Informatics in Education, 2012, Vol. 11, No. 2, 241–255
© 2012 *Vilnius University*

Some Pitfalls in Introductory Programming Courses

Teodosi TEODOSIEV[1], Anatoli NACHEV[2]
[1]*Shumen University "Bishop K. Preslavski"*
Universitetska 115, Shumen, Bulgaria
[2]*Cairnes Business School, National University of Ireland*
Galway, Ireland
e-mail: t.teodosiev@fmi.shu-bg.net, anatoli.nachev@nuigalway.ie

Received: February 2012

Abstract. This paper discusses some difficulties in teaching introductory courses to programming, paying particular attention to their mathematical nature. We consider some aspects, which have not been commented in detail in textbooks and often neglected by course outlines and schedules. Some of these are constructing complex conditions, exceeding array bound, calculating infinite series in conjunction with recursion, etc. We believe that those topics and accompanying notes along with appropriate teaching methodology could be and should be incorporated into introductory programming courses.

Keywords: teaching, programming, condition, algorithm, infinite series, style of programming.

Introduction

Professional software development is a complex process that can be driven by various models, such as waterfall, incremental, RAD, evolutionary, agile, lean, etc. In the light of the software engineering and the software development life cycle (SDLC), the common development stages, such as system analysis, design, detailed design and algorithmization by pseudo-code, coding, testing, and maintenance, can be adapted for the purposes of the introductory programming courses as two basic stages: modelling and implementation. We can broadly define, that the programming is primarily a creative process that can be separated into two main stages:

- building a virtual model of the future program and
- implementing the model as an algorithm and coding it by using the means of the programming languages.

Building the program model is the first stage of the development process and its goal is to express the overall idea of solving a particular task. Programming is the second stage, which specifies the model in details by creating an algorithm, followed by coding. The process of transition from the first to the second stage can be considered as similar to transition from abstract thinking to articulating thoughts by speech or in writing. That transition causes the major difficulties in programming, at the same time it may surface some errors and omissions of the model itself, which eventually leads to its partial or major amendment. Borovin *et al.* (1987) state that “Connection between understanding and writing appears to be as problematic for programming, as it is for any other form of a written exposition”.

We believe that one of the main issues that an Introduction to Programming course has to address is development of a proper way of thinking, which can help building correct models and then facilitate their transition to programs. The style of programming is a style of thinking, manifested in the ability to describe the algorithm by a particular programming language (Borovin *et al.*, 1987). Unfortunately, not everyone involved in the teaching process perceives the programming style as a kind of style of thinking. Often, it is perceived as a technology of programming.

Our previous works have justified the need for developing a good programming style in the introductory programming courses. This allows students to write programs that are easy to write and then read by the others. A stylish program easily reveals its key elements that lead to the solution. Many textbooks used in the programming courses provide recommendations for code arrangement, choice of object names, etc., but they rarely discuss algorithm accuracy and efficiency. We are trying to fill that gap by discussing some common pitfalls in teaching Introduction to Programming. Of course, teaching in any area or topic cannot be perfect and inevitably makes mistakes. This is part of the natural process of acquiring knowledge and skills. Our goal is to understand the most common challenges and problems the students experience during training. That understanding would allow avoiding or at least reducing those problems.

Making errors in programming is a major issue that both teachers and students try to handle and avoid. The first errors that novices face are the syntactic ones. Capturing those errors is not a big issue as the compiler messages help to locate and correct them. A bigger challenge, however, is when programs pass successfully the compilation stage, but afterwards they output wrong results due to logical/semantic errors. Here we offer a range of advices that try to circumvent and avoid logical errors, all illustrated with examples that point the “underwater stones”. We also offer some tips on how to improve the algorithm efficiency in terms of computing time and resources used. On the other hand, our advices can be viewed as an initial step in forming a good programming style.

We should state here that we don’t underestimate the syntactic errors as they may double the number of the semantic ones and can significantly slow down the teaching and learning process.

The set of examples we provide and discuss here aim to demonstrate our view that a good math background is essential for developing a good programming style and provides a lot of advantages for those who have it. Part of our examples is related to mathematical logic and recursive relationships.

