bad and should be changed to better ones.

We have illustrated a bad naming style in Listing I.A. There, the programmer reduced the lengths of the variable names to the minimum of only one (alpha) character. A plausible explanation for such minimalism could be the programmer's typing incompetence (cf. also the previous subsection, §2.2). Furthermore, there we see a "wise" solution on how to elude the effort for inventing the proper identifiers: the first variable has the name `a`, the second one `b`, the third one `c`, etc. We encounter such and other similar "ingenious" naming systems in our educational practice and spend quite some time warning our students against it. Every programmer will experience the bad side of such variable naming as soon as she or he has to use those variables in a meaningful way.[2] That is why several naming conventions have been introduced in computer programming [5].

In a nutshell, the variable names should follow the Latin maxim "*Nomen est omen"* (*the name is a sign*). Following that idea, we always suggest the use of names close to the symbols common in mathematics, physics, engineering, and other areas to which the corresponding programming task belongs. If those symbols are well-known, there is no need to write the full name of a particular quantity. To these common symbols, one should add additional parts that resemble the specifics of those quantities, which are normally written as subscripts, sometimes also as superscripts. For example, good names for three types of velocities, the initial, final, and average ones — with alternative writing of uppercase and lowercase letters to emphasize different name parts in the *PascalCase* [5] — would be:

`Vinit` (or `V0`), `Vfin`, `Vavg`.

The same names with only lowercase letters and the underscores for separating the different word parts would be:

`v_init` (or `v_0`), `v_fin`, `v_avg`.

We stick to the previous style because it is shorter.

The final touch of a good naming convention would be to include a short indication of the variable data type or the object class name at the beginning of an identifier. It often suffices to put there only the first letter of that type or, for the instances of some specific class objects, two or more. With this, a programmer knows the variable type immediately from its name, which makes the consideration of their types and possible type casting easier.

All the above rules together are encompassed within the naming convention known as *Hungarian notation* [6],[7]. It originates from and is now the official choice of

[2] There is a saying that one professor of a basic programming course at the Faculty of Electrical Engineering and Computing of the University of Zagreb (https://www.fer.unizg.hr/en), which all the authors of this paper attended, wrote a program that the lazy students to write proper names had to explain. In it, the first variable was named "_", the second one "__", the third one "___", etc. That was a clear warning to them about what source code can turn into if the identifiers are not treated as they should.

the software giant Microsoft®. It is simple and intuitive, and many programmers, including the first author of this paper, are using it systematically.[3]

For instance, if the variables in the previous example were all of the double-precision floating-point type, in C/C++ declared as `double`, and if we additionally apply the so-called *camelCase* style of writing names [5] — which we usually do — those identifiers would become:

`dVinit` (or `dV0`), `dVfin`, `dVavr`.

By writing the type-designating letter in lowercase, it is visually more distinctive from the rest of the name.

Hungarian notation applies to all other language entities. Since the class objects are essentially compound variables, their names are also written in camelCase, preceded by the class name abbreviation. As for the function names, some people would stick to the camelCase, especially if they indicate the function's return type at the beginning of its name. It may be useful, because function name overloading is not valid if the only difference between them is in the return type. However, not many programmers do that, and in that case, the PascalCase style is more common. Finally, for the class names, the PascalCase dominates.

It is good to remind the readers — especially the younger ones, who might not be familiar with the legacy computer languages — that the idea of connecting variable names with their types is not at all new. The old FORTRAN has an implicit assignment of data types to all the variables that are not declared explicitly, which is based on their names. [8] This *implicit typing* follows the usual mathematical convention that the symbols (starting with) $i, j, k, l, m, n$ are standardly reserved for natural numbers and integers, while the remaining letters are used for the quantities that belong to the other number sets. Even the youngest readers of this paper should recognize that from their elementary mathematics courses. If not, then they — together with those who write their identifiers as in Listing I.A — should be aware of their overall mathematical illiteracy.

The use of Hungarian notation for the variable identifiers is illustrated in Listing I.B. A reader can note that the names presented there are not at all long. This is partially because of the above-depicted naming style, which assures good readability without the insertion of underscores. Another reason is that the words that the names consist of are considerably abbreviated. A good practice is to start with longer names, containing even the full-length words, and then shorten them till the abbreviations are still easily recognizable in the programming context. We do this in the following example for the name of a single-precision floating-point variable, in C/C++ declared as `float`:

`fCircleCircumference` → `fCrclCrcmfrnc` → `fCrcmf`.

[3] The first author of this paper has been using this notation even before it was proclaimed as such, but without the specification of the data types.

The possible introduction of slightly longer names by using this convention should have no influence whatsoever on the *execution times* of programs written in any of the compiled languages (C/C++, Pascal, FORTRAN/Fortran, ALGOL, …), as well as for the execution of the bytecode of intermediary compiled languages (Java, C#, Python). It does not affect the execution time of the programs and could just slightly slow down their *compilation time*, which cannot be considered a disadvantage. For the normal, not excessively long names, the direct interpretation of the programs written in the interpreted languages, as is the console interpretation of Python programs and the standard BASIC code, can be slowed down a bit [9].[4]

Although widely known and without any real disadvantages, the Hungarian notation is not generally accepted in the literature and is even less present in programming practice — of course, except in Microsoft! Many professional programmers use only some of its rules or completely ignore the notation. Some authors in this area even criticize it, but without real arguments. Again, it might be that they are just trying to mask their indolence in concocting proper names or the lack of typing proficiency, in this case, the fast switching between lowercase and uppercase letters. All this justifies one of the aims of this paper: to strongly recommend the use of Hungarian notation to our readers, just as we suggest it to our students.[5]

## 2.4 GenAI Programming Practices — or the Lack of Them

In our educational paper [10], we put ChatGPT version 3.5, as one of the initial and still most popular *GenAI* (*Generative Artificial Intelligence*) tools, to a test of basic programming knowledge. From a series of C++ source codes that this version, and later on, also version 4.0 provided, we learned that:

a) ChatGPT kept the overall abstractness and generality of its programming, i.e., there were no bad programming practices described in §2.1.

b) It sometimes writes and sometimes does not write code comments, which depends on the sources it has taken and "learned" the solutions from. Such an inconsistency can be considered a bad programming practice (cf. §2.2).

c) ChatGPT regularly provides correct descriptions of the required input and the provided output data for program users. Those descriptions are in a style that is less mathematically inclined and precise than we would suggest, but which is acceptable for the wider range of users (confer the examples in [10]).

[4] We could not find concrete data about how (very) long identifiers slow down BASIC interpreters. Still, most casual comments suggest that this is insignificant if the names are within a reasonable length.

[5] Though Hungarian notation is proclaimed here, we know that things are easier said than done! It is fair to admit that only the first author of this paper uses this naming convention strictly and consistently in his programs. The second author uses it only partly. Similarly, only a portion of our students really adopt the Hungarian notation in their programming style.

d) ChatGPT normally constructs identifiers as would be expected from a common programmer. The names are easily understandable because they mostly consist of full words, as in the following examples:

*variables:* `validInput`, `mean`, `sumSquares`/`sumSquared`;
*arrays:* `numbers, text;`
*functions:* `inputNumbers`, `getDecimalNumbers`, `calculateMean`, `calculateStandardDeviation`, `removeExtraSpaces`, … .

