https://doi.org/10.53656/math2021-6-7-int

INTRODUCTION TO COMPUTER PROGRAMMING THROUGH A SYSTEM OF TASKS

Dr. Lasko M. Laskov, Assoc. Prof.
New Bulgarian University (Bulgaria)

Abstract. Computer programming is a fundamental discipline in many academic programs, especially in the fields of informatics, applied mathematics, physics, and engineering. Despite its popularity, computer programming courses does not possess a widely-accepted methodology for its structure, and because of this reason, even introductory courses highly differ in their curriculum, approach, complexity, and even technical background.

In this paper we propose a methodology for introductory computer programming course structure definition that is based on the concept of notion formation through a system of tasks. The approach is intended to be applied in the context of academic education, but it is also applicable in the last years of high-school courses.

Keywords: computer programming; informatics education; system of tasks; teaching through tasks; notion formation

Introduction

The formation of notions in the early stages of computer programming course could be a quite challenging task. Notions play a fundamental role in the knowledge formation which by itself has its huge impact on the practical skills that are acquired by the students. At the same time, the relation between the knowledge and practical skills of the students definitely is not one-directional: the practical skills of the students have their huge impact on the capability for notion formation.

The process of notions formation is widely studied in the fields of psychology and pedagogy (Usuva 2011; Vygotsky 1987, 1; Vygotsky 1987, 2; Aleksandrov 1999; Rubinstein 1946; Davydov 1996). From the pedagogical point of view, Usuva (Usova 2011) formulates 11 stages that describe this complex process. The reader can find a detailed description of the 11 stages of Usuva in the context of the notion formation in the discipline of data structures in (Laskov 2020).

The 11 stages of Usuva (Usuva 2011) are used as a goal in development of a s*ystem of tasks* that provides the formation of the *set of notions* that are needed to build the knowledge covered in the introductory course of com-

puter programming. For this purpose, during the experiment an approach that is similar to the one described by Assenova and Marinov (Assenova & Marinov 2018; Assenova & Marinov 2019) is followed. Considering specifics of computer programming, the relation between the formation of basic notions and more composite notions is investigated. This relation actually suggests the approach that builds the system of notions starting from more primitive, and developing composite on their basis. A good example here is the introduction of the object-oriented programming paradigm on the basis of the procedural programming paradigm (Laskov 2016). However, due to complicated relation between primitive and composite terms in computer programming, the process of notion introduction is a spiral process, exactly as Usuva suggests. This means that a single notion is introduced multiple times during the curriculum, each time from different point of view, and with different complexity that rely on different preliminary acquired terms. For example, the term "character string" can be introduced as an atomic part needed for the most simple possible program, as an example for a standard library class, as an example of application of mechanism of arrays, and even as an example of application of complex algorithms (Horstmann & Budd 2008) and Cormen at al. 2009).

In his work Vygotsky (Vygotsky 1987, 1) finds the relation between the development of speech and development of thinking. Of course, in this case the subject of research is the process of studying of natural language in the early childhood. Programming languages are an example of formal languages, just like mathematics is, and the process of their studying by adult learners is examined. However, there is a certain parallel between the process of assimilation of natural and formal languages in the fact that learning a programming language actually has an impact on the development of thinking of the students. This impact is clearly observed in the difficulties the students should struggle with in their first course in computer programming.

Something more, Vygotsky (Vygotsky 1987, 2) states that scientific notions are not only a problem of the education itself, but also are a subject of the development of the learner, and the formation of the notion by the process of teaching itself is practically impossible.

The conclusions above are extremely important for the methodology of design of an introductory course in computer programming because of two main features of the subject: (i) the notions are complex and the relation between them is complex as well; (ii) the subject is highly related with the practice, and practical skills of the learner directly affect his capability to form notions. In this way the principles deducted by Vygotsky motivate the structure of the curriculum, proposed here, that adhere to 11 stages of Usova, and which is based on a system of tasks.

Programming environment

The technical background in which the course is conducted is often overlooked, or even worse, assumed to be fixed by default. In order to develop the required notions by a system of tasks, the course should start from the very basis, namely from the programming environment, and the tools to be used during the course should be selected in such manner, that non of the stages of computer program creation will remain hidden or "behind the scenes" from the learner.