Finally, our personal teaching experience and that of other colleagues (Skupiene, 2006) shows that it is much easier to make students to develop good programming skills if they have no prior programming experience, rather than working with those who have already “bad” skills and “wrong” programming style, developed in their previous experience.

The paper is organized as follows: Section 1 discusses the problems that teachers experience in the Introduction to Programming courses. Section 2 discusses the importance of developing algorithmic thinking. Examples of characteristics of the machine arithmetic are shown in Section 3. Section 4 discusses proper construction and use of compound terms. Section 5 discusses the need to avoid recalculation of expressions. Calculation of infinite sums in preparation for the topic, "Recursion" is addressed in Section 6. Finally, Section 7 discusses problems with scope of variables.

All examples are written in C/C++.

Exposition

1. Problems in Teaching Programming

Traditionally, teaching in programming relies primarily on ready algorithms from textbooks. This approach, however, does not stimulate development of creative thinking in students. Good understanding the matter and extracting useful information and knowledge from text is not easy. Problems with reading and understanding have been clearly identified in surveys and studies carried out among Bulgarian students. According to the Programme for International Student Assessment (PISA) 2006, nearly a half of all 15-years students are reported to experience some difficulties in analyzing and critically estimating information for the purposes of applying what have been learned in practice. Those students have difficulties in solving geometric and text-formulated math tasks, which require abstract and logical thinking and creativity. There are no ready-made algorithms for those tasks, as well as for the programming tasks. Additional factors, such as poor general knowledge and skills, lack of abstract thinking, and insufficient basic practical knowledge, may contribute to the difficulties that students experience in creating a mathematical and algorithmic model, and thus writing a correct program.

The objectives of the course "Introduction to programming" is to teach algorithmic structures (sequence, selection, iteration, module; Radošević *et al.*, 2009), along with the utilization of a formal programming language. This task is complex, yet very important (Van Diepen, 2005; Govender, 2006). The difficulties that students experience are related to the duality in the way the programming languages are perceived: first as an instrumental tool for formal description of algorithms, secondly as a subject of study. The task of the tutors is to teach students in two directions – algorithmization and programming. Students experience difficulties in acquiring both. Introduction to Programming is a difficult subject in computer science disciplines, and even after more than two years of training uptake rate is low (Kurland *et al.*, 1989). Part of the students make effort just to pass the exam, but at the same time they acquire bad skills such as copying codes from colleagues or learning them off by heart.

This makes many teachers to explore new, non-standard approaches to teach certain topics. For example: problem-oriented training group (Kinnunen *et al.*, 2005), programmable mobile robots (Pásztor, *et al.*, 2010), software tools to support learning programming skills (Radošević *et al.*, 2009; Kiesmüller and Brinda, 2009), focusing on the graph (Djordjevic, 2007), objects at the beginning (Sajaniemi *et al.*, 2006), etc.

Teaching “’Introduction to Programming” at Bulgarian universities presumes that students have little or no background in programming. Only a few students in classes have some experience in creating algorithms from secondary schools. In second level education, however, teaching is focused to understanding, memorizing, and applying ready algorithms or pieces of code. These students, in general, don’t have real and deep understanding of the algorithm meaning and details. Therefore, in order to create computer programs, alongside the programming language, students must learn the principles of programming.

For instance, first year students from the programme “Informatics” at Shumen University have different background from secondary school. Some of them get a taste for software design, others not. Some have a good math background, others state that they wish to study computer science and programming, but no math.

2. Development of Thinking

According to Dijkstra (1982), “Excellent mastery of his native tongue is my first selection criterion for a prospective programmer; good taste in mathematics is the second important criterion”.

A programming process requires first finding a solution of the problem in conceptual and abstract terms and then presenting the solution to the machine as a rigorous algorithm which does not allow whatsoever ambiguities and interpretations in terms of form, syntax, and grammar (Papert, 1980; Szlávi and Zsakó, 2006). Programmers should be able to express algorithms clearly in both natural and formal languages. Mastery in expressing solutions using natural language is important because first, this is the language of description of the task itself and the language of possible modifications of the problem. Secondly, because the formalization follows notion of the solution articulated in “natural” terms and concepts. And finally, because software engineering often uses pseudo-code for algorithm description, which is a mid-way between the natural and formal languages. In other words, getting the task, the programmer must create and define the theory needed to justify the algorithm. While working (s)he is forced to invent their own formal apparatus. This helps students to develop their natural language skills, because they must learn to express themselves clearly if they want the computer – unintelligent machine to do what they want (Hromković, 2006).