Here we see the camelCase-style names as are often used in the Hungarian notation, but without the designation of the data types. Such complete words might at least be partial justification for the frequent omission of their proper comments [cf. point c)]. However, such long names are often very clumsy, and with only a little effort, one can shorten them while preserving their readability, for instance, as follows: `valIn`, `sumSqrs`, `getDecNums`, `calcMean`, `calcStDev, … .`

Despite the above-stated clumsiness, ChatGPT has surprised us from time to time with better naming in its programs. This can be seen in the following examples:

*variables:* `n_1, n_2, n_a, n_b, n1, n2, xl, xh, N_num, min, max, squareDiffSum, correctedStdDev;`
*arrays:* `str;`
*functions:* `gcd.`

These names are shorter and handier, but again without the designations of the data types at their beginnings. After urging ChatGPT several times to apply Hungarian notation correctly — which often looked as if it didn't know what that really was — it managed to provide a solution with satisfactory identifiers. However, some of the variable names were still a bit long (cf. §III.B.3, Fig. 2, and §III.C in [10]):

*variables:* `iN1, iN2, dD, dXc, dXl, dXh, dUserInput, bIsValidInput.`

In this solution, the comments *above* the specific source code sections were correct, but *behind* the individual statements, there were no comments at all. Thus, although the meaning of the above variables could be guessed by an educated reader, those short and mathematically nicely looking names remained without explicit explanations.

Generally, we can give ChatGPT a satisfactory grade for programming style and for avoiding the basic programming malpractices, depicted here in §2.1 and §2.3. However, one can criticize its lack of consistency and a constant urge to change both the programming style and solutions, sometimes from better to worse. More comments on this are given in [10].

# 3 Inadequate Theoretical Background and Bad Examples

After elaborating on the common bad practices that concern the basic principles of good programming (Sec. 2), here we warn against the nowadays quite frequent *easy-going* and *learning-by-example* only approach to programming, as well as the consequences it can cause for the knowledge of both students and professionals, especially the former and beginners.

## 3.1 Teaching and Learning Only "What's Needed to Make the Program Work"

The general problem of today's easily accessible partial information on different subjects is the underlying idea that one can get straight to the final solution to some task without even attempting to learn and understand the broader context needed to comprehend the problem and its solution systematically. Such a shortened, straight-to-the-solution approach, often required by modern learners, misleads the writers of programming tutorials to act in the same line of least resistance. Thus, they tend to ignore or just do not emphasize enough the need for prerequisite knowledge, skip the "unimportant previous chapters" that were considered essential just yesterday, and explain everything in the "easiest possible", but often imprecise way. In such an approach, there is no place for the ideas of D. Knuth about the necessity of a thorough learning method in programming, mentioned in our introduction (Sec. 1).

In other words, there should be a steady pace in the teaching and learning process of the novices, which balances between today's standard of the early hands-on approach and the necessity to pass all the introductory chapters. We advocate the classical order: first introducing the concept of an algorithm and different forms of its presentation, then learning the basic syntax of a computer language, and existing data types and operators. After that comes the implementation of the programming structures (sequence, selection, and iteration) with the tools available in the chosen language, as the basis to go further on with arrays, pointers, functions, etc. In short, the all-important introductory chapters of the old-school teaching method should not be skipped over just because they might be less interesting to novices.

Our population of college students is very diverse, with rather different prior programming knowledge and skills. Because of that, the introductory computer exercises — which accompany the introductory lecture chapters that are logically and programming-wise less demanding — are used to strengthen the needed initial skills in writing the source code and getting used to the chosen IDE. This is accompanied by learning the language syntax and solving simpler programming tasks that include a good understanding of data types and operators. The students must acquire those fundamentals before they get to the implementation of the programming structures of selection (branching) and iteration (looping). These

crucial topics for the program execution control and implementation of algorithms are then elaborated from the simplest and most general to the more complex and specialized forms.

For instance, in C/C++, one should first teach the `while` loop, because it is the basic and yet most general form of a loop in those and other higher-level languages. Although the simplest one, it enables the implementation of every imaginable iteration. Logically, after the `while` loop, one would explain the `do-while` loop. Only after that, follow the famous `for` loop. Such was the order of explaining the loops in the "C Bible," by B. Kernighan and D. Ritchie [11], where the execution of the `for` loop was additionally explained by implementing its functionality using the `while` loop. No other approach to explaining those essential iteration tools could be better for the novices, although we have witnessed the attempts of other approaches among our fellow lecturers. The situation is different in the books that are meant for seasoned programmers, where the authors may expose their ideas more freely, not paying so much attention to the basic formalities (cf., e.g., [12]).

After mastering those programming fundamentals, the learners should be acquainted with the basic data structure — the array. Only then, the other, "advanced" topics should follow, such as pointers and functions. Of them, the former prepares the ground for the latter by assuring understanding of the argument-passing mechanisms that provide their addresses. Interchanging this classical order of teaching programming with some ad-hoc introduced novelties may and will result in gaps in students' knowledge, particularly those with little or no prior understanding of programming and computer science.

## 3.2 The Curse of Bad Examples

If the program examples are designed with the sole criterion of being "simple and easy to understand," they may turn out to be computationally and mathematically misleading. For instance, to avoid the need for input data in explaining the use of the `for` loop, we might reach for the members of the well-known countable sets: the set of natural numbers and the set of integers. The simplest such task that comes to our minds might be the following:

***Probl. 1*** Find the sum of the first $n$ natural numbers.

Indeed, isn't this an excellent example of the `for` loop usage, which simplifies the loop body to a minimum, leaving the emphasis on the three expressions in the `for` loop head? Well, it would be a good example of the use of that loop if the brute-force summing of the consecutive numbers is not needed to get this result in the first place! In other words, if we know that we could get this sum by a formula for the arithmetic series, this is very good for us and our students (see APPENDIX A)! Then this example can clearly show how the `for` loop calculates this in $n$ steps of iterative summing, corresponding to the summation under the summing symbol in the next formula:

$$S_n = \sum_{i=1}^{n} i = n\,(n+1)/2\ . \qquad (1)$$

On the other hand, the usage of the rightmost side of the formula does the same task in just "one step."[6] Thus, the time complexity of the for-loop summing in *big O notation* is $O(n)$, and that of the one-step arithmetic expression calculation $O(1)$ (see e.g., [13]). Although the former uses only the simplest, addition operation, and the latter uses one addition, i.e., incrementation, one multiplication, and one division, the latter is computationally superior because it is independent of $n$.