For that reasons, it is recommended to avoid using an Integrated Development Environment (IDE) during the process of introduction to programming. Besides the comfort it brings, an IDE actually hides the composite process of program compilation from the student. A student who has passed the whole curriculum using an IDE often does not make the difference between a text editor, compiler and debugger, which can seriously impede the development of notions of *computer program*, and *compilation process*. With these arguments, in the presented example the compiler is used directly from a text-base terminal of the operating system (OS) to build the programs, and a simple text editor to enter the texts of the program.

The selection of the compiler, of course, depends on the programming language that is used, in presented case this is C++ (for motivation of language choice see other Laskov's work (Laskov 2016)). We do not use any closed source code software, and the reason for this decision is to widen the technical vocabulary and increase the practical skills of the students by introducing contemporary tools that are widely used in real-life practice. Hence, a good choice of compiler is the Gnu Compiler Collection (GCC) (Griffith 2002) front-end for C++. Following the same logic, the selection of the OS is a Unix/Linux-based distribution, in particular case this is Ubuntu Linux (Helmke et al. 2016). For many of the participating students this is a new software environment, and during the computer programming course their computer literacy is naturally increased by the knowledge how to use Linux, especially from Bash terminal (Ramey & Fox 2015) which is used to run the compiler and all programs that are implemented.

The notions *Operating System* and *terminal* are developed by introducing some of the basic bash commands that are needed to manipulate files, and navigate through the file system.

Task 1. Using the appropriate Bash commands, create a directory in an appropriate location which will be your working directory for this class. Always give meaningful names to your files and directories.

The complete and deep understanding of OS is a subject of the respective courses of computer architectures, operating systems, and Unix-based systems that are taught in the programs.

The development of the notions *compiler* and *program* starts by a set of simple tasks.

Task 2. Check whether the GNU Compiler Collection (GCC) is installed on your system by simply typing:

$ gcc --version

If GCC is not present, install it by typing the following command:

$ sudo apt-get install build-essential

or alternatively by typing:

$ sudo apt install build-essential

Task 2 starts the development of the notion *compiler* by showing clearly that it is a differentiated *program* that is needed in order to be able to build programs with the given programming language. The next step is to demonstrate the purpose of the compiler as a computer program that is used to translate code written in a given programming language to another language, that is interpreted by the system (Griffith 2002). For that purpose we need the very first example that is usually taught – the famous *"Hello, World!"* program.

Understanding compilation process

Compilation process is essential in building notions *compiler* and *program*. The learner must enter the text of the program (Listing 1) initially even without understanding the meaning of its components, to start the compiler from the terminal, and to execute the binary code.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, World!" << endl;

    return 0;
}
```

Listing 1: "Hello, World!" program in C++ programming language

Task 3. Open the **gedit** text editor (you are free to use another, if you prefer). Type the text of the C++ program given in Listing 1 and save it as hello.cpp in your working directory.

Task 4. Compile the program in Listing 1. The command g++ is the front end for C++ of the GCC compiler collection:

$ g++ hello.cpp

It will produce the default binary a.out which can be executed by typing

$ ./a.out

The result of the execution of the program will be the appearing of the message Hello, World! in the terminal. If you would like to specify the name of the binary code, produced by the compiler you have to pass the name to the compiler as an option:

$ g++ -o hello hello.cpp

Then you can run the program by typing:

$ ./hello

Tasks 3 and 4 already clearly show that text editor and compiler are two different programs, which are used for different purposes (Figure 1). Also, the learner observes that the program is written in one form of a computer language, namely the *source programming language*, and it is transformed to another form, the binary file, that can be interpreted by the system.

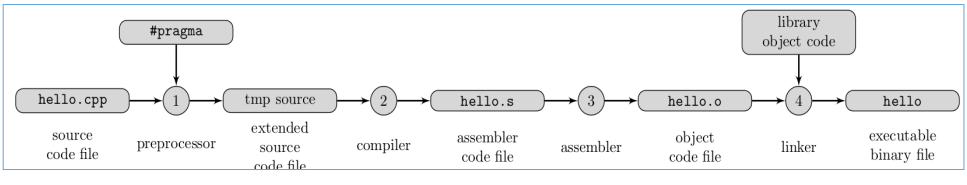

Figure 1. Compilation process of a C++ program

Compilation of a C++ program is a complex process (see Figure 1) that is composed by four distinct steps. Its understanding is essential for building of the notions *program*, *source code*, *library*, and, of course, *compiler* itself. In the proposed approach this process does not remain hidden from the students, but actually it is presented in relative detail using the following Task 5 and Task 6.

Task 5. Examine the compilation process by stopping it after each stage:

– preprocessing: g++ -E hello.cpp
– compile: g++ -Wall -ansi -S hello.cpp
– assembling: g++ -Wall -ansi -c hello.cpp

