TECHNICAL TRANSACTIONS | CZASOPISMO TECHNICZNE
AUTOMATIC CONTROL | AUTOMATYKA
3-AC/2013

GRZEGORZ NOWAKOWSKI*

EFFECTIVE USE OF LAMBDA EXPRESSIONS IN THE NEW C++11 STANDARD

EFEKTYWNE WYKORZYSTANIE WYRAŻEŃ LAMBDA W NOWYM STANDARDZIE C++11

Abstract

In this paper, the possibility of lambda expressions and methods for their effective use in C++ code in the new C++11 standard have been presented. Studies that compare the execution times of the program and use lambda expressions as well as classical methods have been conducted. The results confirm the effectiveness of lambda expressions with respect to traditional methods.

Keywords: *lambda expressions, lambda – introducer, lambda – declarator, compound – statement, function object, STL, the new C++11 standard*

Streszczenie

W artykule przedstawiono możliwości wyrażeń lambda oraz zaprezentowano metody ich efektywnego wykorzystania w kodzie języka C++ w nowym standardzie C++11. Przeprowadzono badania porównujące czasy wykonania programu z zastosowaniem wyrażeń lambda i metod klasycznych. Uzyskane wyniki potwierdzają większą efektywność wyrażeń lambda w stosunku do tradycyjnych metod.

Słowa kluczowe: *wyrażenia lambda, lambda – introducer, lambda – declarator, compound – statement, obiekt funkcyjny, STL, nowy standard C++11*

* M.Sc. Grzegorz Nowakowski, e-mail: gnowakowski@pk.edu.pl, Department of Automatic Control and Information Technology, Faculty of Electrical and Computer Engineering, Cracow University of Technology.

1. Introduction

Each programmer using C++ language standard library is aware of the need to write short functions – predicates, which control algorithms. As a result, the code is being filled with short classes that perform simple operations. Simultaneously, the separation of implementation from the caller significantly impedes the modification and maintenance of the code. One exemplary solution to this problem is the use of lambda expressions which allows for the formation of anonymous function objects defined in the caller. Until the advent of the new C++11 standard, lambda expressions had not been supported by the language itself. The programmer had to refer to external libraries. In this paper, the possibility of lambda expressions and methods of their effective use in C++ code have been presented. Studies that compare the execution times of the program by using both lambda expressions and classical methods have been conducted. The results confirm the effectiveness of lambda expressions with respect to traditional methods. The suggested solutions significantly improve the readability of the code, as well as allowing for the disobeying of namespace classes and functions which are used in the code only once.

2. Theoretical basis

Lambda calculus is a formal system [2] that allows for the defining of functions and for providing them with arguments. The system was introduced by Alonzo Church and Stephen Cole Kleene in 1930. Initially, it was the alternative to the theoretical approach to the fundamentals of mathematics in which the function (intentionally understood as a definition) was the primary concept. Although this project was not successful, it soon became apparent that the lambda calculus is a useful apparatus in the theory of computation. It is also complete in the sense of Turing [4]. It means that all algorithms which are capable of being enrolled in the lambda calculus can be implemented on a Turing machine and vice versa. Later, lambda calculus [2] has proved to be an indispensable tool in the theory of programming languages as a consequence of information technology development, and in practice, became the inspiration for functional programming languages such as LISP.

We take into consideration objects called *lambda-term* [2] in lambda calculus. Traditionally, they are defined as formal expressions of a certain language like regular expressions in algebra and first-order logic. The essential difference is that the lambda-expressions in a remaining equivalence relation (alpha-conversion) are identified with each other (considered as identical) and thus, they are not expressions but the abstraction classes of lambda-expressions (also known as pre-terms).

Let us assume that there is a countable, infinite set of subject variables [1–4]:

– subject variables are *lambda-expressions*,
– if M and N are *lambda-expressions* then (MN) is a *lambda-expression*,
– if M is a *lambda-expression* and x is variable then (λx.M) is a *lambda-expression*.

The expression of the form (MN) is called an **application** [3] who passes (applied) argument N on to the function M. In contrast, the expression of the form (λx.M) is a **functional abstraction** [3] representing an intuitive anonymous function that takes one argument and passes it on to M.