The correct approach to solving *Probl. 1* is illustrated in Listing II. The abundant educational comments serve the purpose of warning future programmers never to calculate such a sum by adding consecutive numbers, but by applying the closed-form formula. Appendix A serves to further remind them that all other imaginable

Listing II. Finding the sum of the first $n$ natural numbers (*Probl. 1*) by: a) summing them in a for loop [complexity $O(n)$] and b) by evaluating a closed-form formula [$O(1)$]. Without the latter, this would be a bad programming example exposing the programmer's mathematical ignorance.

```
// ...   ...
// Declaration and initialization of variables:
unsigned int uN;            // A natural number, uN = n.
unsigned int uSumN = 0;     // Sum of first n natural nums.
unsigned int uSumN_AS;   // Sum of the first n natural numbers
                    // obtained as the arithmetic series.
// A message to the user:
     cout << "Finding the sum of first n natural numbers\n"
     << "=======================================\n\n"
     << "Input n >= 1, n = "; cin >> uN;
 // a) Brute-force approach: summing of the first n natural numbers
//     in a for-loop, which requires n summing operations:
   for(unsigned int uI = 1; uI <= uN; uI++)
   uSumN += uI;
  // b) Calculation of the above sum by the formula for the sum S_n
 // of n members of the arithmetic series, S_n = n(n + 1)/2, which
// requires 1 summing, 1 multiplying and 1 dividing operation,
// independent of the number n.
     uSumN_AS = uN*(uN + 1)/2;
     // Output of the results:
     cout << '\n'
     << "Sum of nums. from i = 1 to i = n = " << uN << '\n'
     << "=========================================\n"
     << " -- by adding:  S_n = " << uSumN    << '\n'
     << " -- by formula: S_n = " << uSumN_AS << '\n'
     << endl;
// ...   ...
```

[6] The mathematical legend says that young Gauss found that very formula while he was in elementary school [14], which means that programmers who don't know it nowadays, some 250 years later, really lag with their elementary math.

arithmetic series must also be calculated by closed-form formulas. There, we have provided such formulas for the sums of the first $n$ odd and even natural numbers, which also often appear in programming examples.

Good use cases for the for and other loops in problems of the above type are to provide *different* numbers of the chosen data type being input by the user. In the early examples, they need not be stored, and later on, they can be stored in an array. Then all sorts of checks and calculations on those numbers must be done on each of them, one by one, within some sort of a loop. When doing that, a programmer would have to recall the basic knowledge of when to use which loop, which is quite often forgotten or at least ignored, even among professionals. Before reminding us of that rule, let us observe the following elementary programming problem:

> ***Probl. 2*** Realize the input of an unknown number of numbers of some data type into an appropriate static array. The input is stopped after entering a negative number, which must not be inserted into the array, or after the array is filled. After finishing the input, the number n of entered elements must be determined and stored. For the concrete implementation, let the data type be the standard integer and let the array have 1000 elements.

We asked ChatGPT to solve this problem again.[7] According to the source code it provided, given in Listing III.A, it also does not know when to use and when *not to use* the for loop. Namely, in this case, the latter applies, because the rule is: of the two *general* loops in C/C++, while and for (where "general" means that they can have an arbitrary number of passages through the loop body, starting from zero), the while loop should be used whenever the number of passages is *not known* in advance, and the for loop should be used whenever the number of passages, i.e., the allowed values of the loop index are known in some way, either by being input by the user or obtained by some calculation. As for the non-general, do-while loop, one should use it on the same occasions as the while loop, but when the loop body is to be executed at least once, i.e., when the loop control (head) is better suited after (the first passage of) the loop body.

Being armed with those rules, we can inspect ChatGPT's solution more critically. Although it will give a correct result, one can immediately notice that the for loop condition is incomplete because it only checks the upper value of the loop/element index. The non-negativity of the number entered into the array is assured later, by a conditional break statement inside the loop body, which stops the iteration if the number is negative. This is formally a bad solution because the break and continue statements within a loop body *disrupt the given loop structure*. Furthermore, there are no ordinal numbers before the input entries, which is an obligatory assistance in a properly designed input. However, it is worth noting that this time, ChatGPT

[7] As in [10], we used the free, ChatGPT 3.5 version. However, when asked about that, it replied that it uses "v2", but "…my underlying model is part of OpenAI's **GPT-4** family.", "… more capable than GPT-3.5 in handling complex tasks, reasoning, and maintaining context across conversations."

Listing III.A A bad example of the usage of a for loop to solve *Probl. 2*, because the while loop is better suited here. Programmed by ChatGpt ver. 3.5, and a similar solution by ver. 4 (cf. footnote 7).

```
// ...   ...
const int ntot = 1000; // Maximum size of the array
int arr[ntot];         // Array to store integers
int n = 0;            // Number of valid elements entered
cout << "Enter up to " << ntot << " non-negative integers”
     << “ (input ends with a negative number): \n";
     << “==========================================\n";
for (int i = 0; i < ntot; i++) {
     int num;
     cin >> num;
     if (num < 0) {
          break;   // Stop input on a negative number
     }
     arr[n] = num; // Store the valid number in the array
     n++;     // Increment the count of valid elements
}
// Print the number of valid elements and the array
cout << "\nNumber of elements entered: " << n << "\n";
cout << "Elements: ";
for (int i = 0; i < n; i++) {
     cout << arr[i] << " ";
}
cout << endl;
// ...   ...
```

provided comments for several statements. Also, although not required in the problem, after displaying the *number* of entered numbers, ChatGPT programmed the output of all the entered numbers. Still, it aligned them one after another in a row, again without their ordinal numbers.

After being prompted that a `for` loop is not the best choice for this algorithm, ChatGPT offered its 2nd solution. In it, the `for` loop was changed to a `while` loop that uses a predefined index, but with the same incomplete condition that required the conditional `break` in the loop body. When warned about that, ChatGPT made a solution similar to the one in Listing III.A, which first checked `if` the input number was non-negative in an `if-else` statement at the end of the block, and if not (`else`), it raised the loop index to the limit value (`i = ntot`) to force the loop exit in the very next step — the condition test. However, that was just a masked and overcomplicated version of the `break` functionality. All in all, the additional two ChatGPT solutions that were explicitly prompted by us had only cosmetic changes, which did not contribute to either the formal or functional improvement of the produced source code.

Listing III.B Exemplary solution to *Probl. 2*. It uses a while loop with a condition that checks both the number non-negativity and the index range, avoiding the need for the break statement.

```
 // ...   ...
// Declarations and initializations:
const int ciNEl = 1000; // Num. of elements of iArrX.
int iArrX[ciNEl];      // Array with ciNEl int elements, where
                       //  X = ... (a concrete specification).
int iHlp;    // Helping var. for accepting input values.
int iN = 0; // Loop and array index. At the end of input, it will
            // contain the number of entered numbers.
// Description of the input and how to stop it:
cout << "Input up to " << ciNEl << " integers >= 0 "
       << "(to end, enter a negative number):\n";
// Initial input to be checked by the while loop cond.:
cout << "iX(" << iN + 1 << ") = "; cin >> iHlp;
// while loop for entering the input values:
while(iHlp >= 0 && iN < ciNEl)
{
   iArrX[iN++] = iHlp;  // iN++ = postincrmenting.
   cout << "iX(" << iN + 1 << ") = "; cin >> iHlp;
}
// Output of the results:
cout << "\nNumber of entered el., n = " << iN << '\n';
cout << "\nList of the " << iN << " entered integers:\n"
     << "=====================================\n";
for (int i = 0; i < iN; i++)
     cout << "iX(" << i + 1 << ") = " << iArrX[i] << '\n';
cout << endl;
// ...   ...
```

Then, on a quite surprising initiative of its own, ChatGPT offered "A Cleaner Approach." However, that was a flaw because the code was as in the 2nd solution, but now without the index incrementation, and therefore incorrect. We had to prompt it again about the mistake. Finally, the 5th ChatGPT's solution had the two needed relational expressions in the while loop condition, but with the parenthesized input operation clumsily crammed in between the multiple AND operations:

```
while(n < ntot && (cin >> num) && num >= 0) { … }
```

Quite astonishingly, this while loop is syntactically correct and works well, but again, without a proper message to the user about which element is being input.