The ability of correct formulation of problems is based on clear logic, systematic design of programs, and cannot be method of “guesswork” using debugger (Dijkstra, 1995; Wirth, 2002). What kind of thinking is necessary for structural programming?

“As a matter of fact, the challenges of designing high-quality programs and of designing high-quality proofs are very similar that I am no longer able to distinguish between the two: I see no meaningful deference between programming methodology and mathematical methodology in general. The long and the short of it is that the computer’s ubiquity has made the ability to apply mathematical method more important that ever.” Such an interesting analogy between programming and mathematics is made by Dijkstra

(1995). The main emphasis is on accuracy and effectiveness of the program, which requires mathematical skills and preliminary work before the programmer sits in front of the computer.

A program structure is mostly result of correct formulation of the sequence of actions and ability to ignore minor details. A good program structure indicates that it's creator has performed structured and logical thinking and has ability to differentiate between significant and insignificant aspects of the problem in question, at the same time understanding its stepwise formulation.

Govender (2006) determined three main technical aspects that students must learn: data, instructions, and syntax. The data relates to two concepts – variables and data types. The instructions are related to understanding how control structures and subroutines should be constructed. Syntax denotes a set of rules that define what is allowed and what not in the programming languages. Syntax rules define how to build programs using constructs, such as loops, branches and subroutines.

3. Features of Machine Arithmetics

Due to differences between the machine arithmetics and the traditional one, students may experience some difficulties in creating algorithms and writing code. In some cases the problems become evident at the stage of writing code, but in other cases the problems pop up during the compilation and runtime.

In a situation where students have been introduced to the algorithmic construction sequence, internal representation of real numbers in computer memory, they already know that floating-point numbers have precision. The students, however, often neglect this fact, as they believe that computers are error-free, quick, and correct. Teaching allows addressing this issue and showing what problems may emerge. This could be demonstrated by one of the first example programs.

$\surd$ *Avoid arithmetic operations with numbers, which have a large difference between their orders of magnitude*
After introduction to basic structure sequence, basics of a programming language, basic data types, arithmetic operations, built-in functions, and assignment statement, the following example can be considered:

EXAMPLE 1. Calculate value of $10^n + 1 - 10^n$, given n (integer) is input.

Our experience shows that students' first impression is that the task is quite easy to be solved and even a bit boring. In fact, they don't realize the importance of the example and the challenges that may emerge. In connection with that and in order to avoid potential problems, we recommend that at this stage the teacher should consider some aspects of the machine arithmetics.

Tests show that in version A if $n <= 19$ output is 1, otherwise 0. In version B, expression $10^n - 10^n{+}1$ is calculated and output is 1 no matter what n is.

Version A
```
int n; float e;
cout≪"Enter n =?";  cin ≫ n;
e = exp(n * log(10));
cout≪"Result:"≪ e + 1 − e ≪"\n";
```

Version B
```
int n; float e;
cout≪"Enter n =?";  cin ≫ n;
e = exp(n * log(10));
cout≪"Result: "≪ e − e + 1 ≪ "\n";
```

The example above illustrates that in computer arithmetics, arrangement of operands does matter, in contrast to certain math rules and laws, which state that the arrangement of operands does not matter. Another conclusion can be made here – the operation addition should be applied to numbers with equal or at least similar order of magnitude, otherwise calculations loose precision.

√ *Avoid using mixed up data types*

EXAMPLE 2.
```
int r, a, d, n;
double p;
  cin ≫ a ≫ r;
  cin ≫ d;
  p = 6 * r * a/2 − 3.14 * d * d/4;
  cout≪ p ≪endl;
```
Variable p assigns value of the expression $6 * r * a/2$, which is an integer. This may cause at least two problems: the value exceeds boundaries of the data type integer, which would end up with a wrong output; division is integer, therefore the result is integer too.

