Computer and Information Science; Vol. 6, No. 2; 2013
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Training in Object-Oriented Programming and C++11

Ivaylo Donchev[1] & Emilia Todorova[1]

[1] Faculty of Mathematics and Informatics, St Cyril and St Methodius University of Veliko Turnovo, Veliko Turnovo, Bulgaria

Correspondence: Emilia Todorova, Faculty of Mathematics and Informatics, St Cyril and St Methodius University of Veliko Turnovo, Veliko Turnovo 5000, 3 Arch. G.Kozarov str., Bulgaria. Tel: 359-62-600-461. E-mail: emilia_todorova@yahoo.co.uk; e.todorova@uni-vt.bg

Received: February 18, 2013 Accepted: March 12, 2013 Online Published: March 20, 2013

doi:10.5539/cis.v6n2p84 URL: http://dx.doi.org/10.5539/cis.v6n2p84

Abstract

C++ is the most commonly used language in introductory and intermediate programming courses in many universities. This language had its rapid development in recent years. Its abstractions are more flexible and affordable than ever before. There are so many new features in the new standard, known as C++11, that it may be considered a new language. All these changes should find their place in teaching and will force us to utterly reorganize our programming courses. In this paper we comment on the situation in programming courses at our University and share experience in teaching C++11 in the course of Object Oriented Programming (OOP). We justify the relevance of the chosen language features and fix certain difficulties in proper mastering by students. We provide some specific recommendations for overcoming these difficulties.

Keywords: object-oriented programming, C++11, teaching, multiparadigm

1. Introduction

Programming is a very important skill which anyone studying computer science must improve (ACM/IEEE-CS 2001, p. 22). This is so because there is a strict correlation between quality of programming and other computer skills as abstraction, conceptualization, design and testing the quality of programs. Programming is a creative process, which allows expressing abstract ideas and this makes it more than a professional tool. Through programming students acquire many decisive for a professional skills - critical and analytical thinking, critical attitude to components and details and so forth. According to the conclusions of report CC2005 (ACM/AIS/IEEE-CS 2005, p. 35), any worthy of respect curriculum must include programming.

Teaching programming is traditionally accompanied by a considerable number of difficulties. In spite of large experience, acquired for more than 50 years, it is still considered a challenge (Caspersen & Bennedsen, 2007, p. 111), especially in introductory courses (Meyer, 2003). Many researchers define the study of programming as a particularly difficult activity (Bergin & Reilly, 2005; Boyle, Carter, & Clark, 2002; Gomes & Mendes, 2007) and (Simon et al., 2006). The reasons for this situation have deep roots: there is no compliance about the nature of the process of programming, and further, about the methods of its teaching. However, there is no doubt that programming is a complicated combination of science and art, a complex occupation, and it is even more complex to teach it (Bennedsen, Caspersen, & Kölling, 2008).

C++ is the most commonly used language for programming in introductory and intermediate courses in Bulgarian universities. This choice is not accidental: C++ is a hybrid language and allows the developer to take advantage of several programming paradigms. On the one hand, it gives teachers the freedom to choose methodology, on the other – the students have to learn at least two software paradigms. Recommendations are to master at least two paradigms (ACM/IEEE-CS, 2008, p. 19), but using C++ allows to accomplish these recommendations at an early stage of training. One cannot deny that C++ is heavy for beginners. Its great potentialities go hand-in-hand with a complex syntax. Thus, given the difficulties encountered by our students, intensive language development in recent years, as well as modern requirements for high reliability, security, efficiency of programming, we began restructuring CS1 and CS2 courses aiming to make them as easy as possible, without compromising the quality of training. The main impetus for this was the new standard (ISO/IES, 2011) for the C++ language. This standard reveals the important changes that must sooner or later be reflected in the curriculum.

2. About Programming Courses at Our University

The course Introduction to Programming for the bachelor programs in our faculty is built according to the classical *imperative-first* approach and the language is C++. This method performs introduction into programming following the historical development of program paradigms and languages: it begins from the basic concepts of imperative and procedural programming, after which goes on to objects. The topics of the introductory course include data types, operators, arrays, files, and technology for debugging and testing programs. Here the most serious call throws the transition from procedural to object-oriented programming (*paradigm shift*). In order "to soften" this transition, instructors use object-oriented languages; however, introductory course focuses on the imperative aspects of the language: expressions, operators, functions and other elements of the traditional procedural model. The techniques of object-oriented programming (OOP) and design are left for the following year of instruction.