The above strange solution provoked us to improve it: to ensure the right message before the input, the logical expression after that, and all of those separated by commas, as multiple expressions should be. This gave the following while head:

```
while(cout << "num(" << n + 1 << ") = ", cin >> num,
      num >= 0 && n < ntot)) { … }
```

It also turned out to be correct C++ code. Except for being cluttered and hard to read, it is functionally the same as our textbook example in Listing III.B. A careful

reader has probably noted that both here and in Listing III.B, we have changed the order of the relational expressions. By putting the relation that is more likely to be false first, one takes advantage of the short-circuit evaluation of the AND operation in C/C++.

Any of the above solutions with the number input buried in the while loop condition cannot be recommended to anyone, not even to experienced programmers. In contrast to them, Listing III.B contains an exemplary solution. The need to write the combined "input line", with the `cout` and `cin` statements, twice is a small sacrifice for the achieved clarity. Also, after being incited by the self-initiated and not required(!) printout of the entered elements in ChatGPT's solution, we provided the same thing in the light gray font at the end of Listing III.B. It gives the numbered list of the entered integers in a way that it should always be when the array elements are output.

We have briefly checked what solutions of the same, *Probl. 2*, would give the newly available GenAI models. ChatGPT-5 and Gemini were giving solutions based on a `for`-loop with an early break sentinel. Claude Sonet 4 and Grok produced a while-loop formulation but still relied on the `break` statement for the early termination of the loop. Obviously, the clumsy solutions of this problem, containing `break` statements are widespread in the AI knowledge bases, and because of that, all these tools deviate from the exemplary and optimal solution. On the other hand, all those cumbersome program solutions work quite efficiently and thus mislead an uneducated user into adopting a formally inferior and sloppy way of programming.

In conclusion to this deliberation on the program codes in Listing III, we will comment shortly on what some readers might consider as our unjustified ban on the `break` statement.[8] Their first argument could be why the language creators provided such a statement if it should not be used. The simple answer is that the `break` generally serves to get the program control immediately behind the block it is placed in. It does it in a similar way the `goto` statement would do, but without requiring a label in front of the first statement after the block. Furthermore, the `break` statements are indispensable for the regular functioning of the `switch` branching. However — as we have just illustrated in the relatively simple case above — they are not needed in the loop blocks. This can also be shown for other positions of the

[8] The simplest case is the one with the conditional `break` statement at the beginning of the loop block, when the negated condition can be moved to the loop condition. When it is at the end, the rearrangement is also straightforward, but with an outward if-statement. When the `break` statement is in the middle of the block and some statements can be rearranged so that the break moves at the top of the block, this leads to our case in Listing III.A. Some specially constructed cases, with several `break` statements inserted among other statements, could be hard to rearrange, but not harder than the code with many goto statements that could be rearranged to a well-structured program code, e.g., using a `switch` statement.

`break` statement within a loop block, but the analysis of those cases exceeds the scope of this paper.[9]

The next bad example comes from the "repository of solutions" collected by our students who would rather rely on other people's programs than on writing their own. This one solves an elementary task described in *Probl. 3*.

> ***Probl. 3*** Write a C/C++ function that finds the minimal and maximal value of $n$ numbers of a given data type entered in an appropriate array, with the total of $n_{el}$ elements, where $n \leq n_{el}$. Let the data type in the first implementation be `double`.

In Listing IV.A, we have sketched a solution to this problem according to a code snippet that we find from time to time in our students' test solutions, because it spreads around widely. It gives correct results, but in a very unwise manner. Anyone who submits such source code indicates clearly that she or he either haven't even read it or don't understand it. Namely, the first `if` statement stubbornly examines the loop index equality with zero for the whole range of index values, although it should be clear that `i = 0` occurs only once — in the first iteration. Because of that, the relatively expensive comparison operation is repeated in every single iteration of the for loop. After that, there is an inept doubling of the `if` statements, which separately check if the same array element is less than the minimal and greater than the maximal value, which cannot happen simultaneously. Instead, one should use the `else-if` construct, as in Listing IV.B.

We have already used one ChatGPT's solution to *Probl. 3* in our previous paper [10], within task 4 (LIST I). There, we didn't show this function, just commented on it (in §III.B.4), noting that it is data-type and platform dependent. Here, we show it in the upper part of Listing IV.B. In its lower part, there is our simpler and universal solution, which also has one iteration less. For the sake of completeness, we mention that ChatGPT used the same initialization of the `min` and `max` values as ours in one of its other solutions, where the finding of the extremal values was written among several other tasks of a larger function. We have described this variability and wavering in ChatGPT's programming, as well as its inability to stick to better solutions and build strictly upon them, in [10].

Another frequent bad example concerns the usage of C/C++ functions. Among other sources, we found it also in a voluminous textbook on programming in C++. The book's title suggests the authors' intention to make the subject very easy, and this idea was pursued further in their "friendly" and quite free writing style. As a starting example of a C-style function within procedural programming, there was the following function declaration and definition:

```
float square(float x) {return x*x;} // DON'T!
```

[9] In several Web forums, there are fiery discussions about the use of `break` and `continue` statements. Some programming "practitioners" eagerly defend this (ab)use wherever it suits them, including the bodies of the structured loops (see e.g. [15]). We invite them to send us examples of such code to be rearranged without a single `break` or `continue`.

