https://doi.org/10.53656/math2022-5-1-err

ERROR MANAGEMENT TRAINING IN COMPUTER PROGRAMMING COURSES THROUGH A SYSTEM OF TASKS

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

Abstract. Errors are an indivisible part of computer programming, and as such their incorporation as a tool in teaching is a natural approach to stimulate learners to be an active side in the educational process. The application of errors as an instrument for illustration of knowledge, and the encouragement of students to learn from them, is the main approach of Error Management Training (EMT). EMT has been shown in number of psychological studies (Frese 1995; Keith & Frese 2008; Dyre et al. 2017) as an efficient teaching technique, even compared to the traditional error-avoidance methods. In this paper we present an application of EMT in computer programming courses, based on different approaches for error handling, which must be an important part of the curriculum.

Keywords: computer programming; informatics education; programming errors; error management training; teaching through tasks

1. Introduction

In the introduction of the chapter Errors of his book (Stroustrup 2014), Bjarne Stroustrup emphasizes: *What we are trying to do is to show what "thinking like a programmer" is about.* Indeed, errors are such an indivisible part of the process of computer programming that the attitude to them and the techniques for error handling largely determine the cognitive process involved in software creation. Even a novice programmer inescapably understands that it is nearly impossible to implement yet a simple program without any errors. In the case of real-life complex applications this rule is even carried to extreme, and different approaches to error handling must be combined together to ensure that the final version of the source code will be relatively safe and suits the tasks which it is intended for.

During the implementation and testing of a computer program errors cannot be avoided. Instead, during the different stages of the process, they have to be discovered, solved, and even predicted – and still a well written program code will possess a high level of confidence for robustness, but never full faultless. Exceptional situations and their handling are turned into a tool for verification of software correctness, and as such the correct approach to them helps the creation of the source code. Thus, it is also natural to incorporate errors as an important teaching tool in computer programming courses.

Of course, error handling mechanisms are a classical part of the curriculum of computer programming courses (Stroustrup 2014; Horstmann & Budd 2008; Horstmann 2019; Laskov 2016) and are usually taught using the traditional error-avoidance methods. In other words, exceptional situations are explained mainly from the precondition goal to escape from them, which is a natural and straightforward approach. However, an innovative teaching paradigm suggests to bring the errors to the foreground of the learning process, and to encourage learners to experiment with them, and learn from them: the Error Management Training (EMT).

EMT has been compared to the traditional error-avoidant methods and it has been shown as a superior teaching technique in number of psychological studies (Frese 1995; Keith & Frese 2008; Dyre et al. 2017). It has been shown that EMT aims to overcome the emotional negatives of making mistakes (Heimbeck et al. 2003), which is a common obstacle for the students in a beginner's course of computer programming.

In our practice we adopt the approach of teaching computer programming by formation of notions through a system of tasks (Asenova &Marinov 2018, 2019; Laskov 2021). In this context, the transfer of knowledge may be improved by encouraging learners to make errors intentionally (Dyre et al. 2017) – a technique that is natural in the case of computer programming training, since exceptional situations themselves can be used to depict clearly different practical aspects in the tasks.

In this paper we put errors and error handling mechanisms in the center of creation of a system of tasks for computer programming courses in three different levels of complexity: basic, advanced and intermediate. These levels are determined by observing the relative effort of the learners to develop the notions needed to describe the causes of the errors. These levels of complexity are used to describe system of tasks used in the following computer programming courses:

1. Introduction to computer programming.
2. Object-oriented programming.
3. Data structures.

We provide a short discussion of different types of errors in computer programming. The classification scheme of programming errors is applied in the EMT approach to the system of tasks that form the three levels of courses complexity.

We will discuss the alternative methods for error handling and we show how we build a system of tasks that use errors as the main teaching instrument.

2. Error classification

2.1. Evolution of classification systems

Software bug is one of the most common slang words indicating error in a computer program that was used so extensively that it even became an official term.