The *imperative-first* approach may be accomplished within two or three semesters. CC2001 (ACM/IEEE-CS, 2001) has the advantage of two-semester courses through the implementation CS111I: Introduction to Programming, and CS112I: Data abstraction. The first course introduces the concepts of procedural programming while the second builds a superstructure: ability to program in OOP style. In our University for the bachelor program in Computer Science this is accomplished in three semesters, through alternative courses. More often in the universities in our country the implementation is three semesters, as defined in (ACM/IEEE-CS, 2001): CS101I - Introduction to Programming, CS102I Object-oriented Paradigms and CS103I Algorithms and Data Structures. The advantage of the second approach is the greater amount of teaching classes that allows us to discuss more topics and be sure that the students have mastered the main elements of each topic before proceeding further. Introductory courses for the specialty Informatics degree course in the VTU are organized this way, but Algorithms and Data Structures course precedes Object-oriented Programming one.

A typical problem for the *imperative-first* approach is the fact that students have less time to get familiar with the techniques of OOP compared to the *object-first* approach. Mastering these techniques is a basic requirement for their training. If we add numerous difficulties when studying OOP, postponement of introducing to the object-oriented paradigm is a disadvantage of the *imperative-first* approach. Therefore curricula developed on it should include additional study of object-oriented design at a later stage. We have implemented parallel study of OOP and Software Technology, and the second includes elements of object-oriented analysis and design. On the other hand it is very important that students are well acquainted with the traditional imperative programming style. Any object oriented language supports procedural constructs and they are parts of the implementation of any method of any class.

In our university declarative style of programming is presented by Functional Programming and Logic Programming, which are studied in parallel in the fourth semester. The study of other important programming languages like Java, C #, and various script and markup languages takes place in the intermediate and advanced levels of training through elective courses.

3. Object-Oriented Programming

3.1 Move Semantics

The first element that we used was the rvalue reference type. This feature was available in Microsoft Visual Studio 2010 and naturally found its place in the CS2 course in OOP in the summer semester of 2009/2010. With rvalue reference it is easy to implement move semantics. It could be contended that from the point of view of efficiency this is the most important feature of the C++11, thus we pay special attention to it. Move semantics allows the compiler to replace the very expensive operation of copying objects with less expensive "state movement" operations.

We have to admit that learning OOP with C++ is more difficult than pure OOP languages (Smalltalk, Eiffel, Java, Simula). These difficulties come from the philosophy of C++ and inherited from C, namely - code efficiency. Price for this efficiency is that the programmer has to take care of many things instead of relying on the compiler. An example that has entered long ago into the training course in OOP is the implementation of the so-called "Big Three" methods (Bergin & Reilly, 2005): destructor, copy constructor and copy assignment operator for each class that has a data member, pointing to external resources, or encapsulating complex data structures ("the law of the big three"). The way of implementing the last two methods determines the well-known and successfully utilized copy semantics or value semantics.

In many situations data copying is a necessary and a proper operation, but in others the duplication of resources is not justified. Such is container reallocation, string concatenation, swap operations. The classic example that we give to our students is swapping the values of two objects:

```
template <class T> void swap(T& a, T& b){
    T buf(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = buf;    // now we have two copies of buf
}
```

In case when arguments are numeric or boolean such implementation of `swap()` is effective. But if string values are being swapped (objects of `std::string`), this means to create a new temporary object, to copy a string into it and then copy strings two more times. If the objects swapped are containers the disadvantages of this approach are even more evident.

Container reallocation implemented by copying is another vivid example of inefficiency. It is the need to allocate a new memory buffer large enough to hold the existing container elements and the new ones as well, to call the copy constructor for each element of the new buffer, to call the destructor for the original elements and to free the memory for the original buffer.

The solution is to write code for moving resources (dynamic memory) from one object to another, i.e. to implement move semantics. For the example of swapping values of strings it means to exchange the values of internal pointers and values of object sizes. The implementation of such an operation requires a special function. In order not to do so in each case in 2002 the first proposal was made to introduce new language elements that would allow direct encoding and universal use of move semantics (Hinnant, Dimov, & Abrahams, 2002). These features are provided by the new reference type (rvalue-reference), and make it possible to move resources from temporaries. This type is also available in C++10 versions of popular compilers even before the standardization of C++11. Thus, in combination with the old reference type (lvalue-reference) you can easily code move semantics (Hinnant, Stroustrup, & Kozicki, 2008). This semantics works precisely because it is possible to move resources from temporaries which cannot be referenced in another point of the program.

