International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 10 (2015) pp. 26103-26114
© Research India Publications
http://www.ripublication.com

Programming Style In Introductory Programming Courses

Teodosi K. Teodosiev[1] and Anatoli M. Nachev[2]
[1]Department of Mathematics and Computer science, Shumen University "Bishop K. Preslavski", Universitetska 115, Shumen, Bulgaria, e-mail: t.teodosiev@fmi.shu-bg.net
[2]Cairnes Business School, National University of Ireland, Galway, Ireland, e-mail: anatoli.nachev@nuigalway.ie

Abstract

This work addresses some issues in introductory programming courses. We discuss an essential attribute of the computer program, which is rarely paid attention to in the first steps of the programming courses – the programming style. The study discusses approaches for incorporating programming style consideration into the introductory programming courses. This is about how that initial training in programming builds foundations for correct algorithmization and program composition.
We argue the definition of this term and discuss the benefits of the programming style and its role in the programming training. Style of programming is presented as a style of thinking manifested in the ability to outline the algorithm in a form, suitable for solving the problem in the terminology of a specific programming language.
We also provide illustrative examples of programming style that can be incorporated in teaching materials of traditional "Introduction to Programming" courses.
Here we don't focus to mastering skills of professional programmers, instead we focus to solidifying skills of 'apprentices'.

Keywords: Algorithm; Introduction to Programming; Programming style; Teaching.

Introduction

Usually, introductory programming courses do not rely on previous knowledge in algorithms or any programming experience. Composing algorithms and their implementation as a computer program is a creative work. Some formal methods for algorithm implementation and proving its correctness are topics of more advanced programming courses that follow. Everything, starting with sketching the algorithm

and finishing with the choice of language for its coding depends on the choice, preferences, and the people's creativity.

According to [1], programming is a process that 'some call it the art of constructive thinking'. The creative nature of the programming suggests that seeking appropriate style of programming is applicable, similarly to other areas of creativity, such as art, literature, fashion, architecture, etc.

What does make us to compare the programming to the other styles, such as art, architecture etc.? In art, style is a specific approach and original 'certificate' of authorship. Closest to the programming style is the architecture style. Similarly to how an architect prepares a project blueprint, which then is implemented by engineers exactly and without improvisation, a programmer composes a program that the computer runs exactly, without improvisations or whatsoever amendments.

Further to that, the programming style offers practical benefits: first, it makes a program clearer and easy to read, given that the program correctness and efficiency are not affected; secondly, the risk of making mistakes minimize; and finally, the style helps to improve the program efficiency in terms of computing resources, such as CPU and RAM usage.

The paper is organised as follows: Comments on various definitions of what programming style are provided in Section 1. Section 2 addresses elements of the programming styles and the benefits of learning style issues. Section 3 provides a rationale of emphasizing on the programming styles in the introductory programming courses. Good program examples, which address the issues discussed, are presented in Section 4. These examples are organised according to major topics the course syllabus.

Programming Style

As noted before, the programming is a creative process attributed by an important characteristic called 'style'. As far as 'style of programming' (or programming style) is an informal concept of a very high order, providing unarguable definition is nearly impossible. However, there are several elements common to a large number of programming paradigms[1] and therefore, it can be characterised descriptively, similarly to the other paradigms. Definitions of programming styles can be given emphasizing on elements, important from authors' point of view.

Regarding the content of the term 'style of programming', there is some inconsistency among works in the area, which is influenced by factors, such as the current stage of development of programming; the time in which those works have been written; and the objectives and instruments used. In most of the cases, these differences can be seen as a broader interpretation of the concept' style of programming'.

[1] Programming paradigm is a set of ideas and concepts, which define the style of writing a program. The programming paradigm determines the terms in which the programmer describes the program logic. For example, in imperative programming, a program is described as a sequence of actions, in functional programming it is presented as a set of expressions and function definitions. In the popular OO paradigm, a program is seen as a set of interacting objects. It should be noted that the programming paradigm is not associated with the programming language used - many modern languages allow using different paradigms [7].

The programming style is widely understood as *typographical differences* between codes. It is related to the program readability. 'Programming style is a term used to describe the effort a programmer should take to make his or her code easy to read and easy to understand' as stated in [2].

The programs should be designed in such a way that they can be read first from people rather than machines, who may need to understand them in order to further develop the programs or make necessary amendments.

According to You [3], the style of programming is not only typographic style of code. 'This style has direct impact on the performance as it affects the use of algorithms, implementation and control flow constructs'.