Listing 1. Examples of lambda expressions

1:	λx.x (*identity*)
2:	λx.y (*constant expression*)
3:	λf.λx.x

Listing 2. Exemplary applications of function to argument

1:	(λx.xyx)z → zyz (*insert 'z' at the place of all x (x is bound by λ) in the expression XYX*)
2:	(λa.xyx)z → xyx (*no change, the variable 'a' is not present in the expression*)

3. Lambda expressions in C++11 and methods of their effective use in the code

Lambda calculus introduced in the new standard C++11 is very similar to the assumptions presented in the previous section – theoretical basis. The notation [7] is slightly different due to the lack of the symbol λ on most keyboards and some extensions.

In order to start working with lambda expressions or some random mechanism of the new standard C++11, we should use the Microsoft Visual Studio 2012 [5], or GCC version at least 4.6 [6] (on the command line using the '–std = c + +11').

In listing 3, some examples of lambda expressions are indicated.

Listing 3. Simple lambda expressions in C ++11

```
1: [ ] (int x) -> int { return x; };
2: [ ] (int x, int y) -> int { return x+y; };
3: [ ] (int x, int y) -> int { int z = x+y; return z; };
```

It is easy to notice that none of the examples match the definition presented in the previous section. There is the identity in the first line that has the specified type *int*, which in turn limits its use only to integers. The original expression could be applied to anything and also to itself. The second and the third lines contain a plus sign, which is not on the approved list of symbols. Additionally, the expression in the third line defines a temporary variable *z*.

Lambda expressions in C++11, in comparison with those of a mathematical lambda calculus, except those indicated in the above listing, can perform any instructions associated with external functions and create objects or use external variables. They also constitute a great convenience for programmers especially in the context of the use of STL algorithms [8]. The following figures show lambda expression grammar in C ++11.

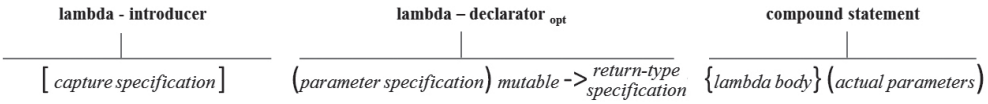

Fig. 1. Lambda expression grammar

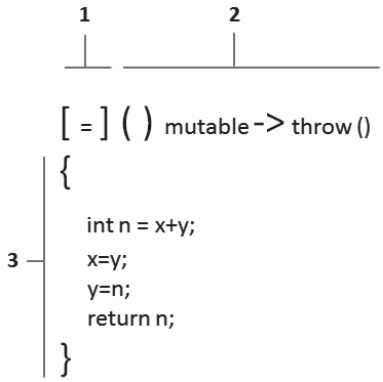

Fig. 2. Sample lambda expression (1 – lambda-introducer, 2 – lambda-declarator, 3 – compound statement) [10]

3.1. Lambda – introducer

Each lambda expression [8] must start with a pair of square brackets [ ] which may contain so-called *lambda-capture*. They are used to pass on additional parameters to the expression. Lambda expression can refer to any automatic variables in the given scope [7]. Also, the so-called modifiers, which specify whether a variable is in lambda visible by reference (by the use of the symbol & before the variable) or by value, have been made available.

Listing 4. Passing parameters by value

```
1:  int x=1, y=1, z=1;
2:  [ ] ( )  { cout << “Example 1”; } ( );
3:  [ ] ( )  { cout << x; } ( );
4:  [x] ( ) { cout << x+1; } ( );
5:  [=] ( ) { cout << x << y; }( );
```

Listing 4 shows the use of *lambda-capture* for capturing parameters. The example of the second line is a phrase which does not capture any variables. The example, in the third line, will not compile because the variable *x* was not passed on and cannot be used inside expressions. In the fourth line, the variable *x* has been explicitly passed on to the expression. In the fifth line, the operator has been used the =, which indicates that all variables have been transferred. In both cases, passing parameters by value have been applied, which causes copy operators to be sequentially dispatched for *all* objects (*all* here means for those which occurred in the expression, and according to this principle the variable *z* will not be copied because it has not been used). The application of variables (as in the fourth and fifth lines) differs slightly from the classical approach as far as the function is concerned. When the function is called, parameters are given explicitly each time, and thus have to exist at the time of the call.