Language development is accompanied by changes in STL library (the standard library). Even in older versions container operations and algorithms are optimized to use move semantics, if possible. As a result efficiency is significantly increased, and tests show (Hinnant, 2010) that handling large and complex structures using STL algorithms is more than 12 times more efficient for move semantics.

Copy semantics has a special place in the OOP training course in C++ since one of the main goals of education is to acquire ability to write (because most of the students are future professional programmers) reliable, secure and efficient code (ACM/IEEE CS, 2008), and that is not possible without the methods of copy semantics. For the same reason we must not neglect the move semantics. Even if students are not trained specifically to apply it, they use it by STL containers and algorithms. Even in the introductory course, studying character strings and swapping two strings using the operator `std::swap(s1,s2)` students use the move-version of the function template. We believe that future professionals need to know the mechanism behind it and to design classes taking advantage of it.

A move constructor and a move assignment operator should be defined to add move semantics to a certain class. In some cases (ISO/IEC, International Standard, 2011, p. 278-285) the compiler does the work adding implicit versions of these methods. In both situations, the copy and assignment, when sources are rvalues, will automatically benefit from move semantics.

We use the following example to make more easy learning of new syntax constructions. It has a single data member that is a pointer to a buffer in memory:

```
class X {
    char* data;
public:
    X(X&& other):data(std::move(other.data)){ // move constructor
        other.data = nullptr;
    }
X& operator=(X&& other){ //move assignment operator
    if(this == &other) return *this; // self-assignment
    if(data) delete[] data;
    data = std::move(other.data);
    other.data = nullptr;
    return *this;
}
//......................
};
```

Initialization and assignment at pointers level is performed, and then the data members of the source object (`other`) are set to their default values. The last is important, as does not allow the destructor to repeatedly release resources (such as dynamic memory). Use of `std::move()` in this case is not obligatory (a reference to integral type is copied), but we encourage students to do so in order to get accustomed to use it for activation of move semantics.

Not only classes, but also individual functions or operators can benefit from the advantages of move semantics. It is useful to demonstrate this by predefining the template function `swap()`, which copy version we reviewed earlier:

```
    template <class T> void swap(T& a, T& b) {
        T buf (std::move(a)); //Move constructor call
        a = std::move(b);     //Move assignment operator call
        b = std::move(buf); } //Move assignment operator call
```

3.2 Perfect Forwarding

Rvalue-reference type can be used for the implementation of the perfect forwarding: a problem of generic programming in C++ unsolved until now with other means (most often developers of different kinds of libraries meet this problem). Perfect forwarding is an important C++11 technique. It allows move semantics to be automatically applied even when the source and the destination of the forwarding are separated by intervening function calls. The idea is that the template function could transfer its arguments through another function and keep the lvalue / rvalue character of the arguments using `std::forward()`. Therefore, avoiding extra copying, the author of the template does not have to repeatedly overload functions for various combinations of lvalue- and rvalue-references.

Often examples include constructors and mutators, which forward the received arguments to data members of the class they initialize or set, and standard library functions as `make_shared()`, which "perfect forwards" its argument to the class constructor for an object of type `T` to be created, and returns an object of type `shared_ptr<T>` that owns and stores a pointer to it. We introduce students to the problems with the classical example of a generic factory function that returns `shared_ptr` for a newly constructed generic type. Factory functions like this are very useful for encapsulating and localizing the allocation of resources. They must accept exactly the same set of arguments as constructors of created objects. We show the disadvantages of implementation tools available before C++11, and following a similar argumentation as in (Hinnant, Stroustrup, & Kozicki, 2008), come to present-day solution:

```
    template<class T, class Arg>
    shared_ptr<T> factory(Arg&& arg){
        return shared_ptr<T>(new T(std::forward<Arg>(arg)));
    }
```

Another suitable example for educational purposes is the perfect forwarding constructor that copies lvalue arguments to the class members and moves rvalue arguments. This can be achieved by templatized constructor, forwarding its rvalue reference parameters to class members.

```
class Widget {
public:
    template<class T1,class T2> Widget(T1&& n, T2&& c):
        name(std::forward<T1>(n)),          // forward n to string constructor
        coordinates(std::forward<T2>(c))    // forward c to vector constr.
     {}
    // ................................
private:
    std::string name;
    std::vector<int> coordinates;
};
```

So when creating objects of the class `Widget` move semantics will be applied where possible, for example:

```
Widget w {"My widget", {10, 20}};
```

The example can be expanded by adding a set of templated mutators to class `Widget`:

```
template<typename T> void setName(T&& n) {
    name=std::forward<T>(n);
}
template <typename T> void setCoords(T&& c){
    coordinates=std::forward <T>(c);
}
```

In general, generic and object-oriented programming are in very closely relations in C++.