The most popular version about the first usage of the word “bug” in the context of computing was told by Grace Hopper and it was that a moth was found in one of the relays of the Harvard Mark II computer in 1945 – see pp.205 – 206 of (Horstmann and Budd 2008). However this term was used before that and the first documented occurrence of “bug” in relation to computing was in a Bob Campbell’s entry in the logbook of ASCII/Mark I computer in April 1944 (Cohen 1994). Despite its popularity, there is no widely established error classification scheme applicable both in software development and education in computer programming. There are many extensive research works on software reliability analysis, historically starting from the late the seventies of the last century.

An early research in the technical report (Motley & Brooks 1977) suggests linear regression analysis application to predict statistically programming errors. The subjects of analysis are linear combinations of program characteristics (used to measure the complexity of the program) and programme variables. Program exceptional situations are defined as errors that are found during the formal testing, can be attributed to the programmer, and require changes in the source code of the program. The errors are classified in eight categories: logic, data handling, interface, data input/output, computational, other, data bases, data definition. Some errors are not classified as programming errors, but still they are result of the development process: requirements specification, design, coding, maintenance (correction of errors).

Another early work (Bowen 1980) compares the state of the art for that time error classification schemes and proposes error categories and subcategories. The classification scheme is based on comparison of the application to multiple projects and attempts to avoid excess granularity. Three categories to provide feedback are defined: source (the phase where error occurs), cause (casual description), and severity (effect of the error). Seven major categories are proposed to support software reliability analysis: design, interface, data definition, logic, data handling, computational, and other.

A notable work is proposed by Donald Knuth (Knuth 1989) in which he provides an extensive analysis of the errors during the evolution of his famous TEX project during a period of ten years. He presents error classification into fifteen categories: A (algorithm), B (blunder), C (cleanup), D(data), E (efficiency), F (forgotten), G (generalization), I (interaction), L (language), M (mismatch), P(portability), Q (quality), R (robustness), S (surprise), T (typo). Further, Knuth classifies the categories in two classes:

- “bugs” – programming errors that obligatory have to be corrected (A, B, D, F, L, M, R, S, T);
- enhancements – situations that can be considered as conditions for enhancement, rather than errors.

Also, the categories are classified with respect to their difficulty:

- simple (T, B, F, L, M);
- technically more complex (A, D);
- special situations from which the program must recover even in wrong input (R);
- complex interactions between different parts of the program (S);
- various types of enhancements during the development of the system (C, E, G, I, P, Q).

Knuth points out that the presented classification scheme is *ad hoc*, and it relies on the functionality of the program, rather than attempting to classify the particular programming errors, like for example misuse of a given operator from the programming language. Thus, this classification scheme is not focused on teaching computer programming; however it clearly shows a scheme for analysis of evolution of a complex project by tracking down the categories of errors. Also, the errors that are cause of wrong initial ideas in the algorithm are clearly separated in the category A. Besides the above fundamental works, a contemporary approach to software errors classification scheme is based on software engineering approach. In the (Ko & Myers 2003) and (Ko & Myers 2005) the authors propose a model of programming errors that is based on an extensive survey of the existing error classification schemes and error causes analysis. Most of the errors are reported to be a result of problems in implementing algorithms, language constructs and uses of libraries.

2.2. Application in education

The above error classification schemes are either result of analysis of complex software projects, or are directly designed to serve the software development process. Even though they are closely related, these classification schemes are not created to aid the educational process. Something more, often when teaching computer programming, the error classification systems are simplified.

Often the focus is on logic (semantic) errors, which is a natural approach because for novice programmers these types of errors are most difficult to resolve. For example, in (Ettles et al. 2018) logic errors are classified in three categories: algorithmic errors, misinterpretation of the task, fundamental misconception. The misconception is concluded as the most frequent and difficult for students.

It is a common approach in the educational literature to list the general categories of errors that are used in the consecutive examples. For example, in (Horstmann and Budd 2008), p.666 the author lists the following types of errors that are used to illustrate the usage of the mechanism of exceptions: user input errors, device errors, physical limitations, component failures. On the other hand, Stroustrup uses the following classification in (Stroustrup 2014): compile time errors (syntax errors, type errors), link-time errors, run-time errors, errors detected by the computer (hardware and OS), errors detected by a library, errors detected by user code, and logic errors. The latter approach is close to the real-life analysis of complex projects, but still is well-suited for the educational purposes, because it is not too granulated (as it was mentioned in (Bowen 1980)).