Listing IV.A A bad solution to *Probl. 3*, as programmed by an unknown author (presumably a student from a nearby college). We have put the solution in the function and applied ChatGPT's programming style from above.

```
// Function for finding the minimal and maximal el.
void findMinMax1(double nums[], int n, double& min, double& max) {
   for (int i = 0; i < n; i++) {
        if (i == 0) {
            min = nums[i];
            max = nums[i];
        }
        if (nums[i] < min) {
            min = nums[i];
        }
        if (nums[i] > max) {
            max = nums[i];
        }
   }
}
```

Listing IV.B Correct solutions of *Probl. 3*. Above: One ChatGPT's version, which is data-type and platform dependent. Below: our exemplary solution, which is universal and has one iteration less than the previous one.

```
// A ChatGPT's ver. (except for the exact writ. style):
void findMinMax2(const double numbers[], int num, double& min,
                 double& max)  {
   min = std::numeric_limits<double>::max();
   max = std::numeric_limits<double>::lowest();
   for (int i = 0; i < num; ++i) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
        if (numbers[i] > max) {
            max = numbers[i];
        }
   }
}
```

```
// Exemplary version:
void findMinMaxInArr(const double dArrX[], int iN, double& dMin,
                     double& dMax)
{
   dMin = dArrX[0];  // Initial values of dMin
   dMax = dArrX[0]; // and dMax.
   for (int i = 1; i < iN; i++)
        if(dArrX[i] < dMin)
           dMin = dArrX[i];
        else if(dArrX[i] > dMax)
           dMax = dArrX[i];
}
```

While some benevolent readers could consider this an ingeniously simple example, we would rather call it an *anti-example*, because it shows what a true function should not be used for. Namely, although functions are allowed to perform even very simple tasks if they are distinctive and general, the function calling mechanism should not be wasted for what can be done "in place" by a simple calculation, as is the multiplication `x*x` in this case. Even the C++ `inline` attribute cannot help save the above declaration because such functions should not be used in theory, education, or practice, except if a reduced case of a broader solution, e.g. with complex numbers We do not remember if the authors emphasized the absurdity of their oversimplified example — perhaps they even did — but one would still ask why use it in the first place. On the other hand, a general `power` function for finding the $n$-th power of a number, let's say for a non-negative $n$ in its first version, would still be a relatively simple but good instance that depicts a calculation for which a function ought to be organized. It could be declared by the following prototype:

```
double dPow(float dX, unsigned short int usiN);
```

# 4 Brute Force Calculations That Ignore Elementary Mathematical and Programming Facts

## 4.1 Investigating the Well-Known Discrete Number Sets

In §3.2, we have already presented an example where a closed-form formula must replace the iterative summing of natural numbers (*Probl. 1*, Listing II). The well-known countable sets of natural and integer numbers often present a sort of "investigational challenge" for programmers who are — to put it mildly — less inclined to mathematics. As an example, let us present the following problem that can be frequently found as an illustration of the usage of the modulo operation:

> ***Probl. 4*** Write a C++ function that outputs all integer numbers $i$, $n_1 \leq i \leq n_2$, which are divisible by $m \geq 1$ ($i, m, n_1, n_2 \in \mathbb{Z}$).

Listing V.A gives the "usual, straightforward" solution, which busily checks the divisibility of every single integer in the given range. It comes as no surprise that ChatGPT gave a similar one, which performs all those unnecessary calculations. Namely, elementary mathematics says that neighboring numbers divisible by $m$ are $m$-apart from each other. This rudimentary fact governs the second, computationally optimal algorithm, presented in Listing VI.B. A formal proof of that, together with the definition of the modulo operation, is in Appendix B. Thus, the only remaining task is to find the first number $i_1 \geq n_1$, divisible by $m$, i.e., such that $i_1 \bmod m = 0$. This is best done by the three-case formula presented in Listing VI.B. We leave the proof of this formula for some future paper or for programmers who have recognized the importance and power of a mathematical approach. A much less elegant alternative would be to use the `while` loop that stops after finding

Listing V.A A simple but redundant algorithm for finding integers divisible by $m$ (*Probl. 4*)

```
void printNumsDivsblByK(int n1, int n2, int m) {
     // The divisor (m) must be a natural number, m >= 1.
   cout << "Numbers between " << n1 << " and " << n2 << " divisible by "
         << m << ":" << endl;
   for (int i = n1; i <= n2; i++) {
        if(i%m == 0) {
            cout << i << " ";
        }
   }
   cout << endl;
}
```

Listing VI.B Mathematically and computationally optimal algorithm for *Probl. 4*.

```
void outputNumsDivsblByK(int iN1, int iN2, int iM)
{
   int iN11 = iN1%iM; // iM >= 1! iN11 init. value.

   if(iN11 == 0)       // iN1 >= iN and iN1 mod iM = 0:
      iN11 = iN1;
   else if (iN1 < 0) // Case when iN1 < 0.
      iN11 = iN1 - iN11;
   else              // Case when iN >= 0.
      iN11 = iN1 - iN11 + iM;

   cout << "Natural numbers from " << iN1 << " to " << iN2
         << " divisible by " << iM << ":\n";

   for (int i = iN11; i <= iN2; i += iM)
        cout << i << " ";
   cout << endl;
}
```

the first such integer by applying the modulo operation. After $i_1$ is known, all other required numbers present an arithmetic progression of integers: $i_n = i_1 + (n-1)m$ (cf. Appendix A).

To illustrate how a badly conceived algorithm can influence the program performance, we have measured the execution times of the functions implemented in Listing V.A and Listing VI.B, which we shortly denote as $f_A$ and $f_B$. The typical results are given in Table 1. The measurements of the algorithm execution times are elaborated in our earlier papers [16] [17], and also in our recent paper [18]. The first two rows in Table 1 show that the inferior solution ($f_A$) is slower, but for the chosen input parameters only insignificantly: 2.9% and 0.7%. This is because the the advantage of the proper solution ($f_B$) in avoiding the unnecessary repetition of

Table 1
Execution times for functions $f_A$ (Listing V.A) and $f_B$ (Listing VI.B), and their reduced versions $f'_{A(B)}$ and $f''_{A(B)}$

| *Func. ver.* | $n_1(-)$, $n_2(+)$ | $m$ | $\Delta t$/ms | | $\frac{\Delta t(f_A)}{\Delta t(f_B)}$ |
|---|---|---|---|---|---|
| | | | $f_A, f'_A, f''_A$ | $f_B, f'_B, f''_B$ | |
| $f$ | $\mp 10^6$ | 17 | 53 317.3 | 51 797.7 | 1.029 |
| | $\mp 10^6$ | 1 023 | 1 103.2 | 1 095.7 | 1.007 |
| | $\mp 2 \times 10^9$ | 10 037 531 | 19 550.7 | 383.2 | 51.020 |
| $f'$ | $\mp 10^8$ | 113 | 1 067.8 | 7.3 | 146.274 |
| $f''$ | $\mp 10^6$ | 1023 | 12.7 | 2.6 | 4.885 |
| | $\mp 10^8$ | 113 | 858.6 | 6.0 | 143.100 |
| | $\mp 10^8$ | 1023 | 813.9 | 2.6 | 313.038 |
| | $\mp 10^9$ | 1023 | 8 170.9 | 7.9 | 1 034.291 |

the relatively complex modulo operation is completely dominated by the extremely time-consuming output via the `cout` object of the `ostream` class.[10]

The third row in Table 1, with $m \approx 10^7$, clearly shows this when the number of outputs is diminished. Then $f_A$ is more than $50$ times slower than $f_B$. To better investigate this, we have changed the output statement, `cout << i << " ";`, with the statements in which a predeclared (and predefined) variable is assigned a new value:

```
iTmp = i; // In the f'_A(B) function versions.
iCnt++;  // In the f''_A(B)  function versions.
```

This is similar to a solution in which the numbers found by the function would be stored in an array.