3.3 Further Development of the Course

In 2010/2011, in the training course on the OOP were included only implementation of move semantics and perfect forwarding through rvalue reference. In 2011/2012 three new language features were added that are available in VC10: automatic deduction of the type by initializer in the declaration (`auto` keyword), `decltype` and the new suffix return type syntax. The following example combines the use of the three:

```
template<class T, class U>
auto sum(T x, U y)->decltype(x+y){return x+y;}
```

The type of the returned by template function `sum()` result is properly defined for different types of arguments.

Since 2012, we pay attention to templates with a variable number of arguments (variadic templates). In the tutorial we show classic examples of the universal and type-safe `printf()` function, simple tuple class, as well as some own examples.

In the past two years, our students choose to attend the elective Graph Algorithms which runs parallel with the course of OOP in the fourth semester. It also uses C++, as students gain experience and can see real application of the studied material.

The current academic year 2012/2013, the course on OOP attend students who have learned C++11 (in the preceding courses). They have more experience with classes and objects. Then CS2 course examines in depth the concepts and mechanisms of the OOP and in particular details of the implementation of inheritance (single, multiple, virtual) and polymorphism through abstract classes and virtual methods. Our idea is to make OOP an advanced C++ programming course, deeply familiarizing with details of class definition and class use and code optimization. We include more features from C++11 like inilializer-list constructors, defaulted and deleted functions, inherited constructors, in-class member initializers as well as more details on variadic templates. A significant part of the course will be selected for more thorough acquaintance with the standard library and especially with incorporating new unordered containers, algorithms, new `std::move_iterator`, which dereferences to rvalue references and new iterator increment and decrement operations. This course will provide special attention to lambda-expressions. These expressions let us define functions locally, at the place of the call, thereby eliminating much of the tedium and security risks that function objects incur. We use lambda expressions also in the introductory course mainly as predicates when working with containers and algorithms. In the course OOP we continue doing this, but now use containers which elements are objects of user-defined types:

```
sort(begin(v), end(v), [](Person& p1, Person& p2){
    return p1.get_year() < p2.get_year();});
cout << "\nSorted by age:\n";
for_each(begin(v), end(v), [](Person& x){cout << x << endl;});
```

Lambdas make the code more elegant and faster and existing STL algorithms becomes more usable. Newer C++ libraries are increasingly designed assuming lambdas as available, and some even require lambdas to use the library at all. So it is good students to know how to use them.

We have a good opportunity to make connection between parallel driving courses of Functional and Logic Programming in the fourth semester, which will in future allow to experiment and solve the same problems with the means of declarative languages and with C++ Castor library (Naik, 2010).

Since our bachelor degree programs have no special course of Object Analysis and Design, and experience shows that in order to know in detail the principles of design themes the course Software Technologies are insufficient, we include in OOP course theoretical topics related to the characteristics of the object model - finding relationships between objects and classes, planning, communication between objects, etc. We believe that this knowledge will have a significant impact on a programmer's skill.

Program verification and validation have an important place in the curriculum of Informatics and Computer Science bachelor programs (Todorova & Kanev, 2012). Three from 31 core semester hours in Software Engineering knowledge area (ACM/IEEE-CS, 2008) are devoted to these topics. This area is covered by several subjects in our curriculum, but the topics regarding verification and validation of primary are discussed for ready software systems and are somewhat isolated from the stage of programming. Therefore, this year we will try to give adequate attention to these issues in the framework of the educational model, respectively, proposed in (Todorova & Kanev, 2012).

4. Class Results

4.1 Assessment Results

Here we share some findings on mastering the use of certain language features. These findings are derived from quiz, test and examination results of students in our two major specialties studying C++11, taken in the period October 2010 – January 2013. As stated above, their curricula are not the same. Specifying the details of which is better is a subject of further and more detailed study.

Table 1. Assessment results on some C++11features

	Indices	Computer Science students (%)	Informatics students (%)	Average (%)
1	Know well copy semantics	71	70	**71**
2	Know well move semantics	63	57	**60**
3	Find that move constructor is easier	75	70	**73**
4	Find that move assignment operator is easier	25	30	**28**
5	Know well perfect forwarding	30	33	**32**
6	Overdo use of auto keyword	30	35	**33**
7	Use and write trivial lambdas	76	78	**77**

Indices under numbers 1, 2, 5, 6 and 7 show the proportion of all students that took part in the quizzes, tests and exams.