We must also point out that the error classification schemes may vary depending on the programming language being used. For example, if we consider the classification scheme in (Stroustrup 2014), the errors in the linking step cannot be directly applied in the case of Java programming language (Horstmann 2019; Eckel 2011). And vice versa, some mechanisms in Java (like checked and unchecked exceptions) are not directly applicable in C++ programming language, and for that reason the errors that are classified to be handled by these mechanisms cannot be directly transferred.

2.3. Proposed classification scheme

The complex and too detailed error classification schemes like those in (Bowen 1980; Knuth 1989; Ko & Myers 2003, 2005) are not suitable for our purposes in the courses in computer programming that are designed for absolute beginners. On the other hand, since the examples used in our curriculum use errors as an instrument for knowledge formation, we cannot adopt the approach towards errors which otherwise is successful in the standard error-avoidance educational methods. See for example (Horstmann & Budd 2008) and (Laskov 2016).

The above motivates us to propose our error classification scheme (Fig. 1) that is applicable in the case of EMT in computer programming. Our programming courses are designed using the C++ programming language, and for that reason naturally the technical aspects of our scheme are highly influenced by those proposed in (Stroustrup, 2014). We will point out that the selection of the programming language will not limit the described approach. For example, with few technical modifications, it is absolutely applicable in courses built based on Java programming language (Horstmann 2019; Eckel 2011).

The error classification scheme (Fig. 1) is hierarchically organized and has a tree structure. The root of the tree is the general category *error*. The *error* category is divided in two separate classes:

1. *Internal*: represents all errors that are direct result of constructions that comprise the computer program.

2. *External*: represents all errors that result from circumstances that are not part of the program itself.

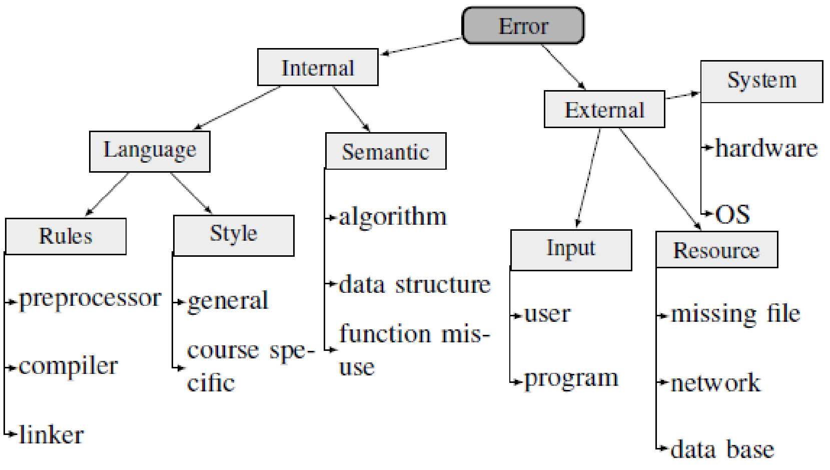

Figure 1. The tree structure of the proposed error classification scheme.

The *internal* category consists of:

• *Language*: these are all errors that violate the formal rules of the programming language and the accepted style. They can be one of the following two sub-categories.

– *Rules*: the errors that violate the rules of the language. In the case of C++ an error can be caused in the preprocessing, in the compilation or during the linking step. In our courses we pay an extra attention to compiler warnings as well.

– *Style*: many programming environments and languages have their widely accepted style rules which are not part of the programming language itself. They include code formatting, program elements naming, code organization in source code files. We distinguish between widely accepted style rules, and some rules that are imposed specifically for the purposes of our courses. For example, in our courses we pay special attention to sourcecode formatting, and program elements naming rules. Also, we insist that the `main()` function of a C++ program always ends with return 0; statement, even generally it could be omitted (Stroustrup 2014).