According to Van Tassel [4] under style of programming we mean a set or methods of programming, which are used by experienced programmers to obtain *correct, efficient, easy to implement and easy to read programs.*

When a programmer masters a certain programming style, their programs become significantly easier to understand. This view is shared by Bodrikov [5], who says: 'The style, in my opinion, is the key to composing nice, well-tested programs. Thus, if the programmer strictly adheres to a style, then he *"sees" better their program, makes fewer mistakes, and passing through the code doesn't turn into a nightmare'*.

Common in the above definitions is that style of programming is presented as a style of thinking manifested in the ability to outline the algorithm in a form, suitable for solving the problem in the terminology of a specific programming language. Unfortunately, looking at style of programming as style of thinking is not fully realised by all. Often, programming style is considered as a technology programming. Ability to write nice and elegant programs is a very important skill that is not always in the focus of computer science and programming courses. A good observation about the programming style is made by Kernighan and Pike [6], who state: 'In a world of …relentless pressure for more of everything, one can lose sight of the basic principles-simplicity, clarity, generality-that form the bedrock of good software'.

Characteristics of Programming Style

Characteristics of programming style can be divided into three groups:

- Related to readability (*indentation, spaces, blank lines, length of code*) and standardization of the code (*naming conventions, comments, etc.*). This is usually called coding standard. Recommendations on readability are most often discussed in the literature. The rules are mainly prescriptive and not mandatory, which means they do not affect the correctness and effectiveness of the program. However, narrowing the term 'style of programming' to those rules only would be incorrect. The style that a programmer adheres to shows up during runtime of the program.
- Related to debugging (*left comparison; initialization; scope of variables and loop parameters*) and avoiding bugs (*using parentheses to avoid ambiguity; compound statement; unnecessary semicolon; avoiding out of scope bugs; out of array boundaries bugs*) [8]. These are recommendations of how poor programming style can cause common errors, frequently made by novices, or at least drawing attention to them as 'danger zones' in the program. The

introduction to programming as introduction to any subject inevitably addresses issues related to errors and their occurrence. The errors can be part of the natural process of building knowledge and skills in certain area. The aim the teaching methodology is to understand the most common problems that the students are exposed to, and through that understanding to find an approach to avoid or at least control these problems.

- Increasing efficiency (*optimization of internal loops; moving invariant components out of the loop, change the data presentation*) and ensure correctness of the program (*by avoiding: equality of real numbers, integer arithmetic errors, loss of accuracy from implicit type conversion and mixed expressions; breaking task into subtasks*) [8]. Recommendations related to the selection of accurate and efficient algorithm, breaking a task into subtasks in order to save time and memory, consideration of relations between machine arithmetic and the programming language. Therefore, these recommendations are most important for the program quality from one hand, but from the other - most difficult to realize.

The style of programming is one of the aspects of software quality. According to Mohan and Gold [9], style of programming is one of the major problems in software maintenance. The same point of view is stated in [10]: '…is a critical component of programming quality, and many authors offer specific guidance about what good style should, or should not, be'.

At the same time, introductory programming courses rarely comment on the meaningful elements of the programming style, or this is done at the very end of the course. Questions on how programs are written are they efficient, are they easy to read and understand, are either neglected, or addressed to a small cohort of motivated students. Inclusion of such questions into the course is often overlooked, hoping that students can learn these issues on their own.

Perhaps this approach makes sense – first, studying the instruments and then mastering skills for handling them. However, taking into account the fact that re-learning and replacing already acquired bad skills is very difficult and slow, the methodical approach of parallel learning of programming style and programming languages and instruments appears more reasonable and beneficial.

If introductory programming courses pay attention to the style of programming to some extent, that is primarily towards shaping the code and making comments, hoping that students will learn the rest on their own looking the 'stylish' programs as models, which they take from textbooks or electronic sources.

Reviewing the publications on this topic, we noticed that the discussion on style concerns recording the source code. Narrowing down the term programming style to those issues only, however, would be inappropriate as these stylistic features are overlooked due to the fact that they are just recommendable.

Our interest here is focused to those elements of the style, which influence readability, performance, and correctness of the program.

Place in the Course Content

In terms of positioning of the training of programming style in the course timeline, the prevailing opinion supported by studies in the area and common practices is that programming style should be scheduled as following the core topics.

In the preface to his book [4] the author made several findings, which today have many supporters:

- Style is commented to students who already know how to program;
- Style issues are rarely commented in introductory programming courses;

There is another point of view. For many things in life, first impression is most important. Programming is not exception. Support to that view is made in [11]: 'Yet few programmers have ever been taught what style is, as we can see from even cursory inspection of their code'.