Almost all of the students who understand copy semantics deal with the move semantics too. What is not reflected in this table and is a result of qualitative methods of evaluation rather than quantitative: there are students at intermediate level of knowledge and understanding who would deal with copy semantics but when move semantics is added they get confused.

Results show that students overdo the use of auto keyword. And when they do not know the type of the variable or expression, it is more difficult to navigate in the program logic.

For revealing what is easier to write - move constructor or move assignment operator, (3 and 4 in the table), we assumed that 100 % are those students, who passed successfully assessment on move semantics.

Perfect forwarding anticipatedly appears to be the most difficult feature because it requires experience with writing templates – a common work for developers of libraries, but not for beginners. And if we add variadic templates most of the trained get confused.

Most of the students write simple lambda-expressions, but including of nontrivial types of parameters and return value make things go wrong.

4.2 Encountered Problems

Students cope relatively well with move semantics and a bit worse with perfect forwarding. Initially, we had problems with understanding the functions `std::move()` and `std::forward()` and therefore showed the activation of move semantics through explicit conversion to rvalue reference. The practice to demonstrate new features by expansion of the same classes, which so far have shown only copy semantics, happened to be successful. We see that a good example should include class data members referring to dynamic resource. User-defined "smart" array and string class with overloaded operations (+, +=, =) are suitable for this. It is necessary to show the core and implementation of move semantics in hereditary hierarchy and composition of classes. In this situation, the teacher should pay particular attention to the potential source of errors, namely, the initialization of the new object in move constructor of the derived class. This constructor must call the move constructor of its base class. If when accessed the parameter is not interpreted as an rvalue (contains no `std::move()`), instead move constructor the copy constructor of the base class would be executed. This example shows that it is important if `std::move()` or explicit conversion to rvalue is used to activate move semantics.

Implementation of a move constructor and a copy assignment operator provide the move semantics of a class. However, it is difficult for students to track how this semantics works (and does it works at all?). In some cases, it is activated automatically:

```
vector<Student> v ;
v.push_back(Student("Name Family",1993,17985,2,"Informatics"));
```

and in others it needs to be activated explicitly:

```
Student s1("Name Family",1993,17985,2,"Informatics");  //constructor
Student s2(std::move(s1));                //move constructor
```

The distributed nature of object-oriented programs always creates difficulties for students to understand exactly how their code works. To see which method performs it is necessary to execute the program step by step and even print intermediate results.

The concept of recursion is usually one of the most difficult to understand. This exerts influence on our work with variadic templates, as there recursion is a characteristic feature, even more difficult since it is compile-time recursion. Thus we decided to use easier examples, such as the following template that finds the sum of its arguments:

```
template <class T> T sum(T x){
    return x;
}

template <class TF, class... TR> TF sum(TF first, TR ... rest){
    return first+sum(rest...);
}
```

We compare this example to the implementation that was done earlier:

```
template < class T > T sum(std::initializer_list<T> args)
{
    T s=0.0;
    for (auto x:args)
        s+=x ;
    return s ;
}
```

Now we point out the difference in the calls of these templates: in the case of a variadic template it accepts in one and the same list arguments of different types, while in the case of initializer list, the arguments should be of the same type:

```
        std::cout << sum(4.25,5,6,7) << std::endl;
        std::cout << sum({4,5,6,7}) << std::endl;
```

We tried hard to explain details of lambda expressions usage - the ways to capture external variables, their parameters, etc. Their syntax is not intuitive and students wonder, for example, how to determine the type of the result of lambda. To reduce the negative effect, we decided to use type qualifiers for each lambda (the construct ->).

5. Conclusion

With the new organization of the educational content for the OOP in our work we tried to avoid frequently encountered negative tendency to focus on the syntactic features of C++, instead of developing algorithmic thinking. Focusing on the technical details gives a negative result in two directions: on the one hand prevents students to understand the algorithmic model, on the other - produces an incorrect idea of programming as a process, based on trial and error. Another well-known fact is that the intensive programming courses are difficult to students who have no previous experience, and this has a demotivating effect. On the other hand, students with some experience think they know more than in real, and cannot get rid of bad habits. Therefore our main idea is using the new features of C++11 to focus on technique of programming and problem solving, pay adequate attention to many other important aspects of the process of creating programs such as analysis, design and testing, and develop ability to properly structure programs and show the possibility to reuse code.