• *Semantic*: these are logical errors, and are the category of errors that are hardest to find and correct. Logical errors can be in the very idea of the program or the algorithm itself. They can be caused by the selection of wrong data structures, or in data structures implementation itself. Also, we will consider a semantic error, when a function or routine of the program is not used correctly (for example calling the `sqrt()` function with a negative argument).

The *external* category consists of:

• *Input*: these are all types of errors that result from wrong input to the program from both user and other programs. These types of errors are not the same as the wrong function input that are classified in the *semantic* category of *internal* errors, mostly because the input to the program is often not controlled by the programmer. However, a programmer may try to predict the wrong program input, and make the program code robust against these types of external circumstances.

• *Resource*: these are errors that arise because of a missing resource, such as missing file, failed network connection, or failed database connection. Different frameworks have tendency to handle differently these types of errors, for example in Java the mechanism of checked exception deals with missing resources.

• *System*: the programmer has to consider that the system itself, on which the software is running, may possibly fail. For example, a hardware device may go out of order, or the OS may malfunction.

Besides the above described error categories in the hierarchical structure, we will distinguish two categories that are most often emphasized during programming courses:

1. *Compile-time errors*: these are errors and warnings that are generated by the compiler during the compilation process of the program. Apparently, the errors from the *rules* category fall inside this category. Note that in particular cases, errors from other categories, may also lead to *compile-time errors*.

2. *Run-time errors*: these are errors that are encountered during the execution of the program. They do not violate the rules of the programming language and are not caught by the compiler. For example, the errors from the *input* category are *run-time errors*.

These two error categories are described outside the hierarchy structure on Fig. 1 because they introduce another classification scheme with the respect of the compilation process and execution of the program. Also, a category may be classified as a *compile-time error* or a *run-time error* depending on the programming language being used. For example, a missing file may cause a *run-time error* in a C++ program, but it will result in a *compile-time error* in Java.

Something more, many programming languages have mechanisms that attempt to reduce the *runtime errors* to *compile-time errors*. Such mechanisms are for example the checked exceptions in Java (Horstmann 2019; Eckel 2011), or constant function parameters and constant member functions in C++ (Stroustrup 2014; Horstmann & Budd 2008; Laskov 2016). In the design of the system of task in our courses, we pay extra attention to these language constructions.

3. Error handling mechanisms

Error handling mechanisms have been evolving together with the programming paradigms and programming languages, and as a result different alternative approaches exist (Koenig & Stroustrup1990; Horstmann &Budd 2008). For the needs of a computer programming course, we must consider the programming paradigm and language, but also the appropriate complexity of the selected approach.

The methods that are used in our programming courses are mainly influenced from the selection of the C++ programming language, but also consider the legacy of the C programming language. For that reason we combine features presented in (Stroustrup 2014; Koenig & Stroustrup 1990; Horstmann& Budd 2008) to achieve the following system of alternative error handling mechanisms.

3.1. Assume that no error will occur

Definitely the easiest approach, and of course the less applicable in the development of real-life applications, is to assume that no error will occur at all, and not to try to handle any exceptional situations. On the other hand, this attitude towards errors is practical in the case of the *basic level* of complexity of examples. From the point of view of EMT this approach is actually appropriate, because the errors that we would like to emphasize can stand out.

Task 1. *A function circleRad()takes as a parameter the area A of a circle and calculates its radius* `r. Recall that r =` $\sqrt{\frac{A}{\pi}}$ *where* `p= 3.141592654` *must be a global constant. Trigger the function with a negative argument.*

```
double circleRad(double area)
{   return sqrt(area / PI);}
```

Listing 1. Calculate circle radius, given its area. Function assumes no error will occur.

The implementation of the function that is given in Listing 1 assumes that no error will occur, and if the argument is a negative number, the `sqrt()` function will return `-nan`. Note that the minus sign in front of the *not-a-number* constant is a result of its intrinsic representation: all the bits of the exponent of the floating-point number are set to 1, while the mantissa remains as it is, so is the bit that represents the sign of the number. In the proposed classification of errors, this is an error of type *semantic/ function misuse*.