The results for the modified functions show that $f'_B$ and $f''_B$ are significantly faster than $f'_A$ and $f''_A$: at first for the factor of around $5$, and after that for the factor of the order of magnitude $10^2$ and even $10^3$.

Further analysis of these two algorithms, which would also include the inspection of their assembly code as was done in [18], exceeds the scope of this paper.

There are many variations of *Probl. 4*, and some of them may seem a bit enigmatic at first glance. However, they are all reducible to the solution presented in Listing VI.B. We gave one of them in task 2 in [10], which ChatGPT failed to solve in general. Another such example, with a mathematically weakly formulated text and bad naming, appeared in a high school test that we obtained by chance. Its edited version is as follows:

[10] A rough analysis of the assembly language code shows that the output statement `cout << i << " "` requires more than 10KiB of memory (roughly more than 3500 instructions). The short substituting statement for function $f'_{A,B}$ ($f''_{A,B}$) require only 3 (4) instructions, placed in 8B (10B). This suggests that the output via `cout` is more than 1000 times slower.

> ***Probl. 5*** Write a program that first inputs only three-digit natural numbers, $n_1$, $n_2$, and $k$, where $n_1 \leq k \leq n_2$. Then, by using a for loop, it lists all natural numbers $i$, $n_1 \leq i \leq n_2$, divisible by the two-digit number obtained from $k$ by removing its digit that represents hundreds, i.e., the leading one. If the two-digit number is zero, the program displays a warning about an illegal division by zero.

Here, we shall only briefly outline the program that solves this problem properly. The main function does the usual managing tasks, starting with the message about what the program does, and organizes the input of the required natural numbers. To ensure that the input values are as needed, there must be an input filter that prohibits integers less than 1, and an additional function that checks the number of digits in the input numbers. This is, of course, assumed in the ordinary sense, without leading zeros. Because of that, there are two preparatory functions in our version of the program:

```
int iPow(int iB, int iK);  // iB^iK needed in the next function.
bool bIsLDgtNum(int iB, int iL, int iN); // True if iN has iL digits
                                         // in the base iB posit. sys.
```

The second function can check the numbers from an arbitrary positional system with base `iB >= 2`. After the input is completed, the program calls the function implemented in Listing VI.B by inserting the input $n_1$, $n_2$ values for parameters `iN1`, `iN2`, and $k \bmod 10^2$ for `iM`.

In [10], under task number 2, we provided a similar example that "explores the properties of integers." It requires finding the integers within the given limits—including also the negative ones—that are odd and divisible by five. The solution provided by ChatGPT back then was a brute-force checking of every number in the given interval, in the same way it has been done now, a year and a half later (§4.1). Moreover, despite our several hints, it just could not make that solution work correctly when the lower limit was negative. ChatGPT has not even come close to the optimal solution given in Listing VI.B. In our optimal solution (Listing II in [10]), the formula for the initial number that is in the given range and that satisfies the above condition is a bit more complex than the one used in Listing VI.B. Of course, this is because *Probl. 4* refers to the basic such task. However, the formula still requires a straightforward mathematical derivation. A more detailed overview of this problem is given in [10].

At the end of this subsection, we hope that readers have realized that there is no need to computationally "investigate" the well-known discrete, ordered, and regular sets—such as the set of integers and the set of natural numbers—one number at a time. The problems of this kind should be reformulated to explore the properties of non-regular sets of numbers that are input by the user or already stored in an array. Then the numbers must be checked one at a time, in a loop with the usual minimal incrementation or decrementation of the loop index.

## 4.2 Ignoring Computational Optimality

Programmers who are not acquainted with the basics of computer science and do not know how the programs and the parts thereof are executed will often fall into the trap of coding something in a way that might look appealing and simple, without realizing that they produce inefficient and even detrimental programming results. Namely, if a computer language allows some programming construction, it does not automatically mean that this construction is suitable for a concrete program, nor that it is computationally justified. This is why future programmers should not skip a single chapter of their introductory programming courses, starting with the basic syntax, data types, operators, and so on.

To illustrate this, let us observe the following elementary example of a function performing a simple task on a C-string.

> ***Probl. 6*** Write a C/C++ function that replaces lowercase letters in a C-string with the corresponding uppercase letters and returns the number of performed changes.

One solution to this problem "that works!"—meaning only that it gives a correct result—is presented in the upper part of Listing VII.A. By the way, we spotted this solution in "somebody's" teaching materials! It shows a blatant misuse of the overall freedom of programming constructions in the C/C++ languages. First of all, the author of this programming bravura obviously did not know the rule that for an unknown number of iterations, the `while` loop is better (cf §3.2, *Probl. 2*), and turns to the use of the omnipresent `for` loop. However, this is almost nothing compared to the worst part of this code—which was possible because just about everything can stand in the C/C++ loop-head expressions (cf. comments on the ChatGPT's solution to *Probl. 2*). That is, the upper limit of the index was regulated by the call of the often-abused function, `strlen(char*)`. Certainly, such a shortening saved the programmer from declaring a new variable and writing one more statement in front of the loop. However, this indolence produced a slow algorithm with the time complexity of $O(n^2)$, for a task that can be simply solved by an algorithm of only $O(n)$ complexity. Thus, in each of the $n$ passages of the `for` loop, where $n$ is the unknown number of ASCII characters in the given C-string, function `strlen` is called to find that $n$. To do that, it needs $n$ iterations. Thus, this function performs in total $n \times n$ iterations.

As already suggested, the simplest correction to the above bad solution is to call the `strlen` function once and store its result in an integer variable, as is done in the lower part of Listing VII.A. However, this is still not the best possible solution, because the character array that contains the C-string need not be passed twice in the search for the terminating, null character (`'\0'`). The function in Listing VII.B shows how it ought to be done. Its `while` loop head resembles the one in the `strlen` function.

Listing VII.A Inferior solutions to *Probl. 6*. Top: very bad, with time complexity $O(n^2)$. Bottom: time complexity corrected to $O(n)$.

```
int toUpper1(char cTxt[]) {
   int nchg = 0; // Number of changes
   for (int i = 0; i < strlen(cTxt); i++)
        if(cTxt[i] >= 'a' && cTxt[i] <= 'z') {
           cTxt[i] -= 32;
        nchg++;
   }
   return nchg;
}
```

```
int toUpper2(char cTxt[]) {
   int nchg = 0; // Number of changes
   int iStrL = (int) strlen(cTxt); // Single func. call!
   for (int i = 0; i < iStrL; i++)
        // As above …  …
   return nchg;
}
```

Listing VII.B Exemplary solution to *Probl. 6*, with a single passage through the C-string

```
int lwrToUpprcsLttrs(char cTxt[])
{
   int iSC = 0; // iSC = the number of changed letters.
   int i = 0;  // The array and loop index.
   while (cTxt[i] != '\0')
   {
      if(cTxt[i] >= 'a' && cTxt[i] <= 'z')
      {
           cTxt[i] -= 0x20; // 'a' – 'A' = 20h.
           iSC++;
      }
      i++;
   }
   return iSC;
}
```