Most often, such errors occur when dividing integer operands. Students can easily understand that the result will be cut if stored in an integer variable, but it isn't obvious for them why that result may loose accuracy when stored in a variable of type double or float. That misunderstanding is caused by the discrepancy between their math background and the rules that the computer implementation of arithmetic operations imposes.

In order to familiarize students with the iteration and loop statements, we can use the two examples below. Note that the students should keep in mind that values of real constants, variables, and/or functions are approximate, not exact, due to their fixed precision. The examples below illustrate that comparison for equality between two real expressions is generally speaking wrong. In contrast, that comparison between two integers is correct, as the operands are exact values. Anyway, the question 'what should we do if the task requires comparison between real values' still remains unanswered.

√ *Avoid comparison for equality of real values*

EXAMPLE 3. Write a program that outputs in a table form values of $\sin(x)$ in [1,2] with step of increment 0.1.

Version A
```
float x = 1;
do
  {cout<< x <<" "<< sin(x) <<endl;
  x+ = 0.1;
  }
while (x <= 2);
```

Version B
```
float x = 1;
cout<< x <<" "<< sin(x) <<endl;
do
  { x+ = 0.1;
    cout<< x <<" "<< sin(x) <<endl;
  } while (x < 2);
```

Version A of the example above seems to be more "natural", but the program does not calculate the last value of $sin(x)$ in $x = 2$. Version B illustrates how comparison for equality can be avoided – it outputs $\sin(x)$ every time value of x is incremented.

√ *Be careful when divide integers as the result is also an integer*

EXAMPLE 4.
```
int n; double s = 0;
  cin>> n;
  for (int i = 1; i <= n; i + +)
  s = s + 1/i;
cout<< s <<endl;
```

This example illustrates how a syntactically correct algorithm may cause wrong calculations, which can be detected at runtime only. The result is always 1 regardless of n, which is due to using integer division.

In order to avoid wrong results when calculations require real operands:

√ *Initialize all variables (see underlined code in Example 12)*
√ *Use brackets in order to avoid ambiguities in calculation (see double underlined code in Example 12).*

4. Compound Conditions

Considering control structures, teachers can pay particular attention to the topic correct construction of conditions (Boolean expressions) and their proper use in control structures. The simplest forms of conditions are relations. Compound conditions can be constructed by combining relations with logical operations. The following examples illustrate the importance of correctly constructed compound conditions and their usage.

√ *Use left-hand comparison*
Popular programming languages, such as C, C++, C#, Java, PHP, Perl, etc., use single symbol (=) to denote their assignment statement and double (= =) for comparison operation. These languages also allow using the assignment statement within control structures, such as if, while, etc. That may cause confusion in students, as misuse of single and double symbols cause unexpected errors, which are impossible to be detected by the compiler and the only way to be captured is at runtime (Spolsky, 2005). In order to

avoid these problems, students can be encouraged to use constants and expressions as left operands of the comparison operator. For short, we call that operation left-hand comparison. That advice, however, does not have counterpart in the math – students have been taught that it doesn't matter if a comparison operand is left-hand or right-hand. To some extend, learners may feel confused of that.

EXAMPLE 5. Use `if ('!'==ch|| '.'==ch || '?'==ch)`, instead of `if (ch=='!'||ch=='.'||ch=='?')`.

In math position of operands does not matter, but in programming left-hand comparison would help in finding errors.

Compound conditions also can be used to help to avoid nesting of two or more programming structures.

EXAMPLE 6. Write a program that inputs year and outputs whether it is bissextile.

Note: An year is bissextile if

1. The last two digits are 0 and the first two are multiple of 4.
2. For all other years if they are multiple of 4.

Version A
```
void main()
{int g;
 cin≫ g;
 if (g%400 == 0) cout≪"yes \n";
 else if (g%4 == 0)
    if (g%100) cout≪"yes \n";
    else cout≪"no \n";
  else cout≪" no \n";
}
```

Version B
```
void main()
{int g;
 cin≫ g;
 if (g%400 == 0||g%4 == 0
           && g%100)
       cout≪" yes \n";
 else cout≪" no \n";
}
```

Version A uses nested condition statements, in contrast to Version B, which uses compound conditions.