3.2. Output error message

Another basic technique to treat error situations is simply to output an error message either to the standard output, or to a log file for more complex examples. This approach is also not very practical, however in basic examples it gives a good representation where and why an error can happen.

```
double circleRad(double area)
{   double result = 0.0;
   if (area < 0)
   {cerr << "Negative area." << endl;}
   else{   result = sqrt(area / PI);}
   return result;
}
```

Listing 2. The circle radius task solved with error message in case of negative input.

Note that using the solution in Listing 2 the error is detected and it is displayed in the standard error output stream, however the program flow is not interrupted, and the function will even return a value in case of exceptional situation. This approach is also suitable in the cases students have to implement fast solution of a given task, for example during an exam, and they need a simple mechanism to reduce the amount of errors in their code.

3.3.Terminate program with an error message

In this error handling technique, once an error is detected, a message is displayed, and the program is terminated. On the first glance this might appear quite ruff approach, and definitely is not the best way to deal with exceptional situations in real-life applications. However, this technique is extremely suitable for prototyping, and it may help programmer to reduce the mistakes in the initial version of the code. In C and C++ it is easy to implement it using the `assert()` macro (see Listing 3 below).

```
double circleRad(double area)
{   assert(area >= 0);
    return sqrt(area / PI);
}
```

Listing 3. The circle radius task solved with assertion to terminate the program with error message.

In this case we proceed from the assumption that detected errors must be eliminated from the final version of the source code. For that reason this technique is used in the tasks of *intermediate complexity*, for example in the initial implementation of some data structures such as linked lists, stacks and queues.

3.4. Return flag or external flag

In all previous error handling techniques, the error is detected inside the function, and it is processed again inside the function. In many situations in the practice, the error can be detected in the function, but it has to be processed in another program scope, very often where the caller of the function is. One of the standard techniques to achieve this effect is to signal to the functional caller the error using the return value of the function (Listing 4).

```
double circRad(double area)
{   double result = -1;
     if (area >= 0)
     {     result = sqrt(area / PI);}
     return result;
}
```

Listing 4. The circle radius task solved with return value denoting the error state.

Another approach is to set the value of a global flag. In C++ many mathematical routines use this way for error signalling. In Listing 5 we trigger the function `circRad()` (its initial version in Listing1) causing *function misuse* error. The `sqrt()` function will set the global flag errno to the state EDOM value that indicates a domain error.

```
errno = 0; // reset the error flag
circRad(-1);
if (errno == EDOM) // domain error
{cerr << "circRad() caused a domain error." << endl;
 }
```

Listing 5. The *function misuse* error indicated using the global flag errno set by `sqrt()` function.

Since error handling using return values and flags is typical for procedural programming languages, it is widely used in standard library functions inherited from the C programming language, and students must be familiar with it.

3.5. Error handler functions

Another approach that is also used in some standard libraries in C++ includes the error handler functions. The library provides a function that is triggered when exceptional situation is detected. For example consider the following function:

```
void error(){   action();}
```

The function `action()` is declared as a pointer to a function `void (*action)();`. The user implements a function that defines the action that must be taken, when the error occurs, so it can be set using the following installation function:

```
void setErrorAction(void (*userAction)())
{   action = userAction;}
```

Now the user can implement the function that determines how exactly the error will be handled

```
void argError()
{ cerr << "Wrong argument." << endl;}
```

and then installs it using the call `setErrorAction(argError);`. Then our circle radius example will be as given in Listing 6.

```
double circleRad(double area)
{   double result = 0.0;
   if (area < 0){      error();}
   else{  result = sqrt(area / PI);}
   return result;
}
```

Listing 6. The *function misuse* error processed using error handler function.

Even though this approach can look limited from a given point of view (Horstmann & Budd2008), it is actually quite powerful (Koenig and Stroustrup 1990), especially when using the principles of functional programming that are also available in C++ programming language. Also, note that errorhandler functions are used by low-level memory management system of C++, and hence a student at the *advanced level* must be familiar with them.