Our personal teaching experience and that of other colleagues [12, 13, 14] shows that in programming, initial training plays an essential role in the efficiency and quality of programming.

Grigas [12] analyzed the medalist of IOI'94 and commented that the best students impresses by a good programming style. He also argued that the program readability has a positive impact on its correctness and that assessment on the result only was criticized because performance only does not reveal the main features of the algorithm.

Skūpienė [13] also argues that programs with good and poor style should be assessed differently. It is much easier to develop skills for good programming style with students who have no prior programming experience and hopelessly hard to get students to adopt good programming style once they have written programs with bad style for several years.

As stated in [15] '…my main concern in teaching students computing science was to train to think first and not to rush into coding…' It is obvious that novice programmers have long enough to train with 'pencil and paper' developing of elements of the algorithm logic, proper construction of the loops, etc. before they are allowed to enter code of a usable program.

Examples

Further to the discussion above and considering the fact that it is more difficult to change a skill than to create one, we favour the methodical approach, which incorporates teaching of programming style issues as soon as possible and throughout the introductory programming course. Elements of style can be taught in nearly all major topics of the course.

Let's consider a few examples, which enable learning algorithmization and language specifics while paying attention to the programming style. A common teaching approach of gradual increasing of the algorithm complexity will also be illustrated.

Topic: 'Variables, Data Types and Expressions'.

Writing a program must be tailored to the characteristics of computer calculations. This not only illustrates a good style, but also protects from making mistakes in calculations.

A common mistake made by beginners is evaluating mixed expressions containing integer division. Students can understand that the result is the whole quotient if they are to assigned it to an integer variable, but not easy understand the loss of precision assigning the expression value to a variable of type double or float.

After introducing the language basics and program structure, which includes data types, operations, expressions, and assignment, the following example can be considered:

Example 1: Write a program, which enters a decimal value for the variable x of type double and calculates: $\frac{1}{2}(x^3-2x^2+7)$.

```
//Example 1
//version 1
int main()
{
 double x, y;
 cout << "Enter number:";
 cin>>x;
 y = 1 / 2 *(x * x * x – 2 * x * x + 7);
 cout << y <<endl;
}
int main()
{
 double x, y;
 cout << "Enter number:";
 cin >> x;
  //version 2
 y = (double) 1 / 2 * (x * x * x – 2 * x * x + 7);
  //or version 3
 // y = 1.0 / 2 * (x * x * x – 2 * x * x + 7);
 cout << y <<endl;
}
```

Note: Be careful when using mixed expressions as integer division evaluates to integer result. The output of version 1 above is 0 regardless of the x value. Avoiding this problem is illustrated in version 2, which uses explicit type conversion (casting), or implicit type conversion (coercion), illustrated in version 3.

Topic: 'Boolean Values. Conditional Statements. Multiple Selection and Switch Statements'.

Example 2: Write a program that enters a string interpreted as arithmetic expression and calculates it. The string consists of 5 characters: the middle one is an arithmetic operation; all others are digits forming two 2-digit numbers.

```
//Example 2
//version 1
int main()
{
 int arg1, arg2;
 char ch1, ch2, ch3, ch4, op;
 cout << "Enter string:";
 cin>> ch1 >> ch2 >> op >> ch3 >> ch4;
 arg1 = (ch1 - '0')* 10 + ch2 - '0';
 arg2 = (ch3 - '0')* 10 + ch4 - '0';
 if (op == '+') cout << arg1 + arg2 <<endl;
 if (op == '-') cout << arg1 - arg2 <<endl;
 if (op == '*') cout << arg1 * arg2 <<endl;
 if (op == '/') cout << arg1 / arg2 <<endl;
}
//version 2 (fragment)
if ('+' = op) cout << arg1 + arg2 <<endl;
if ('-' = op) cout << arg1 - arg2 <<endl;
if ('*' = op) cout << arg1 * arg2 <<endl;
if ('/' = op) cout << arg1 / arg2 <<endl;
```

In languages that use single '=' for assignment and double '==' for comparison (e.g. C, C++, C#, Java, PHP, Perl, etc.), where comparisons is used within control structures, it is makes sense arranging literals as left operands, e.g. (' ' == ch). Left-comparison expressions, such as those used in version 2, would a compilation error in omission of one '=' character (marked in gray above), which is a common mistake made by beginners. This wouldn't happen with right (standard)-comparison as in version 1. That example would go through compilation, which does not detect the error. Spotting and fixing such an error becomes difficult as the program runs, but not correctly – it has a logical error.

Topic: 'Repetition structure (Loops)'.