```
#include <iostream>

using namespace std;

#define PROGRAM int main()
#define BEGIN {
#define END }
#define PRINT(x) cout << x << endl;
#define STOP return 0;

PROGRAM
BEGIN
    PRINT("Hello, World!")
    STOP
END
```

Listing 2: "Hello, World!" program code modified using pargmas

Task 5 (Listing 2) demonstrates the intermediate steps of the compilation process to the novice programmer. If the standard approach of teaching through some of the standard IDEs was used, the latter would to be hard to demonstrate. The understanding of the individual stages of the compilation process has a huge impact on clear comprehension of the elements even of the simplest possible program. The *pragmas* (preprocessor directives) are a good example of program elements, which are hard to explain without any knowledge of compilation process.

Task 6. Implement the program from Listing 2. The pragmas used in the code allow to implement the *"Hello, World!"* program in a way that remind the syntax of another programming language.

Even though the example demonstrated in Task 6 may look redundant, or too complex for a novice learner, it shows both power, and drawback in using the important mechanism of the prepocessor, that was already introduced as a notion in the previous Task 5: (i) the power is in the flexibility of the extended source code; (ii) the drawback is in the fact that pargmas may affect source code readability.

By combining the examples from the latter two tasks, it is demonstrated both the purpose/usage of pragmas and the extended source code. These notions are significant also in more advanced stages of the course – just imagine the hardness of introduction of the term *include guards* in the context of separate compilation to a learner, who does not already clearly understand prepocessor and pragmas.

Programming paradigm

Object-oriented programming (OOP) is one of the most popular programming paradigms now-a-days (Roy & Haridi 2004). Very often it is adopted as the first imperative programming paradigm used in courses, mainly because of the popularity of Java programming language and platform (Horstmann 2019), and other commercial platforms, like .NET for example. OOP is widely adopted in other languages, that are used for interesting applications like computer games, data science and machine learning like Python, for example (VanderPlas 2016).

When introduced in beginners courses of computer programming, OOP is very often directly taught, starting from the basic terms *objects* and *classes* (Horstmann 2019), (Horstmann & Budd 2008). This approach could be effective in the case when learners already are familiar with computer programming in some introductory high-school level, however we assume that our students have no experience in computer programming at all. Also, considering one of the major notions, that must be acquired during the course, *abstract data type (ADT)* (Laskov 2020): there is a **major difference** with the concept of "objects" that are the basic term in OOP. While ADT clearly separates values and operations applied on them, objects actually combine them together (see Figure 2, and refer to Roy and Hardi (Roy & Haridi 2004) for more details).

From this point of view, is considered much more appropriate to introduce OOP by initially teaching the procedural programming paradigm (Laskov 2016). The advantage of this approach is that the basic notions *variable* and *function* are clearly introduced to learners, and they will have the needed set of programming instruments to perceive the complex notion *object* latter in the course. From the point of view of the programming language that is used in the course, this transaction is fluent, because C++ is a multi-paradigm language.

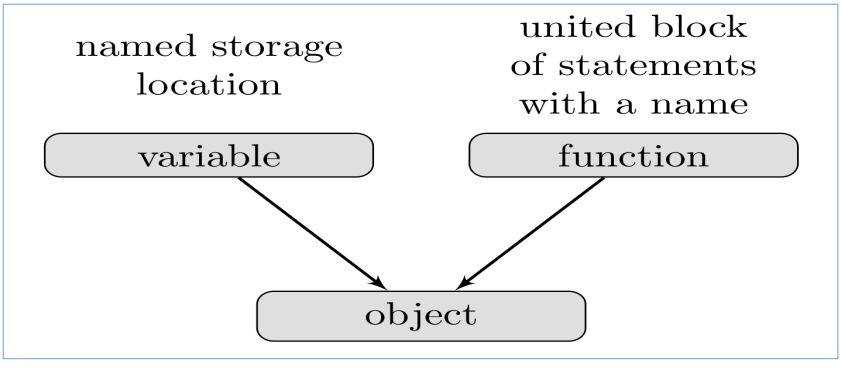

Figure 2. The notion object as composed by the notions variable and function

Formation of the notion "variable"

The purpose of the following Task 7 is to introduce the learner to the notion *variable*. The goal is to describe variable as a storage location in computer memory that has a name (identifier), can store values of a particular data type (note that type system of C++ is nominal (Stroustrup 2013)).