3.6. Exceptions

Exceptions exist in C++ since C++98 standard, and they have the tendency to replace the older mechanisms for error handling in many systems. In (Koenig & Stroustrup 1990) Stroustrup and Koening give the following motivation for exception handling mechanism: *A library routine can detect an error, but it cannot process it. A routine that is a user of the library knows how to process an error but it cannot detect it.*

In many cases we can consider that mechanism of exceptions, as Stroustrup and Koening point out, is more readable, more regular in style and more integrable with other parts of the program, compared to the alternative mechanisms to handle errors within a library. See also (Stroustrup 2019).

In some programming languages exceptions are even an indivisible part of the language. Java is an outstanding example (Horstmann 2019; Eckel 2011) with its mechanism of checked exceptions that will not allow the program to compile in the case they are not handled. However, we must also consider that exceptions are criticised (Sutter 2019), and even there are frameworks based on C++that do not allow exceptions – for example Qt (Eng 2019).

```
double circleRad(double area)
{   if (area < 0)
   {throw logic_error("Illegal parameter value.");}
   return sqrt(area / PI);
}
```

Listing 7: The *function misuse* error resulting in throwing an exception.

The circle radius example from above, implemented using exception throwing in Listing 7, shows the compactness and clearness of the error handling code using this mechanism. The fact that exceptions can be caught exactly in the scope of the program in which the code can recover successfully from the error shows its flexibility. In our courses on both *intermediate level* and *advanced level* exceptions are widely used, especially in advanced implementations of data structures such as trees, priority queues, heaps, maps and hash tables.

4. EMT based system of tasks

In contrast to the standard error-avoidance educational methods, EMT uses errors as an instrument to illustrate knowledge and develop skills. Students are not taught directly how to avoid errors, but actually are encouraged to provoke them, to analyze what causes them, techniques to handle, and correct them. Using a system of tasks that are formulated based on this principle, we show how complex notions can be formed and practical skills are developed. In this section we present few example tasks taken from our system of tasks that form the curriculum of our computer programming courses. Each task is classified with respect to the following features: *complexity*, *course*, *error*, and *handling mechanism*. The classification that we provide (given as a table in front of each task) is based on the proposed error classification scheme and error handling mechanisms.

complexity	basic
course	Introduction to computer programming
error	preprocessor, compiler, course specific
handling mechanism	assume no error will occur

Task 2. *Try to compile the "Hello, World!" program by consecutively omitting the following*
lines:

- `#include <iostream>`
- `using namespace std;`
- `return 0.`

When the student tries to compile the program with the omitted `include` preprocessor directive, the compiler responds with error message `'cout' was not declared in this scope`, and also suggests that `std::cout is defined in header <iostream> did you forget to #include <iostream>?`. Even though the error is generated by the compiler, we can classify it as a preprocessor error, because the problem is a missing preprocessor directive. This error clearly shows that the basic I/O system of C++ is not part of the core of the language; rather it is defined in a standard library.

Omitting `using namespace std;` introduces the notion namespace. Also this compile-time mistake is very common for novice programmers. Besides that, this error opens the discussion whether it is a good style to use the `using namespace` construction, or it is better to use the namespace with scope resolution operator.

The last error from Task 2 is an example of *style/course specific* category. In many literature sources omitting the `return` statement at the end of the `main()` function is not considered an error, however in our courses we have decided to label it as a bad style, because of the formal rule that a function must return data compatible with its return type.

complexity	basic
course	Introduction to computer programming
error	semantic/algorithm
handling mechanism	return flag

Task 3. *The sphere volume is given by the following expression:* $V = \frac{4}{3}\pi r^3$, *where r is the radius of the sphere. Write a program that reads the radius r of a sphere from the standard input and calculates its volume using the expression:* **double** *volume = 4/3*PI*r*r*r;. If the calculated volume is not a valid value, return flag -1. Why the answer produced by the above code is not correct?*

The error demonstrated in Task 3 shows the difference between integer division and floating point division. The usage of wrong operation, even though the cause of this error can be misunderstanding of the language rules, is a *semantic* error in the logic of the algorithm that is very common and hard to find by beginners.

complexity	intermediate
course	Introduction to computer programming
error	semantic/ function misuse
handling mechanism	terminate program with an error message

Task 4. *Implement a program that generates a string of small Latin letters with random length and random content. Consider the function* `randString()` *given below.*

```
string randString(int max_size)
{   assert(max_size >= 0); // assert function input
    int size = 1 + rand() % max_size;
    string result;int i = 0;
    while (i < size)
    {result.push_back(randSmallLatin());i++;}
    return result;
}
```

How the program will behave in negative inputs? What is the result, if there is no call to the `srand()` function?