Composing compound conditions should take into account the fact that conjunction (logical AND) estimates its operands from left to right by reaching the first false value, whereas disjunction (logical OR) does the same, but by reaching the first true value. By rearranging operands so that the simpler and easier to calculate ones appear to be leftmost, we achieve efficiency of the algorithm. In the case of disjunction, if the operand with most likely true value is leftmost, that would probably complete the calculations in an early stage; in the case of conjunction, the leftmost operand should be the one with most likely false value. Those considerations are very important when compound conditions are estimated many times, for example being part of loops.

In summary, students should pay particular attention to the following:

√ *Order of estimation of conjunction and disjunction operands*

Sometimes, arrangement of operands affects only algorithm efficiency and can by neglected, but in other cases wrong arrangement can lead to errors. The following example illustrates that.

EXAMPLE 7. Write a program that inputs a sequence of real numbers until entering a number which natural logarithm is negative. The program should output how many numbers have been entered.

```
void main()
{ float x;
  int br = 0;
  do
    { cin≫ x;
      br++;
    }
  while (x > 0 && log(x) > 0); //(1)
  cout≪ br − 1 ≪endl;
}
```

The compound condition of the do-while loop has to be arranged in that way, otherwise the program will output error as it attempts to calculate logarithm of a negative value. Similarly to what math rules say, an estimation of a function should be preceded by a check if argument values belong to the function domain.

When teachers familiarize students with the data structure array, they can pay particular attention to the following pitfall:

√ *Check array bounds*

Incorrect arrangement of compound conditions can lead to 'exceeding array bound' error or if not captured by the compiler can lead to even worse results and wrong calculations.

EXAMPLE 8. Write a program that inputs currency rate of the US dollar against euro for every day of a month and outputs the first day which counts falling rate.

```
void main()
{int k,i; float a[31];
..............
i = 0;
do i++;
while (i < 31 && (k = a[i − 1] <= a[i]) );
if (k) cout≪" no fall in the rate "≪ endl;
else cout≪" first day with fall in the rate "≪ i + 1 ≪endl;
}
```

If the two operands of the underlined compound condition above are swapped, given that a month has 31 days and each day satisfies the condition, this would produce an exceeding array bound error (languages as C and C++ do not control getting out of the index bound which makes it possible for programs to output wrong results).

Next observation: if array elements are arguments of functions and their values are out of the function domain (as shown in Example 7), the program can output error. And finally, it is evident that the first operand is easier to calculate.

$\surd$ *Using Boolean data type*

Using the Boolean data type is an interesting and important topic for consideration. The following example shows implementation of a branched algorithm, which does no use conditional statements.

EXAMPLE 9. Calculate the following function $y(x) = \begin{cases} x+2, & x<0, \\ x^2+2, & 0 \leqslant x < 1, \\ 3x, & x \geqslant 1. \end{cases}$

Version A

```
void main()
 {float x,y;
  cout≪" Enter x "; cin≪x;
  if(x < 0)y = x + 2;
  else if(x >= 1) y = 3 * x;
      else y = x * x + 2;
  cout≪" y("≪x≪")="≪y≪endl;
 }
```

Version B

```
void main()
{float x,y;
cout≪" Enter x "; cin≪  x; y  =
(x + 2) * (x  <  0) + (x * x + 2) * ((x >=
0)&&(x < 1)) + 3 * x * (x >= 1);
cout≪" y("≪x≪")="≪y≪endl;
}
```

Version B illustrates a solution using the Boolean data type and without involvement of conditional statements.

5. Avoid Multiple Recalculations

"Loops" is a basic topic in programming. Although, the nature of loops is to perform multiple executions of its body statements and expressions, repetition of exactly the same calculations in any form is senseless, wasting of computing resources, and should be avoided, where possible. We can refer to the math and state that this is valid even there – recalculation is useless. The following examples illustrate the problem with recalculations and how can be avoided.

EXAMPLE 10. Find the total of a sequence of shots to a target. The target circles have radiuses 1,2,...,10 and shot coordinates (x, y) are available. Terminate when the shot missed the target.