In fact, a modified version of this function can return the same value as `strlen` if one removes the statements with the declaration and incrementation of the `iSC` variable and changes the `return` statement to

```
return i; // Returns the C-string length.
```

Table 2 shows the average execution times of the three functions presented in Listing VII. As predicted, the relative performance of the first solution ($f_1$) compared to the performances of the other two functions ($f_2$ and $f_3$) is very inferior. Its concrete execution times are rising very close to the expected, quadratic progression. Thus, for an increase of $n$ by a factor of 10 (100), the execution time

Table 2
Execution times for functions $f_1$ and $f_2$ (Listing VII.A upper and lower part), and $f_3$ (Listing VII.B)

| $n$ | $\Delta t/\mathrm{ms}$ | | | $\frac{\Delta t(f_1)}{\Delta t(f_3)}$ | $\frac{\Delta t(f_2)}{\Delta t(f_3)}$ |
|---|---|---|---|---|---|
| | $f_1$ | $f_2$ | $f_3$ | | |
| $1 \times 10^4$ | 310.4 | 0.065 90 | 0.040 30 | 7 703. | 1.635 |
| $1 \times 10^5$ | 31 464.7 | 0.657 62 | 0.381 26 | 82 528. | 1.725 |
| $4 \times 10^5$ | 519 543.0 | 3.040 82 | 1.559 38 | 333 173. | 1.950 |
| $1 \times 10^6$ | 3 295 500. | 6.662 94 | 3.769 18 | 874 328. | 1.768 |

increased roughly by a factor of $10^2$ ($10^4$). The functions $f_2$ and $f_3$ are very close to linear dependence in $n$, and $f_3$ is, again as expected, superior and slightly less than two times faster than $f_2$.

An unnecessary repetition of statements within a (for) loop was also illustrated earlier, in Listing IV.A. Although the time complexity of that code stays within $O(n)$, the slowdown is for a factor not far from two. Generally, every programmer should carefully inspect the code she or he has written, and ensure that there are no unnecessary repeating calculations in the loops. For instance, if a loop uses a variable with an unchanged value (for instance, iC), and also needs some derived value from it (iC ± 1, iC/2, ...), this result should be stored in an additional variable before the loop. Furthermore, for very short functions, a good practice is to declare them as inline, because this forces the compiler to install the function operation in the place of the function call instead of using the relatively expensive function-calling mechanism. This is effectively similar to the use of macro statements, but syntactically nicer. A good general guiding principle for every programmer is to write and analyze the written program code as if it will be repeated *millions and billions of times*.

**Conclusions**

After exposing several bad programming practices and presenting how to combat and correct them, we start this section with a thought that will never lose its importance: "There is no substitute for thinking." In fact, in this new era of AI, we should rephrase this saying as "There is no substitute for *human* thinking." Also, there is no substitute for the programmers' solid prerequisite knowledge of mathematics, computer science, the computer language they are programming in, as well as the overall intellectual zeal and working discipline to write well-coded computer programs. This equally refers to presenters and learners of *programming science and art*.

Learning only by examples—especially the bad or dubious ones—is simply not enough. In oversimplified pieces of program code, one can lose the abstractness and generality, which are pillars of good programming. Furthermore, the lack of working discipline can lead to unclear and incomplete comments in the source code, poor descriptions of the input and output data, and badly composed identifiers. All

of this together can obscure the meaning of the written program code and result in programs that are hard to read, comprehend, and maintain.

Among the notoriously bad examples are those that "explore" the sets of integers and natural numbers — as is checking their divisibility by some number, or something similar — ignoring the facts that one should know about those well-known sets from elementary mathematics. Finding a good example for a presentation, or a problem for a home assignment, or a programming test, which is not too easy and not too hard, might be tricky. However, ignoring the fact that there exist more elegant, mathematically sound solutions is the wrong way, because such solutions always assure superior programming code. We have illustrated that in this work. Still, those bad examples can be easily improved to check the desired properties for one number at a time in a loop, but for various integers input by the user or stored in an array. Of course, this requires a few additional, preparatory statements and thus a bit more complicated example, but this is what ought to be done.

In short, many of the presented inadequate examples insist on simplifying things that cannot and should not be simplified. A plausible explanation for such bad practices is either due to the teacher's or presenter's ignorance or an attempt to achieve some educational goals with minimal teaching effort.

In several aspects, those bad practices or the elements thereof can also be found in the programs generated by the free version of ChatGPT, now a widely known and available GenAI tool. Those programs served us as a reference to what might be considered a common program code. We have shown that many of ChatGPT's solutions are far from being exemplary, hardly ever lucid, and — as we have discussed more in our previous paper on this subject [10] — often inconsistent and variable in quality. However, none of those solutions were so bad as to fail the big O-notation time complexity of the exemplary algorithms. They were just possibly slower by a certain, usually small factor. On the other hand, with more and more "correct" but mediocre solutions on the Web, the quality of the ChatGPT- and other GenAI-produced programming content could, besides improving, also deteriorate, unless the collection of that content is strictly supervised by experts. Even the newest versions of AI tools continue to show deficiencies in their "programming practices" and the resulting source program code, suggesting that some prevalent coding patterns, although suboptimal or just clumsy, stubbornly remain in those GenAI systems, instead of being improved and substituted by better programming models. However, we touched on this topic only lightly in this paper.

Finally, we have shown very bad and inefficient pieces of program code that could result from hasty, thoughtless, and untested programming. The students must learn from the beginning, and the professionals must always keep in mind that the aim of programming is not just to obtain programs "that work!" This is simply not enough, and a great deal of computer science is about teaching us just that. Programmers have to pay attention to how well their programs are written, how efficient they are,

how easily they can be updated, and how successfully their procedures (functions) can be reused. Without paying attention to every single step of the algorithms that they implement in their programs, as well as the correctness and efficiency of every single statement by which they implement those algorithms in their computer programs, neither students nor programming professionals can achieve good results in this discipline. A trivialized approach to learning programming cannot contribute to anyone's true programming knowledge and skills. It can only worsen their habits in writing program code.

In addition to the basic and most common bad programming practices presented herein, there are many more from all aspects of this area that are more subtle and perhaps not so obvious. Well-educated and trained programmers should easily recognize and avoid them. Hopefully, this might also refer to the careful readers of this paper who were assiduous enough to come to the end of it. We leave the exposition and analysis of those cases, as well as the study of their implementation in some other computer languages, for a possible future continuation of this topic.

**Acknowledgment**

This work was supported by the authors' affiliating institution, which is funded by the Ministry of Science, Education, and Youth of the Republic of Croatia.

**Contributions**

The first author gave the concept of the paper and provided the majority of its contents. The second author contributed to this research by checking the provided program code, collecting the results produced by the GenAI tools, and measuring the execution times of the analyzed code snippets.