Listing 5. Passing parameters by value

```
1:  function< int ( ) > fvalue;
2:  {
3:    int x=0;
4:    fvalue = [x] { return x+1; };
5:  }
6:  cout<< fvalue();
```

Listing 5 (which uses a template *function* with a header *functional*) illustrates the fact that when the value of the expression is induced in the sixth line, variable x – no longer exists despite the program working correctly. It results from the fact that a copy of the variable x is located in a lambda expression. It should be noted that the variables captured by value cannot be modified. It may seem inconsistent with the classical function call where any operations can be made on the copy of received variables. However, if we look at it in terms of a constant object, there seems to be a logical explanation.

However, in case there is a need to modify the captured variables, they should include the keyword *mutable* [8–9] which causes the lambda expression not to be treated as constant, and thus allows such a modification (listing 6).

Listing 6. Use the keyword mutable

```
1:  function< int ( ) > fvalue;
2:  {
3:    int x=0;
4:    fvalue = [x] ( ) mutable { return ++x; };
5:  }
6:  cout<< fvalue() << " " << fvalue();
```

As it has been mentioned above, in addition to passing parameters by value, there is also passing it by reference [7]. Both types of capture can be combined (listing 7). However, note that if there is capturing by value we have a copy of the object whereas if there is capturing by reference we do not have one. Passing by reference works here more quickly. Yet, do not apply it to a non-existing object because it will result in a runtime error.

Listing 7. Passing parameters by value and references

```
1:  int x=1, y=1, z=1;
2   // all variables are passed by reference
    [&] ( ) { cout << x << y << z; }( );
3:  // z passed by reference, the rest of the variables by value
    [=,&z] ( ) { z = x + y; }( );
4:  // x passed by value, the rest of the variables by reference
    [&,x] ( ) { y = x; z = x; }( );
```

3.2. Lambda-declarator

The second component of the lambda expression [8] is a *lambda-declarator* (Fig. 3), wherein it should be noted that it is optional, and the correct lambda expression cannot included in it at all.

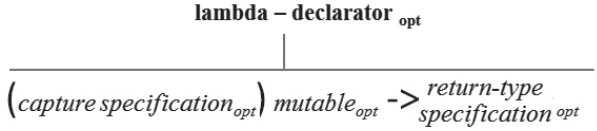

Fig. 3. Lambda-declarator

Lambda expression [12] as well as the ordinary functions can have arguments which are passed on to the function at runtime. It should be remembered that the same rules are applied here as for the function arguments. If the expression consists of more than a single return statement, it must specify the type of the return value. Listing 8 shows examples of declarations of arguments with the unauthorized case (in comments).

Listing 8. Declaration of the list of arguments

```
1:  // expression takes two parameters by value
    // specify the return type is not required
    [ ] (int x, int y) {return x + y; } (1,2);
2   // compile error
    // trying to pass x variable rvalue by not constant reference
    //[ ] (int &x, int y) {return x + y; } (1,2);
3:  // expression takes one parameter as a constant reference, the second by value
    // specify the return type is required
    [ ] (const int &x, int y) -> int {int z = 3; return x + y + z; } (1,2);
4:  // returning a reference to a temporary variable
    [ ] (int x, int y) -> int& {int z = 0; return z; } (1,2);
```

When we need a lambda expression to apply only once, we can call a lambda function in the place of the definition (assignment to the named variable is unnecessary). The use of a complicated variable declaration by means of the *function* can be replaced by the application of *auto* type [8–9]. It allows for the use of the expression repeatedly without the necessity to define the expression each time. The following considerations have been presented in listing 9.

Listing 10 shows the use of a lambda expression to solve the problem of sorting a (descending) vector that contains elements of an integer. For comparison, the implementation of the same problem in the classical approach has been presented. A programmer who does not know lambda functions would have to write their own sorting function or use the standard library *sort* algorithm. However, in the case of using the *sort* function, they would have to

create a class with overloaded *operator* (), whose object could perform a comparison. Such an approach would require writing a few lines of code and naming the function for a single use. Additionally there would be a separation here from the implementation of the caller, which would significantly hamper modification and maintenance of a code. It is difficult to disagree with the notion that the use of lambda functions is much simpler and clearer.