The first examples of this topic could be computing well-known series, such as (sum of the squares, sum of harmonic series, etc.). To solve simple tasks students tend to follow previously given examples and models. Thus, teachers could expect that showing a solution of one task will influence solution of the following one.

These tasks are very similar previous ones and the first that springs in mind is to apply the same 'model'. Indeed, reproductive methods are used as methodical approaches in introductory programming courses, which encourage students to use components of previous solution in new one.

Programmers do not always assign initial values to the variables, relying on the compilation process for this. One of the challenges constructing counter-controlled loops is setting the initial and final value of the control variable(s). The choice of those values should be made carefully, because incorrect initial values can lead to erroneous results in a generally correct algorithm. Moreover, that choice should guarantee completion of the loop.

Example 3: Write a program, which enters a natural number n and calculates n!.

```
//Example 3
int main()
{
 int n,s;
 cin >> n;
 for (int i= 1; i<= n; i++)
   s = s * n;
 cout << s << endl;
 }
```

Not initializing variable s in this example leads to incorrect performance of the program and unspecified output value. To avoid this error, initialization of all variables is necessary.

Let's consider a task that allows to illustrate several recommendations for improving the effectiveness of the program, and then demonstrates the usefulness of careful constructing an algorithm.

Example 4: Write a program, which enters x and n and calculates the following function: $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$

This example of approximate calculation of a series is important, because many math and physics functions can be calculated with very high precision using series. Here we present four versions of solution so that each one is simpler and clearer than the previous one. In addition to that, the first two versions allow to comment on how to optimize loops.

It seems that solving this task we can calculate successively each term and add/subtract it to the sum. But how to calculate the next term?

This version separately calculates the products of the numerator and denominator of each term of the series. These products are very similar to the 'model' of the examples above. Here we have a problem of calculating factorials, which might be very large numbers.

```
//Example 4 version 1
int main()
{
 double a = 1, x, s = 1; int i = 1, n;
  cout << "Enter x: "; cin>>x;
  cout << "Enter n: "; cin>>n;
  do
  {
   double num = 1;
   for (int j= 1; j <= 2 * i; j++)
     num = num * x;
   double denom = 1;
   for (int j = 1; j <= 2 * i; j++)
     denom = denom * j;
   if (i%2) a = -num / denom;
```

```
 else a = num / denom;
 s += a; i++;
}
while (i <= n);
cout<< "cos(" << x << ")=" << s <<endl;
}
```

In this version we calculate the numerator and denominator of each term by a separate loop.

At this stage, a discussion should pay attention to the problem of alternative change of the sign and the error of calculation for large values of n (say n > 310).

Attempting to speed up the program run, attention should be focused to the loops as they consume much of the execution time and any optimization may have a significant effect on the overall performance, much more than the effect of an out-of-loop statement. One way to reduce the number of iterations is to merge two or more loops together. This reduces the execution time and memory needed.

This technique leads to the following version of the task solution. It can be noticed, that the numerator and denominator are calculated in the same way, allowing their calculations to be combined in one loop. Combining the bodies of the separate loops reduces the amount of calculations, improves the program efficiency and shortens the code.

```
//Example 4 version 2 (fragment)
do
{
 double  num = 1, denom = 1;
  for  (int  j = 1; j <=  2 * I ; j++)
   {
    num = num * x;
    denom = denom * j;
   }
 if (i%2) a = -num / denom;
 else a = num / denom;
 s += a;
 i++;
}
while (i <= n);
```

It would be better if each loop's iteration can use the results obtained in the previous iteration. This is illustrated in the following version 3, which employs the fact that each term can be calculated by multiplying the previous one by (x / j),

```
//Example 4 version 3 (fragment)
do
{
 a = 1;
 for (int j = 1; j <= 2 * i; j++)
  {
   a = a * x / j;
```

```
  }
 if ( i % 2) s -= a;
 else s += a;
 i++;
}
while (i <= n);
```

Finally, version 4 illustrates implementation, which involves recurrence relation between two terms of the series. Revealing this relation, in our opinion, is the biggest challenge for students when they go through the topic 'Recursion'. The difficulty can also be explained by lack of understanding of the recursive math techniques studied in school.