**References**

[1] D. Knuth, The Art of Computer Programming, 3rd ed., Vol. 1, Addison-Wesley, Reading, Mass., 1997

[2] R. Logozar, M. Horvatic, I. Sumiga, and M. Mikac, "Challenges in Teaching Assembly Language Programming — Desired Prerequisites vs. Students' Initial Knowledge," 2022 IEEE Global Engineering Education Conference (EDUCON), Tunis; p.p 1689-1698, doi: 10.1109/ EDUCON52537.2022.9766737. [↗]

[3] C. S. Horstmann, Mastering Object-Oriented Design in C++, John Wiley & Sons, New York, 1995

[4] Wkp. article: "Magic number (programming)" [https://en.wikipedia.org/wiki/ Magic_number_(programming)]

[5] Wkp. article: "Naming Convention (Programming)" [https://en.wikipedia.org/ wiki/Naming_convention_(programming)]

[6] K. Gregory, Using Visual C++ 5 (Special Edition), Que Corporation, Indianapolis, IN, 1997

[7] Wkp. article: "Hungarian Notation" [https://en.wikipedia.org/wiki/Hungarian_notation]

[8] Wkp. article: "Fortran" [https://en.wikipedia.org/wiki/Fortran]

[9] Quora forum topic: "What impact does the length of variable names have in different programming languages?" [↗]

[10] R. Logozar, M. Mikac, and J. Hizak, "ChatGPT on the Freshman Test in C/C++ Programming," 2023 IEEE 21st Jubilee International Symposium on Intelligent Systems and Informatics (SISY), Pula; p. 255-264, doi: 10.1109/SISY60376.2023.10417871. [↗]

[11] B. W. Kernighan and D. M. Ritchie, C Programming Language (1st and 2nd Editions, respectively), Englewood Cliffs, NJ: Prentice Hall.D. M., 1978, 1988

[12] B. Stroustrup, The C++ Programming Language, 3rd ed., AT&T Labs, Murray Hill, New Jersey, 1997-2000 (2003)

[13] T. H. Cormen et al., Introduction to Algorithms, 3rd edition, MIT Press Cambridge MA, 2009

[14] Brilliant (Learn by doing) article: "Gauss: The Prince of Mathematics" [https://brilliant.org/wiki/gauss-the-prince-of-mathematics]

[15] StackExchange forum question: "Are there any problems with using continue or break?" [https://softwareengineering.stackexchange.com/questions/434124/ are-there-any-problems-with-using-continue-or-break]

[16] R. Logozar, "Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees," Computer Technology and Application (David Publishing), Vol. 3, No. 5, May., pp. 374-382, 2012 [Avail. at: http://bib.irb.hr/ datoteka/718251.RLogozar_RecrAndNonRecrsTravrAlgForDynCrtdBinTrs.pdf]

[17] R. Logozar, "Algorithms and Data Structures for the Modeling of Dynamical Systems by Means of Stochastic Finite Automata," Technical Gazette (Tehnički vjesnik), Vol. 19, No. 2, Apr. –Jun., pp. 227-242, 2012 [Available at: https://hrcak.srce.hr/en/clanak/124744]

[18] R. Logozar, M. Mikac, D. Radosevic, "Exploring the Access to the Static Array Elements via Indices and via Pointers — the Introductory C++ Case Expanded," Journal of Information and Organizational Sciences, Vol. 48 (1), pp. 49-80, June 2024 [Available at: https://hrcak.srce.hr/en/317974]

## Appendix A. Arithmetic Progression and Series

An *arithmetic progression* or *arithmetic sequence* is a sequence of numbers, $a_1, a_2, \ldots, a_i, \ldots$, defined by the initial $a_1$ term (member) of the progression, and the constant difference $d = a_{i+1} - a_i > 0$ between the neighboring terms, both of which are real numbers: $a_1, d \in \mathbb{R}$.

The $n$-th term of the arithmetic progression $a_n$ is

$$a_n = a_1 + (n-1)d\,. \tag{A.1}$$

The sum $S_n(a_i, d)$ of $n$ consecutive members of an arithmetic progression, from generally $a_1 = a_k$ till and including $a_n = a_{k+n}$, where $k, n \in \mathbb{N}$, is called *arithmetic series*. It amounts to

$$S_n(a_i, d) = \sum_{i=k}^{k+n} a_i = n\,(a_1 + a_n)/2. \tag{A.2}$$

With $a_1 = 1$ and $d = 1$, geometric progression reproduces the set $\mathbb{N}$ of natural numbers, for which $a_n = n$ and its arithmetic series is

$$S_n(1,1) = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \mathrm{i} = n\,(n+1)/2. \tag{A.3}$$

As a further example of the use of eq. A.2, the sums of the first $n$ odd and $n$ even *natural* numbers are:

$$S_n(1,2) = \sum_{i=1}^{n} [1 + 2(i-1)] = n^2. \tag{A.4}$$

$$S_n(2,2) = \sum_{i=1}^{n} [2 + 2(i-1)] = n(n+1). \tag{A.5}$$

E.g., for the sum of odd integer numbers greater than or equal to $-100$ and less than or equal to $300$: $a_1 = -99$, $a_n = 299 = a_1 + (n-1)d \Rightarrow n = 200$.

$$S_{200}(-99,2) = 200\,(-99 + 299)/2 = 20\,000.$$

$$S_{200}(-99,2) = S_{100}(101,2) = 100\,(101 + 299)/2 = 20\,000.$$

For the sum of even numbers greater than or equal to $-100$ and less than or equal to $300$: $a_1 = -100$, $a_n = 300 \Rightarrow n = 201$.

$$S_{201}(-100,2) = 201\,(-100 + 300)/2 = 20\,100.$$

$$S_{201}(-100,2) = S_{100}(102,2) = 100\,(102 + 300)/2 = 20\,100.$$

## Appendix B. Modulo Operation

Modulo operation (mod) is implicitly defined by the following expression,

$$k = (k/m) \times m + k \bmod m\,, \tag{B.1}$$

in which $k$ and $m$ are integers, $m \geq 1$ ($k \in \mathbb{Z}$, $m \in \mathbb{N}$), and division is the integer division.

*Definition:* $k$ is divisible by $m$ if and only if $k \bmod m = 0$.

Let $k_0$ be divisible by $m$. Then the numbers that are also divisible by $m$ are $k_0 \pm lm$, where $l = 1, 2, \ldots$ .

*Proof.* The operation $k \bmod m$ defines exactly $m$ classes of equivalence, with the results $k \bmod m = 0,\ 1,\ \ldots\ ,\ m-1$. If $k_0 \bmod m = 0$, then $k_0 = (k_0/m) \times m$, and also:

$$k_0 \pm lm = (k_0/m) \times m \pm lm = (k_0/m) \times m \pm (lm/m) \times m$$
$$= [(k_0 \pm lm)/m] \times m\ .$$

This implies that $k_0 \pm lm$ is divisible by $m$. Q.E.D.

The numbers $k_{0,\mp m}$ divisible by $m$ and closest to $k_0$ are those for $l = 1$: $k_{0,\mp m} = k_0 \mp m$.