The example in Task 4 shows how the input of a function can be verified using assertions –a technique that is widely used in our courses on intermediate level.

Also it helps demonstrating the random seed purpose, which is a key notion for explanation of the pseudo-random numbers generators.

complexity	advanced
course	Object-oriented programming
error	compiler/warning
handling mechanism	assume no error will occur

Task 5 *Implement a function that returns the address of a local variable. In the* `main()` *function print both the address and variable stored at that address.*

```
int* getAddress()
{   int local = 42;return &local;}
```

Even though the example in Task 5 may look simple on the first glance, it actually introduces important notions, such as the call stack of functions. The example code will result in warning that it attempts to return the address of a local variable, and to understand why this is a serious error, the learner has to develop also knowledge about different memory sections of a program, static variables and dynamic variables. The mechanism of pointers is always a source of confusion for students even at more advanced level, and errors similar to the demonstrated above help the development of these complex notions.

complexity	Advanced
course	Data structures
error	compiler/error
handling mechanism	assume no error will occur

Task 6. *Add a member function that returns the position of the parent node in a rooted tree. The function constructs* `Position` *objects from the parent pointer*

```
ptr_node->ptr_prnt
```

and returns it as a result. Add a member function that returns a reference to the list of children nodes pointers. Omit the keyword `typename` in the implementation of the two functions:

```
template <typename TKey>
typename GTree<TKey>::Position
GTree<TKey>::Position::parent() const {. . .}
template <typename TKey>
list<typename GTree<TKey>::Node* >
&GTree<TKey>::Position::chldList() const {. . .}
```

Task 6 is a part of sequence of tasks that leads the learner towards the implementation of the general tree data structure that represents a rooted tree. The omitting of the keyword `typename` in the implementation of the two functions will produce a compile-time error whose understanding requires deeper knowledge of generic programming in C++.

complexity	advanced
course	Data structures
error	semantic/data structure
handling mechanism	exceptions

Task 7. *For our implementation of a priority queue, we will need a user-defined exception class that is derived from the standard runtime error. Consider the implementation below:*

```
class PQueueExcept : public runtime_error
{   public:
  PQueueExcept(const string &msg);
};
// parameter constructor
PQueueExcept::PQueueExcept(const string &msg)
: runtime_error(msg){}
```

What will happen if you access the minimum element of the priority queue, or you try to remove it from the data structure in the case when the container is empty?

```
// access the min element
template <typename TElm, typename TCmp>
const TElm &PQueue<TElm, TCmp>::min() const
{   return llist.front();}
// remove minimum element
template <typename TElm, typename TCmp>
void PQueue<TElm, TCmp>::removeMin()
{   llist.pop_front();}
```

The solution of Task7 is to throw exception of the user-defined type within the member functions of the class PQueue. The students are encouraged to provoke the errors that are common for the given data structure, and to protect their code from them using the mechanism of exceptions. The understanding of these types of errors also aids the development of the notion of the particular data structure and the method that is used to implement it.

5. Conclusions

EMT is a natural approach of teaching in computer programming courses. This is because of the significance and specific nature of computer programming errors which we can observe from the variety of error classification schemes and different methods to handle errors. On the other hand, the process of provoking errors during the educational process highly diminishes the negative emotional influence on the learners (Heimbeck et al. 2003), which enhances both the theoretical notion development and the practical skills.

As future work we plan to adopt the EMT using the techniques of testing and debugging aiming to enhance the development of notions needed in teaching algorithms on beginner and advanced levels. We also plan to develop a system of tasks that uses errors as an instrument for illustration of knowledge in our course in parallel computing, following the specific nature of errors in the implementation of parallel algorithms.