Listing 9. Calling the lambda

```
1:  // assign a lambda function in place of the definition of the named object
    function< void ( int ) > f_1 = [ ] ( int x ) { cout << "Example 1" <<x; };
2   // assign a lambda function in place of the definition of the named object
    // use of 'auto' (the compiler itself in this case will determine the type of the
    // variable)
    auto f_2 = [ ] ( int x ) { cout << "Example 2" <<x; };
3:  // call these functions to named objects
4:  f_1(1);
    f_2(2);
5:  // lambda function call in place of its definition in the unnamed object
    [ ] (int x) { cout << "Example 3" <<x; } (3);
```

Listing 10. Solving the problem of sorting vector by means lambda expressions

```
1:  //new standard C++11
    sort(v.begin(), v.end(), [ ] (int x, int y) { return x > y;});

2:  // classical approach
    class comparator
    {
      public:
            bool operator() (int x, int y) const
            {
                  return x > y;
             }
    };

    sort (v.begin(), v.end(), comparator());
```

The final component [8] of the *lambda-declarator* is the *return type*. In the previous listings it has been indicated that lambda functions can return a value by means of the *return* like ordinary functions. The difference is that the return type has not been declared anywhere. The standard [11] not only allows for defining the type explicitly, but also imposes an obligation on the compiler automatic determination (only for expressions like *return ...*).

Listing 11 shows the different combinations of returned values by lambda expression with illegal cases (in the comments).

Listing 11. Examples of different combinations of returned values by lambda expression

```
1:  // explicitly provide a return type
    int p1 = [ ] (int x, int y) -> int {return x+y;}(1,2);

2:  // no return type – compiler itself determines the type of
    int p2 = [ ] (int x, int y) -> {return x+y;}(1,2);

3:  // abuse (but works by using gcc 4.6)
    // compiler is required to determine the type returned only for expressions like // return ...
    //int p3 = [ ] (int x, int y) -> {if (x%2) return x; else return y;}(1,2);

4:  // compile error - there is a mismatch type declared and returned
    //int p4 = [ ] () -> int {return false;}();

    // compile error - it is impossible to determine the return type (such as
5:  // inconsistent with the specification)
    //int p5 = [ ] (int x) { if x > 5 return x; else return true;}(1);
```

3.3. Compound-statement

The third component of the lambda expression [8] is a *compound-statement* (Fig. 4). Body expressions can contain anything that the contents of an ordinary method or function would contain.

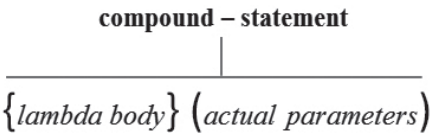

Fig. 4. Compound-statement

4. Practical applicability: function pointers, functors and lambda expressions

The example has been investigated, on the basis of which three approaches of passing information to the STL algorithm have been illustrated: through function pointers, functors and lambdas [8]. A list of random integers has been generated and it has been examined how many of them are divisible by 5 and how many of them are divisible by 7. Generating a sequence of values is based on the use of *vector <int>* array to hold the numbers on the use of the STL *generate* () algorithm to stock the array with random numbers.

The *generate*() function takes a range, specified by the first two arguments, and sets each element to the value returned by the third argument that is a function object taking no arguments. In the described example, the function object is a pointer to the standard *rand*() function.

Listing 12

```
1:  vector<int> n(1000);
2:  generate(vector.begin(), vector.end(), std::rand);
```

4.1. Passing on information to the STL algorithm by function pointer

In order to count the number of elements divisible by 5 or 7, *count _if* () algorithm has been applied. The first two arguments should specify the range just as in the case of *generate*(). The third argument should be a function object that returns true or false. Thus, the *count _if* () counts all the elements for which the function object returns true. To find elements divisible by 5 or 7, these function definitions have been used:

Listing 13

```
1:  bool fcount_5(int x) {return x % 5 == 0;}
2:  bool fcount_7(int x) {return x % 7 == 0;}
```

By means of all above definitions, elements have been counted in the following way:

Listing 14

```
1:  int count_5 = std::count_if(n.begin(),n.end(),fcount_5);
2:  cout << “Count of numbers divisible by 5:” << count_5;

3:  int count_7 = std::count_if(n.begin(),n.end(),fcount_7);
4:  cout << “Count of numbers divisible by 7:” << count_7;
```

4.2. Passing on information to STL algorithm by functors

A similar functionality has been implemented by means of a functor. The functor is a class object that can be used as the name of the function thanks to the class defining *operator*()() as a class method. The advantage of this functor in this example is the fact that one can use the same functor for both counting tasks. The following definition of the functor has been applied:

Listing 15

```
1:  class f_count
2:  {
3:        private:
4:           int divisor;
5:        public:
6:           f_mod(int d = 1) : divisor (d) {}
7:           bool operator() (int x) {return x % divisor == 0;}
8:  };
```

The elements have been counted as follows:

Listing 16

```
1:  count_5 = std::count_if(n.begin(),n.end(),f_count(5));
2:  cout << “Count of numbers divisible by 5:” << count_5;
3:  count_7 = std::count_if(n.begin(),n.end(),f_count(7));
4:  cout << “Count of numbers divisible by 7:” << count_7;
```

The argument *f_count*(5) creates an object that stores the value 5, and *count_if*() uses the created object to call the *operator*()() method, setting the parameter *x* equal to an element of *n*. In order to count the numbers which are divisible by 7 instead of by 5, one has to use *f_count*(7) as the third argument.

4.3. Passing on information to STL algorithm by lambda expression

As it has already been mentioned, lambda expressions allow for applying an anonymous function definition (a lambda) as an argument to functions that are expecting a function pointer or functor. The lambda corresponding to the *fcount_5(int)* function has been defined below:

Listing 17

```
1:  [ ] (int x) {return x % 5 == 0;}
```

It resembles the definition of *fcount_5(int)* to a great extent:

Listing 18

```
1:  bool fcount_5(int x) {return x % 5 == 0;}
```

In fact, the differences consist in replacing the function name with [ ]. Also, there is no declared return type. The return type is the type that *decltype* would define on the basis of the return value, which would be *bool* in this case. If the lambda did not have a return statement, *void* would be the return type. In the example below, lambda expressions have been applied in the following way:

Listing 19

```
1:  count_5 = std::count_if(n.begin(), n.end(), [ ] (int x) {return x % 5 == 0;});
2:  cout << “Count of numbers divisible by 5:” << count_5;
3:  count_7 = std::count_if(n.begin(), n.end(),[ ] (int x) {return x % 7 == 0;});
4:  cout << “Count of numbers divisible by 7:” << count_7;
```

It can be easily noticed that the entire lambda expression has been put in the place of a pointer or a functor constructor. The automatic type deduction for lambda works only if the body consists of a single return statement. In other cases one has to explicitly specify the type of the return value by using a new syntax and later by means of type declaration.

5. Studies comparing the execution times of the program by means of both lambda expressions and classical methods

Tests have been conducted on a PC equipped with an Intel Core i5 2.80 GHz and 4 GB of memory. The test program was based on the application of a lambda expression to solve the problem of summing one hundred thousand products of the individual elements of the two-dimensional array of a randomly selected size range of (1, 100.000), compared to solving the same problem in a classical approach by using standard function. The computation does not matter, random numbers have been selected to avoid any compiler optimizations in this area. As a consequence we are exclusively interested in the difference between standard functions and lambda expressions. Different mechanisms for passing parameters have been tested: (1) lack of variables, (2) one variable in explicitly, (3) one variable by reference, (4) two dimensional arrays explicitly. The program marked the execution times of the implemented test.

Table 1

Execution times of the tests

Test number	Lambda expression [ms]	Standard function [ms]
1	34+–1	35+–1
2	33+–1	36+–1
3	35+–1	35+–1
4	35+–1	34+–1

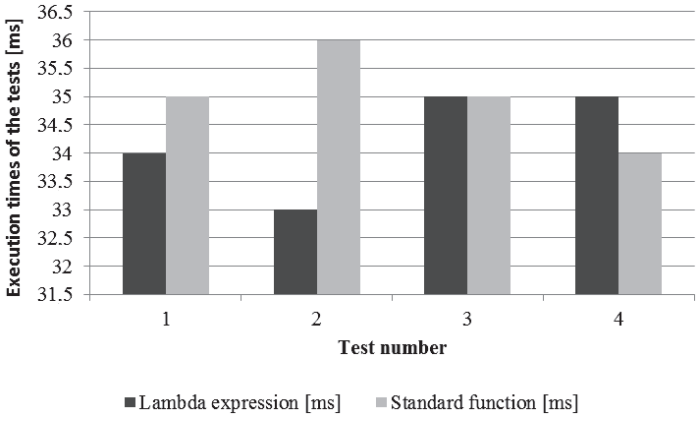

Fig. 5. Execution times of the tests

By analyzing the obtained results of the tests, it was found that both the standard functions and the lambda expression perform in a similar period of time, which means there is no significantly different loss of performance when using lambda. Only in the case of the fourth test were better results obtained by standard functions (Fig. 5). However, taking into account all the possibilities of these expressions mentioned in the article, the obtained results confirm better efficiency of lambda compared to traditional methods.