```
#include <iostream>

using namespace std;

int main()
{
    cout << "First exam of practice: ";
    int ex1;
    cin >> ex1;
    cout << "Second exam of practice: ";
    int ex2;
    cin >> ex2;

    ...                 // continue in similar manner

    cout << "Grade: " << ex1 + ex2 + ex3 + hw1 + hw2 +    hw3 <<
endl;

    return 0;
}
```

Listing 3: Calculate the grade for the course in points

Task 8. Write a program that calculates the final grade in points of a student for the course. Read the points for the three exams of practice and three homework assignments, sum them and display the result. Use the code in Listing 3. How many variables do you need using this approach?

Task 9. Rewrite the program from previous exercise using only two integer variables. First create an integer variable that will store the sum of all points, and a variable for the current score. Do not forget to initialize them:

```
int sum = 0;
int curr = 0;
```

On each step of the algorithm, read the current score, and add it to the variable sum.

```
cout << "First exam of practice: ";
cin >> curr;
sum += curr;    // the same as sum = sum + current
```

Finally, print the accumulated result, stored in sum.

Task 9 provokes learner to discover the redundancy in the variable usage in the simple example in Listing 3, and puts an accent on one of the main feature of the variables – the value stored by can change during the execution of the program.

Formation of the "function"

The purpose of the following Task 10 is to introduce the learner to the notion *function*, in the beginning as a black box: the function takes as an argument the given data, performs intrinsic calculations that remain hidden, and returns the result to the caller (see Figure 3).

Task 10. Using #include <cmath> before using namespace std; write a program that calculates the square root of a floating-point variable.

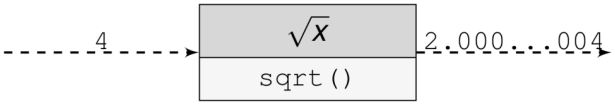

Figure 3. Function as a black box

After this initial development of the notion *function*, it can be augmented by demonstrating to the student how functions are actually implemented. At this point the term *function* is used as a synonymous to *global function* (a function that is not part of a class).

```
double pointDist(double x1, double y1, double x2,
                                       double y2)
{
    double square_x = (x1 - x2) * (x1 - x2);
    double square_y = (y1 - y2) * (y1 - y2);
    double distance = sqrt(square_x + square_y);
    return distance;
}
```

Listing 4: Function that calculates distance between two points

Task 11. Write a program that calculates the distance between two points using the function in Listing 4.

In similar manner, all connected notions are introduced, including function signature, function prototype, passing parameters by value, passing parameters by reference, parameter default value, functions that return no value (void functions), inline functions.

Task 12. Write a program that reads an integer as a string, and then transforms the string to an integer. Use function in Listing 5 that implements Horner's method:

$$P_n(x)=(...(a_nx+a_{n-1})x+...+a_1)x+a_0$$

```
int strToInt(const string& str, int rad = 10)
{
    int result = 0;
    for (int i = 0; i < str.length(); i++)
    {
        result *= rad;
        result += str[i] - '0';
    }
    return result;
}
```

Listing 5: Transforms a string of digits to integer

Task 12 builds the notion default parameter value, and also demonstrates the implementation of the Horner's method, that is a relatively simple but important algorithm to calculate the value of a polynomial, used in applications like string to integer transformation, transformation between numeral systems of different bases, and many others. Task 13 further demonstrates *reference parameters*.

Task 13. Implement a function that swaps the values of the variables passed as its arguments at the function call. To swap the values of two variables, you will need a third temporary variable (see Figure 4, Listing 6.).

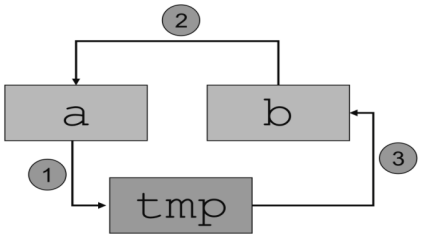

Figure 4. Swap values of two variables