Programming languages evolve. User requirements lead to the implementation of new features that undoubtedly make languages more complex. New features require new programming techniques and even new paradigms, which in turn have found their place in the learning process. The skill to choose the right paradigm or the right combination of paradigms for solving the problem is very important. Reorganizing our training courses we take into account the fact that C++ is hard, “industrial” language. On the other hand, C++11 offers constructs that are easy to use. There are those that are difficult to learn, even at the intermediate level, but very important (lambda-expressions, move semantics, variadic templates). The difficulties that were detected in the process of learning do not give us reason to abandon reorganization of courses. In addition, although we do not have enough data for statistical conclusions, the results showed by students trained in the new method are not worse than that of their counterparts trained in the traditional way.

References

ACM/IEEE-CS Joint Curriculum Task Force. (2001). Computing Curricula 2001. *Journal on Educational Resources in Computing, 1*(3), Article No. 1. http://dx.doi.org/10.1145/384274.384275

ACM/AIS/IEEE-CS Joint Task Force. (2005). Computing Curricula 2005. The Overview Report. In *Proceedings of SIGCSE the 37th Technical Symposium on Computer Science Education*, ACM New York. http://dx.doi.org/10.1145/1121341.1121482

ACM/IEEE-CS. (2008). *Review Taskforce, Computer Science Curriculum 2008: An Interim Revision of CS 2001*. Technical report, ACM/IEEE CS.

Bennedsen, J., Caspersen, M., & Kölling, M. (2008). *Reflections on the Teaching of Programming. Methods and Implementations*. Springer. http://dx.doi.org/10.1007/978-3-540-77934-6

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *Proceedings of the 36th SIGCSE*

Technical Symposium on Computer Science Education. St. Louis, Missouri, USA. pp. 411-415. http://dx.doi.org/10.1145/1047344.1047480

Boyle, R., Carter, J., & Clark, M. (2002). What makes them succeed? Entry, progression and graduation in computer science. *Journal of Further & Higher Education, 26*(1), 3-18. http://dx.doi.org/10.1080/03098770120108266

Caspersen, M., & Bennedsen, J. (2007). Instructional design of a programming course: A learning theoretic approach. In *Proceedings of the Third International Workshop on Computing Education Research*. September 15-16, Atlanta, Georgia, USA. pp. 111-122. http://dx.doi.org/10.1145/1288580.1288595

Gomes, A., & Mendes, A. (2007). Learning to program – difficulties and solutions. In *Proceedings of the 2007 International Convergence on Engineering education*. Sep. 3-7, Coimbra, Portugal. Retrieved from http://ineer.org/Events/ICEE2007/papers/411.pdf

Hinnant, H., Dimov, P., & Abrahams, D. (2002). *A proposal to add move semantics support to the C++ language* (Report No. N1377=02-0035). Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm

Hinnant, H., Stroustrup, B., & Kozicki, Br. (2008). *A Brief Introduction to Rvalue References, The C++ Source*. Retrieved from http://www.artima.com/cppsource/rvalue.html

Hinnant, H. (2010). Howard's STL / Move Semantics Benchmark. Retrieved from http://cpp-next.com/archive/2010/10/howards-stl-move-semantics-benchmark/

ISO/IEC. (2011). International Standard ISO/IEC 14882:2011(E) Information technology - Programming languages - C++ (3rd ed.).

Meyer, B. (2003). The Outside-In method of teaching introductory programming, in Perspective of System Informatics. In *Proceedings of Fifth Andrei Ershov Memorial Conference*. Akademgorodok, Novosibirsk, ed. Manfred Broy and Alexandr Zamulin, Lecture Notes in Computer Science 2890. Springer-Verlag, pp 66-78.

Naik, R. (2010). Introduction to Logic Programming in C++. Retrieved from http://www.mpprogramming.com/resources/CastorTutorial.pdf

Simon, S., Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., … Tutty, J. (2006). Predictors of success in a first programming course. ACM International Conference Proceeding Series. In *Proceedings of the 8th Australian conference on Computing Education*. Hobart, Tasmania, Australia, pp. 189-196.

Todorova, M., & Kanev, K. (2012). Educational framework for verification of object-oriented programs (pp. 23-27). In *Proceedings of the 2012 Joint International Conference on Human-Centered Computer Environments*. New York: ACM.