Version A

```
do
{cout≪"Enter coordinates :";
 i++;cin≫x≫y;
if (x*x+y*y <= 1) s+ = 10;
else if (x*x+y*y <= 4) s+ = 9;
    else if (x * x + y * y  <=  9)
s+ = 8;
        else if (x*x+y*y <= 16)
s+ = 7;
/* write if statement for others
cases */
    else if (x * x + y * y  <=  100)
s++;
}
while (x*x+y*y <= 100);
```

Version B

```
do
{cout≪"Enter coordinates :";
i++; cin≫x≫y;
double z = x*x+y*y;
if (z <= 1) s+ = 10;
else if (z <= 4) s+ = 9;
    else if (z <= 9) s+ = 8;
        else if (z <= 16) s+ = 7;
/* write if statement for others
cases */
else if (z <= 100) s++;
}
while (z <= 100);
```

By using an auxiliary variable z in version B, recalculation of $x * x + y * y$ 10 times is avoided, *in contrast to version A. This also shortens the code.*

EXAMPLE 11. Check if integer n is prime.

Version A
```
i = 1;
do {i++;} while (i < n/2+1 && n
% i);
if (i          >=          n/2    +    1)
cout<<"yes"<<endl;
 else cout<<"no"<<endl;
```

Version B
```
k = n/2+1; i = 1;
do {i++;} while (i < k && n % i);
if (i >= k) cout<<"yes"<<endl;
 else cout<<"no"<<endl;
```

Version B illustrates how using an auxiliary variable k saves recalculation of $n/2 + 1$ k times. This does not change the algorithm correctness; it just makes it faster and shorter.

6. Calculation of Infinite Series by Recursive Relations

Providing students with examples of approximate calculation of series is useful and justified, as many functions used in natural sciences today, can be represented and calculated with any accuracy by infinite series. The following example illustrates one of them:

EXAMPLE 12. Write a program that input x and ε $(0 < \varepsilon < 1)$ and calculates the function

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots \quad \text{with precision } \varepsilon.$$

This example is also a good illustration for the topic "Loops with unspecified number of iterations". It offers a discussion on the following issues:

(a) What precision ε means in the context of infinite series with unspecified number of terms.
(b) Alteration of the term sign.
(c) Our experience shows that understanding of this example is essential for understanding the concept of recursion at all.

At first glance, a possible approach to the solution could be calculating each term sequentially and adding/subtracting it to the sum. No doubts, it would be better if each step uses what has been calculated so far. That is use the value of each term in order to calculate the new one. Thus, gradually and naturally, students can be introduced to the idea of recursion and recursive relationships.

If we denote the common term of the series by

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

One of the challenging issues that arises composing loops is to figure out how to initialize parameters of the loop. Those values require careful selection, as any mistake can lead to wrong results in a generally correct program. Another issue that teachers have to address is how to guarantee loop termination.

```
void main()
{
 float a = 1, x, eps, s = 1; int i = 0;
 cout<<" Enter x: "; cin<< x;
 do
 {
  cout<<" Enter 0 < eps < 1 "; cin<< eps;
 }
while (eps <= 0||eps >= 1);
do
{ i++; a = -a*x*x/(2*i*(2*i-1));
  s+ = a;
 }
 while (fabs(a)>eps);
 cout<<"cos("<< x <<")="<< s <<endl;
}
```

7. Scope of Objects

Teaching the topic 'scope of objects' is sometimes challenging for both teachers and students. Our experience shows that the concept of scope is confusing for students, particularly when object scope is defined in do-while blocks, in the headline of for-loops, and in a function body. Teaching should facilitate students to grasp that the scope of objects span from the point of declaration/definition to the end of the block in which they have been declared. The following example illustrates how a variable (x) defined in the block of a while statement is unavailable in the while condition. At the same time, the variable (i) defined in the head line of a for-loop can be used in the block of that loop.

EXAMPLE 13.

```
int cost;
do
{
 int x = 2;
 cost = cost +x;
 x = x - 1;
 }
while(x > 0); // compile error here
for(int i = 0; i < 3; i++)
 { int x = x + i; }
```

Improper local declarations within a function can cause even greater problems. The error in the example below is caused by overlapping scopes of the non-local and local variables. While the previous example outputs compilation error, the following one produces wrong result.