```
void swap(int& a, int& b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Listing 6: Swap values of the arguments

```
inline void sort2(int &a, int &b)
{
    if (a > b)
    {
        swap(a, b);
    }
}
```

Listing 8: Sort two integers

```
inline void sort3(int &a,
                  int &b, int &c)
{
    sort2(a, b);
    sort2(b, c);
    sort2(a, b);
}
```

Listing 7: Sort three integers

Task 14. Implement a function that sorts two integers (Listing 7). Use the function swap(). Implement a function that sorts three integers, using the implementation of the algorithm for sorting two integers (Listing 8).

Task 14 further helps the development of the notion *function* by showing how function calls form the functions call stack. The task demonstrates how a complex problem may be broken down on simple, easy to implement sub-problems naturally separated in functions whose interconnections form the final program. This a demonstration of the procedural programming paradigm, in which the main building block of a program is function.

Fotmation of the notion object

Having introduced student to the procedural programming paradigm, it is much more natural and fluent to develop the notion *object*, and to explain the composite mechanism of classes. The goal of this section of the curriculum is to introduce the learners to the four principles of the OOP:

1. Data abstraction.
2. Encapsulation.
3. Inheritance.
4. Polymorphism.

In the introductory part of the curriculum, only the first two principles are developed. Inheritance and polymophism are left for the second part of the course, in which OOP principles are deeply explained, together with more complex programming concepts like pointers and dynamic memory management, streams, recursion, operator overloading, generic programming, exceptions, and introduction to some algorithms and data structures. We start with Task 15 that provokes the learner to discover the need of a mechanism that may unite the concepts of function and variable, and a language mechanism to create new data types in the program (see Figure 5). In this manner Task 15 naturally leads to the principle of *data abstraction*.

Task 14. Write a program that reads the data for a student and prints it in the standard output. The data is composed by the student's first name, surname and faculty number (see Listing 9). What if you need to process the data for another student?

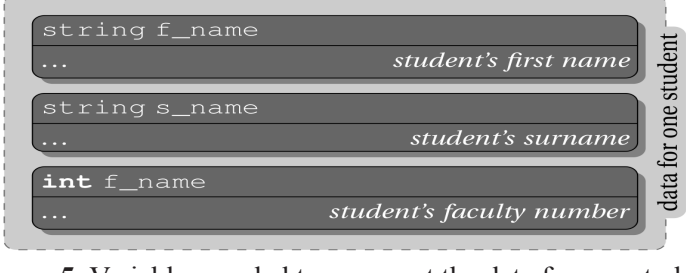

Figure 5. Variables needed to represent the data for one student

```
cout << "First name: ";
string f_name;                          // first name
cin >> f_name;
cout << "Surname: ";
string s_name;                          // second name
cin >> s_name;
cout << "Faculty number: ";
int f_number;                           // faculty number
cin >> f_number;
cout << f_name << " " << s_name << " F" << f_number << endl;
```

Listing 9: Process the data for a student

Task 15. Instead of duplicating the same code a new data type can be defined that will represent the concept. A class is a user-defined type that combines both data and functionalities. Write a program that has a main() function and the definition of the class Student before it. What is the compiler response if you try to crate an object of type Student?

Task 15 introduces the notion *class* and continues with the next step of the development of the complex notion *object*. The class is represented as a blueprint to create objects – in this way the mechanism to create objects in the programming language helps to sketch out the notion. Also, an introduction to the notions *constructor* and *member function* is given.

```
class Student
{
public:
    Student();                          // default constructor
    void readData();                    // mutator: modify data
    void printData();                   // accessor: access data
private:
    string f_name;                  // data fields
    string s_name;
    int f_number;
};
```

Listing 10: Class Student definition

In the tasks that are related with Task 15 all the elements of the class in Listing 10 are implemented. Together with the implementation of the member functions, two helper notions are introduced: (i) accessor – a member function that does not change the intrinsic state of the object; (ii) mutator – a member function that modifies the state of the

object. Also the usage of public and private sections of the class are clearly explained, which gives introduction to the second principle of OOP, namely *encapsulation*.