REFERENCES

ASENOVA, P.&MARINOV, M.,2018. Teaching mathematics with computer systems. *Mathematics and Education in Mathematics.* Proceedings of the Forty-seventh Spring Conference of the Union of Bulgarian Mathematicians, Borovets, April 2 – 6, 2018, 213 – 221.

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

BOWEN, J. B., 1980.*Standard error classification to support software reliability assessment*. Proceedings of the National Computer Conference AFIPS'80, 697 – 705.

doi:10.1145/1500518.1500638

COHEN, B., 1994. The use of "bug" in computing. *IEEE Annals of the History of Computing,***16**(2), 54 – 55.

DYRE, L., TABOR, A., RINGSTED, C.& TOLSGAARD, M. G., 2017. Imperfect practice makes perfect:error management training improves transfer of learning. *Medical Education*, **51**(2), 196 – 206.

ECKEL, B, 2011.*On Java 8, Version 2*, Lean Publishing. http://leanpub.com/onjava8.

ENG, L. Z., 2019.*Qt5 C++ GUI Programming Cookbook*, 2nd edn, PACKT Publishing.

ETTLES, A., LUXTON-REILLY, A.& DENNY, P., 2018.*Common logic errors made by novice programmers*. Proceedings of the 20th Australasian Computing Education Conference ACE'18, 83 – 89.

FRESE, M., 1995. Error management in training: conceptual and empirical results.*Organizational Learning and Technological Change*,**16**(2), 112 – 124.

HEIMBECK, D., FRESE, M., SONNENTAG, S.&KEITH, N., 2003, Integrating errors into the trainingprocess: the function of error management instructions and the role of goal orientation. *PersonnelPsychology*,**56**(2), 333 – 361.

HORSTMANN, C., 2019.*Big Java: early objects*, 7th edn. Wiley.

HORSTMANN, C.& BUDD, T., 2008.*Big C++*, 2nd edn. Wiley.

KEITH, N.&FRESE, M.,2005. Self-regulation in error management training: emotion control and metacognition as mediators of performance effects.*The journal of applied psychology,* **90**, 677 – 691.

KEITH, N. & FRESE, M.,2008. Effectiveness of error management training: a meta-analysis.*The journal of applied psychology,***93**(1), 59 – 69.

KNUTH, D. E.,1989, The errors of tex.*Software: practice and experience,***19**(7), 607 – 685.

KO, A. J. & MYERS, B. A., 2003.*Development and evaluation of a model of programming errors*.Proceedings of IEEE Symposium on Human Centric Computing Languages and Environments, 7 – 14.

KO, A. J. & MYERS, B. A.,2005. A framework and methodology for studying the causes of softwareerrors in programming systems. *Journal of Visual Languages & Computing,* **16** (1 – 2), 41 – 84.

KOENIG, A. AND STROUSTRUP, B., 1990. Exception handling for C++', *Jornal of Object-Oriented Programming,* 3(2), 16 – 33.

LASKOV, L. M., 2016.*Programming in C++, examples and solutions, part one: From procedural towards object-oriented paradigm*, 1st edn. New Bulgarian University.

LASKOV, L. M., 2021.Introduction to computer programming through a system of tasks.*Mathematicsand Informatics,* **64** (6), 634 – 649.

MOTLEY, R. AND BROOKS, W. D., 1977.*Statistical prediction of programming errors*. ADA041106, Technical report, IBM Corporation.
https://apps.dtic.mil/sti/pdfs/ADA041106.pdf

STROUSTRUP, B., 2014. *Programming: principles and practice using C++*, 2nd edn, Addison-Wesley Professional.

STROUSTRUP, B., 2019.*C++ exceptions and alternatives*.P1947 Technical report, C++ Standards Committee Papers [online].
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1947r0.pdf.

SUTTER, H., 2019. *Zero-overhead deterministic exceptions: throwing values*.P0709 Technical report, C++ Standards Committee Papers [online].
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0709r4.pdf.

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