EXAMPLE 14.

```
int sum_digits(int n)
 {int sum = 0;
   for (int n;  n > 0;  n/ = 10) //1
     sum + = n%10;
   return sum;
 }
int main()
 {cout≪ sum_digits(654)≪endl;}
```

Line // 1 in Dev C++ and Code Blocked does not cause error for a double declaration. It gives wrong result.

Name of a function parameter and name of a local variable should not match.

Conclusion

We believe that having a decent math background is a key success factor for creating correct and efficient algorithms. In many cases programming means building mathematical models. This paper discusses how math can influence teaching practices, methodologies, and issues that have to be considered in an introductory programming course. We have illustrated those discussions with examples that are ready-to-use or partly included in teaching materials.

By learning components of good programming techniques and style, students build skills that would allow them to feel more confident solving complex problems from the field of the natural sciences and real life.

Teaching that builds those skills, however, is not seamless. It depends on many factors, such as teaching facilities that encourage students to work individually, access to good textbooks, quality and methodological experience of the teaching staff, and last but not least – the programming language used for that course along with the software development environment.

Selection of good examples is a key factor for successful teaching of abstract and theoretical concepts. Here we provide such a selection that facilitates the topics taught in the first twenty hours of a typical introductory course to programming. The tips and comments that accompany the examples help to avoid the first pitfalls in the programming practice. Most of the examples are language-independent (Examples 1, 2, 3, 6, 7, 8, 9, 10, 11, and 12) and can be used with other programming languages. Some of the tips are quite rigorous and cannot be interpreted freely. Students can stick to them and quickly learn the matter. These are in Examples 4, 5, 7, 8, 10, 11, 13, and 14. Other tips, however, require decision on whether or not and how to implement certain rules. These are in Examples 1, 2, 3, 6, 9, and 12.

The teaching practices and topics discussed here are relevant to initial stages of teaching introduction to programming only. Other pitfalls in programming can be considered with more advanced topics, such as functions, classes, structures, recursion, etc., but these are not subject of consideration in this paper.

References

Abramov, S., Zima, E. (1990). *The Beginning of Informatics*. Science, Moscow (in Russian).

Arnold, R., Langheinrich, M., Hartmann, W. (2007). InfoTraffic: teaching important concepts of computer science and math through real-world examples. In: *SIGCSE '07 Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, ACM, New York.

Borovin, G., Komarov, M., Yaroshevski, V. (1987). *The Error Trap When Programming in Fortran*. Science, Moscow (in Russian).

Dijkstra, E.W. (1982). Why is software so expensive? In: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 338–348.

Dijkstra, E.W. (1995). Why American Computing Science Seems Incurable? *EWD1209*, 26 August. http://www.cs.utexas.edu/~EWD/ewd12xx/EWD1209.PDF.

Djordjevic, M. (2007). Teaching introductory programming course with progressive graphics examples. In: *Proceedings of the 2007 Computer Science and IT Education Conference*, Mauritius, 177–185.

Govender, I. (2006). *Learning to Program, Learning to Teach Programming: Pre- and In-service Teachers'Experiences of an Object-oriented Language*. University of South Africa.

Gries, D. (1981). *The Science of Programming*, Springer-Verlag.

Hartmann, W., Naëf, M., Reichert, R. (2006). *Informatikunterricht Planen und Durchführen*, Springer Verlag.

Hromković, J. (2006). Contributing to general education by teaching informatics. In: Mittermeir, R.T. (Ed.), *ISSEP, LNCS*, 4226, 25–37.

Kiesmüller, U.,Brinda, T. (2009). Automatically identifying learners' problem solving strategies in-process solving algorithmic problems. In: *ACM*, ACM-Press, New York.

Kiesmüller, U., Brinda, T. (2008): How do 7th graders solve algorithmic problems? A tool-based analysis. In: *Proceedings of the Conference on Integrating Technology into Computer Science Education*, ITiCSE.

Kinnunen, P., Malmi, L. (2005). Problems in problem-based learning – experiences, analysis and lessons learned on an introductory programming course. *Informatics in Education*, 4(2), 193–214.