The next step in notion object development is to provide a new abstract concept to the learner, and to incite him to implement the class that realizes the *data abstraction*. The student must decide which parts of the class must be included in the public, and which parts must be placed in the private sections, which consolidates the notion *encapsulation*.

Task 16. Design the definition of a class Point . A point is composed by its x- and y-coordinate, which are floating-point numbers (Figure 6).

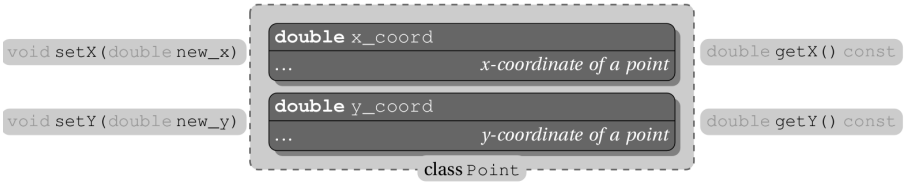

Figure 6. Class Point encapsulation

The final development of the notion *class* actually completes the introduction of the notion *function* by the explanation of *member function.* Before that, the notion *function* was actually used to build the notion *object*. The interrelation between these notions shows the complication of the system of terms that is perceived by the students in the introductory course of computer programming, and the spiral process described by (Usova 2011) of notion formation (see Figure 7).

Conclusion

In this paper we present an approach for development of introductory course of computer programming. Complex notions formation is provided through system of tasks, and the knowledge is developed on the basis of the practical approach, which from pedagogical point of view is highly motivated by the works of Vygotsky (Vygotsky 1987, 1), (Vygotsky 1987, 2) and (Usova 2011).

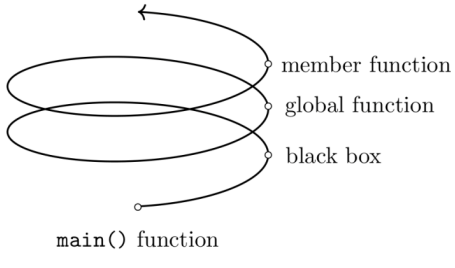

Figure 7. Formation of the notion function

As technical environment of the course they are adopted the basic tools of simple text editor, compiler (in our case GCC), debugger (GNU Debugger), and the selected OS is a Linux distribution. This approach allows the introduction of the notions starting from the most basic ones, and developing them to the more complex. The individual steps of program compilation are not hidden from the learners, as they would remain in the case an IDE was used, as usually it is done in most of the courses. Thanks to this method, the students clearly understand the purpose of the compiler, the individual stages of compilation process, and thus – individual parts of the C++ program that highly depend on those stages (a good example are preprocessor directives). Also, the technical vocabulary of the learners is increased with contemporary tools that are often used in real practice, and Linux distributions, which are widely used in the curriculum of the university program. The motivation behind this approach relies on the comparison between the process of natural language development described in (Vygotsky 1987, 1) and learning of formal languages such as mathematics and computer programming languages, according to which the formation of the ability to use the language is strictly connected with the development of the thinking of the learner.

Introduction of the concepts of OOP, starting from the procedural programming paradigm relies on the same pedagogical arguments. According to Vygotsky (Vygotsky 1987, 2) the formation of a scientific notion is deeply related with development of the thinking of the learner, and the presented approach allows this development to occur gradually, without huge gaps of incompleteness of the terms that are introduced through the system of tasks. However, it is important to note that the relation between notions in computer programming is complex and because of that, it is impossible to form them in a linear manner.

A good example is the notion *function* (see Figure 7). Initially it is introduced through the main() function of a C++ program. On the second stage, it is shown as a black box of the library functions used in the examples. The third stage is the demonstration how a global function is implemented. The final stage is given by the tasks that implement class member functions. At the stage of class implementation, the notion *function* is already adopted in formation of the notion *object*.