6. Conclusions

In this paper, the possibility of lambda expressions and methods for their effective use in C++ code in the new C++11 standard have been presented. For this purpose, four qualities have been taken into account: proximity, brevity, efficiency and capability as well as the lambda expression have been juxtaposed with functions and function objects (functors).

Many programmers claim that it is useful to place definitions close to where they are used. This practice allows them to avoid searching for multiple pages of source code to find the definition used in a particular place. It also facilitates modifying the code because the components are immediately modified within the definition. From this standpoint, a lambda expression is the most effective because the definition is at the point of usage. Functions are the least efficient because they cannot be defined inside other functions, so the definition will be located possibly quite far from the point of usage. Function objects in this regard are more efficient than functions, because a class (including a function object class) can be defined inside a function so that the definition of the functor can be located close to the point of use.

In terms of brevity, the function object code is much more extensive than the equivalent function or lambda expression. Functions and lambdas are approximately equally brief. The exception is when we want to use a lambda expression more than once. However, the repeated use of lambda expressions is not really necessary. Although it is admittedly an anonymous function, it can be associated with the name (using the type of *auto*), and then this name may be used as often as desired. In contrast to the ordinary function, a lambda expression can be defined inside a function.

While comparing execution times of the program by means of both lambda expressions and classical methods, lambda expressions proved to be more efficient in comparison with traditional methods.

As far as the issue of the possibility is concerned, lambda expressions offer some additional capabilities. In particular, a lambda can access by name any automatic variable in scope. Variables to be used are captured by having their names listed within brackets. If just the name is used, as in [x], the variable is accessed by value. If the name is preceded by an &, as in [&x], the variable is accessed by reference. Using [&] provides access to all the automatic variables by reference, and [=] provides access to all automatic variables by value. For instance, [x, &y] would provide access to x by value and y by reference, [&, x] would provide access to x by value and to all other automatic variables by reference, and [=, &y] would provide access by reference to y and by value to the remaining automatic variables.

The main motivation for adding lambdas to C++ was to enable using a function – like expression as an argument to a function that is expecting a function pointer or functor as an argument. So the typical lambda expression is a test expression or comparison expression that can be written as a single return statement. This keeps the lambda function short and easy to understand and enables the automatic deduction of the return value. However, it is likely that a subset of the ingenious C++ programming community will develop other uses.

References

[1] Hankin Ch., *Lambda Calculi. A Guide for Computer Scientists*, Clarendon Press, 1994.
[2] Urzyczyn P., *Rachunek lambda*, wykład monograficzny, (online) homepage: http://www.mimuw.edu.pl/~urzy/Lambda/erlambda.pdf, (date of access: 2014-04-01).
[3] Barendregt H., Dekkers W., Statman R., *Lambda calculus with types*, Cambridge University Press, 2013.
[4] Barendregt H.P., Manzonetto G., *Turing's contributions to lambda calculus*, 2013.
[5] Microsoft Visual Studio Professional 2012, (online) homepage: http://www.microsoft.com/en-us/download/details.aspx?id=30682 (date of access: 2014-04-01).
[6] GCC 4.6 Release Series, (online) homepage: http://gcc.gnu.org/gcc-4.6/ (date of access: 2014-04-01).
[7] Siddhartha R., *Sams Teach Yourself C++ in One Hour a Day (7th Edition)*, Pearson Education, 2012.
[8] Prata S., *C++ Primer Plus, Sixth Edition*, Addison Wesley, 2011.
[9] SDJ, (online) homepage: http://sdjournal.pl/magazine/1778-zarzadzanie-pamiecia-w-sun-jvm (date of access: 2014-04-01).
[10] Lambda Expression Syntax, (online) homepage: http://msdn.microsoft.com/en-us/library/dd293603.aspx (date of access: 2014-04-01).
[11] Stroustrup B., *The C++ Programming Language*, Addison Wesley, 2012.
[12] Stroustrup B., (online) homepage: http://www.stroustrup.com/C++11FAQ.html (date of access: 2014-04-01).