Finding relations between two subsequent computations is a prerequisite of finding the recurrence relation, which is part of the topic 'Recursion'. Let's denote the general term of the series as follows:

$$a_n = (-1)^n \frac{x^{2n}}{(2n)!}\ ,\ then\quad a_0 = 1 \qquad a_{n+1} = -a_n \cdot \frac{x^2}{2n(2n-1)},\ for\ n = 1,2,\ldots.$$

which is the recurrence relation of the general term.

```
//Example 4 version 4
int main()
{
 double a = 1,x, s = 1; int  i = 1,n;
 cout << "Enter x: "; cin >> x;
 cout << "Enter n: "; cin >> n;
do
{
  a = -a * x * x /(2 * i *(2 * i - 1));
  s += a;  i++;
}
while (i <= n);
cout<< "cos(" << x << ")=" << s <<endl;
}
```

Of course, the tasks considered do not comprise the diversity of methodical approaches of training programming style to beginners. They just illustrate that there are no prerequisites to consider appropriate examples extending the programming style issues to other topics, such as subroutines / functions / methods, arrays, etc.

Conclusion

Novice programmers often underestimate the style of programming considering it as time-consuming and not helping in solving problems. For them, it is often difficult to understand the meaning of the guidelines for programming style, while trying to concentrate on the creation of the first programs.

In this paper, we argue that the methodical approach of incorporating a programming style discussion into the syllabi of introductory programming courses as soon as possible is preferable and more beneficial comparing to the approach of

discussing those issues after the main course topics. We back our points with years of teaching experience and estimating results of applying different methodical approaches. We also provide a set of examples, which illustrate style issues and are ready to be used in the course.

In programming, style usually achieves unity between form and content of the program. The form should illustrate, explain and document the content. The style also has a role to ease understanding of the program in the learning process. The style of programming is directly related to the professional training of the teacher and his teaching experience.

Some may argue that certain points discussed here might be addressed by automatic tools [16], such as components of the integrated development environments (IDE), but relying on those tools contradicts to the pedagogical aspects of building a solid ground for algorithmisation skills.

Acknowledgements

This work has been partially supported by the Fund "Scientific Research" at the University of Shumen, Bulgaria – Contract № RD – 08-273/11.03.2015.

References

[1] Wirth N., 2002, Computing Science Education: The Road not Taken, ITiCSE Conference, Aarhus, Denmark, http://www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm (2/05/15)

[2] McCann, 1997, Toward Developing Good Programming Style C version, http://www.cs.arizona.edu/~mccann/style_c.html (2/05/15)

[3] You, H. L., 2009, Tool Support for Learning Programming Style, http://www.csse.uwa.edu.au/UWAJavaTools/publications/YouHai.Dissertation.pdf (2/05/15)

[4] Van Tassel, D., 1978, Program style, design, efficiency, debugging, and testing, 2d ed. Prentice-Hall, N. J.

[5] Bodrikov, S.V., 2005, C /C ++ - programming style (in Russian), http://www.codenet.ru/progr/cpp/C-Style.php (2/05/15)

[6] Kernighan, B. W. & Pike, R, 1999, The Practice of Programming. Addison-Wesley, https://docs.google.com/file/d/0B2Q8Nd2L-6PjN2I0MzEzZDYtM2JhNC00NzJlLWFhMGQtZWUyMWE0N2M4MGM4/edit?pli=1 (2/05/15)

[7] Nepejvoda, N., 2003, Programming styles as foundation of a notion system for informatics, Kluwer Academic Publishers.

[8] Teodosiev T. & Nachev A., 2012, Some Pitfalls in Introductory Programming Courses, Informatics in Education, Vol. 11, No. 2, 241–255.

[9] Mohan, A. & Gold, N., 2004, Programming Style Changes in Evolving Source Code. In: IEEE Proceedings of the 12th International Workshop on Program Comprehension, Bari, Italy, June 24–26, 236–240

[10] Glaser, A., 2009, Notes on Programming Style, Octagon Research Solutions, Inc. http://www.lexjansen.com/nesug/nesug09/as/AS03.pdf (2/05/15)

[11] Kernighan, B. W. & Plauger, P. J., 1974, Programming Style: Examples and Counterexamples. ACM Comput. Surv. 6(4): 303-319.

[12] Grigas, G., 1998, Investigation of the relationship between program correctness and programming style (draft), http://olympiads.win.tue.nl/ioi/ioi94/style.txt (2/05/15)

[13] Skūpienė, J., 2006, Programming Style – Part of Grading Scheme in Informatics Olympiads: Lithuanian Experience, ISSEP, 545-552.

[14] Tkachev, F.V., 2006, Education system as a factor of national sovereignty in the area of information technology, http://www.inr.ac.ru/~info21/texts/2006-09-SFO/v2public.htm (2/05/15)

[15] Dijkstra, E. W., 1982, Why is software so expensive? http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD648.html (2/05/15)

[16] McConnell, S., 2004, Code Complete, 2nd Edition, Redmond, Wa.: Microsoft Press