Such relations, as decried above, are inevitable in complex system of notions, as those used in computer programming. On the other hand the bottom up approach, decried in this paper aids the gradual notion formation.

Finally, we would like to point out that the tasks, presented here, are examples that can be used to form a whole family of similar systems, whose complexity may be adjusted according to the level of the learners. In this way the approach can be adapted for the needs of the high-school, or even for the needs of more advanced courses.

REFERENCES

Abelson, H., Sussman, G. J., Sussman, J., 1996. *Structure and Interpretation of Computer Programs* (2nd ed.), MIT Electrical Endeepgineering and Computer. Science, ISBN-13: 978-0262510875, ISBN-10: 0262510871.

Aleksandrov, A. D., Kolmogorov, A. N., Lavrent'ev, M. A., 1999. *Mathematics: Its Content, Methods and Meaning*, Dover Books on Mathematics Series, Courier Corporation, ISBN 0486409163.

Asenova, P., Marinov, M., 2019. System of tasks in mathematics education. *Mathematics and Informatics*, **62** (1), 52 – 70.

Asenova, P., Marinov, M., 2018. Teaching mathematics with computer systems, *Math. and Education in Math*, **47**, 213 – 121.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press, ISBN-13: 978-0262033848, ISBN-10: 9780262033848.

Davydov, V. V., 1996. *The Theory of Developmental Education*, INTOR Moskow. [In Russian].

Griffith, A., 2002. *GCC: The Complete Reference* (1st. ed.). McGraw-Hill, Inc., USA.

Helmke M., Joseph, E. K., Antonio Rey, J., 2016. *The Official Ubuntu Book* (9th. ed.). Prentice Hall Press, USA.

Laskov, L. M., 2016. *Progrmming in C++, Examples and Solutions, Part One: From Procedural Towards Object-Oriented Paradigm,* New Bulgarian University, ISBN 978-954-535-903-3.

Laskov, L. M., 2020. Abstract data types, *Mathematics and Informatics*, **63**(6), 608 – 621.

Horsmann, C., Budd, T. A., 2008. *Big C++* (2nd ed.), Wiley, ISBN: 978-0-470-38328-5.

Horstmann C., 2019. *Big Java: Early Objects*, 7th Edition, ISBN: 978-1-119-49909-1.

Ramey, Ch. and B. Fox, 2015. *Bash 4.3 Reference Manual*. Samurai Media Limited, London, GBR.

Roy, P. V., Haridi, S., 2004. *Concepts, Techniques, and Models of Computer Programming* (1st. ed.). The MIT Press.

Rubinstein, S.L., 1946. *Fundamentals of general psychology*, State Study-Pedagogical Publishing House of the Ministry of Education of Russian Soviet Federative Socialist Republic, Moscow, USSR, 704. [In Russian].

Stroustrup, B., 2013. *The C++ Programming Language* (4th. ed.). Addison-Wesley Professional.

VanderPlas, J., 2016. *Python Data Science Handbook: Essential Tools for Working with Data* (1st. ed.). O'Reilly Media, Inc.

Vygotsky, L. S., 1987, 1. Thinking and speech. In *The collected works of L.S. Vygotsky. Problems of general psychology* **1**, (37 – 285) (translated by Norris Minick). New York, London: Plenum Press.

Vygotsky, L. S., 1987, 2. Collected works, Problems of psychics development, *Pedagogy*, **3**, Moscow. [In Russion]. Original title: Выготский, Л.С., 1983. Собрание сочинений. Проблемы развития психики. *„Педагогика“*, **3**, Москва.

Usova, A. V., 2011. Some methodological aspects of the problem of scientific notions formation in learners in schools and students in universities, *Mir nauki, kulturi, obrazovania*, **4**(29). [in Russian]. Original title: Усова, А.В., 2011. Некоторые методические аспекты проблемы формирования научных понятий у учащихся школ и студентов вузов. *Мир науки, культуры, образования*, **4** (29).

✉ **Dr. Lasko M. Laskov, Assoc. Prof.**
ORCID iD: 0000-0003-1833-8184
Web of Science Researcher ID: K-7516-2012
Department of Informatics
New Bulgarian University
21, Montevideo Blvd.
1618 Sofia, Bulgaria
E-mail: llaskov@nbu.bg