Kurland, D.M., Pea, R.D., Clement, C., Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In: Soloway, E., Spohrer, J.C. (Eds.), *Studying the Novice Programmer*, London, Lawrence Erlbaum Associates, 83–112.

Mannila, L. (2010). Invariant based programming in education – an analysis of student difficulties. *Informatics in Education*, 9(1), 115–132.

Mannila, L. (2007). Novices' progress in introductory programming courses. *Informatics in Education*, 6(1), 139–152.

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, 236–240.

Pásztor, A., Pap-Szigeti, R., Lakatos, Török, E. (2010). Effects of using model robots in the education of programming. *Informatics in Education*, 9(1), 133–140.

Papert, S. (1980). *Mindstorms. Children, Computers and Powerful Ideas*. Basic Books, Inc. Publishers, New York.

Radošević, D., Orehovački, T., Lovrenčić, A. (2009). Verificator: educational tool for learning programming. *Informatics in Education*, 8(2), 261–280.

Reichert, R., Nievergelt, J., Hartmann, W. (2001). Programming in schools – why, and how? In: Pellegrini, C. , Jacquesson, A. (Eds.), *Enseigner l'informatique*, Georg Editeur Verlag, 143–152.

Saeli, M., Perrenet, J., Jochems, W., Zwaneveld, B.(2011). Teaching programming in secondary school: a pedagogical content knowledge perspective. *Informatics in Education*, 10(1), 73–88.

Sajaniemi, J., Hu, C. (2006). *Teaching Programming: Going beyond "Objects First"*, University of Joensuu, Department of Computer Science, Technical report, Series A. http://www.cs.joensuu.fi/~saja/publications.html.

Skupiene J. (2006). Programming style – part of grading scheme in informatics olympiads: Lithuanian experience. *ISSEP*, 545–552.

Spolsky J. (2005). *Making Wrong Code Look Wrong*. Joel on Software. http://www.joelonsoftware.com/articles/Wrong.html.

Szlávi, P. Zsakó, L. (2006). Programming versus application. In: Mittermeir, R.T. (Ed.), *ISSEP, LNCS*, 4226, 48–58.

Tam, W.C. (1992). Teaching loop invariants to beginners by examples. In: *Proceedings of the 23rd SIGCSE Symposium*, ACM Press, New York, 92–96.

Teodosiev, T., Teodosieva, G. (2003). Notes training in style of programming. In: *Proceedings of National Conference "Information Research*, Varna, 84–90 (in Bulgarian).

Teodosiev, T., Teodosieva, G. (2005). Once more for training of programming. In: *Proceedings of the 30th International Conference*, Sofia, 118–124 (in Bulgarian).

Van Diepen, N. (2005). Elf redenen waarom programmeren zo moeilijk is (in English: Eleven reasons why programming is so difficult). *Tinfon*, 14, 105–107.

Wirth, N. (2002). Computing science education: the road not taken. In: *ITiCSE Conference*, Aarhus. `http://www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm`.

T. Teodosiev is an assistant professor in the Department of Mathematics and Informatics at Shumen University "Bishop K. Preslavski". His research interests include education in informatics and information technology, programming languages, artificial intelligence.

A. Nachev is a lecturer at Cairnes Business School, National University of Ireland, Galway. Research interests include data mining, pattern recognition, neural networks, SVM, AI, and education technologies.

Keletas sudėtingesnių įvadinio programavimo kurso temų

Teodosi TEODOSIEV, Anatoli NACHEV

Straipsnyje aptariami kai kurie sunkumai, su kuriais susiduriama mokant įvadinių programavimo kursų. Ypatingas dėmesys skiriamas matematinio pobūdžio sunkumams: omenyje turimi tam tikri aspektai, kurie nėra išsamiai paaiškinti vadovėliuose ir dažnai pamirštami kursų planuose bei aprašuose. Keletas pavyzdžių: sudėtingų sąlygos sakinių konstravimas, masyvo ribų viršijimas, begalinių eilučių skaičiavimas naudojantis rekursija ir kt. Autoriai mano, kad šios temos ir atitinkamos pastabos, skirtos mokymo metodikai, gali ir turėtų būti įtraukiamos į įvadinius programavimo kursus.