Informatics in Education, 2013, Vol. 12, No. 1, 59–79
© 2013 *Vilnius University*

Experience in Teaching C++11 within the Undergraduate Informatics Curriculum

Ivaylo DONCHEV
Department of Information Technologies, Faculty of Mathematics and Informatics
University of Veliko Turnovo
G. Kozarev str. 3, 5000 Veliko Turnovo, Bulgaria
e-mail: i.donchev@abv.bg

Received: June 2012

Abstract. C++ is the most commonly used language in introductory and intermediate programming courses in Bulgarian universities. In recent years this language has developed greatly. Its abstractions are more flexible and affordable than ever before. Such great number of changes are related to the launch of the new standard (known as C++11) that we have grounds to consider it even a new language. It is inevitable to reflect all these changes in training courses and this prompted us to consider not only some updating of academic curricula but also a comprehensive reorganization of our programming courses. So, in this article we share our successes and difficulties in this direction.
Keywords: C++11, teaching, programming, introductory, intermediate, multiparadigm, pedagogy.

1. Introduction

Programming is an essential skill that must be mastered by anyone studying computer science (ACM/IEEE-CS, 2001, p. 22). This is so since there is a strict correlation between the programming skills and many other important computer skills in the field of abstraction, conceptualization, design and assessment of software merits. Programming is a creative activity providing us with a way to express abstract ideas. This makes it something more than a vocational skill. Through programming students acquire numerous skills crucial for all professions – critical and analytical thinking, paying attention to detail, etc. Among the 10 elements that, according to the overview report CC2005 (ACM/AIS/IEEE-CS, 2006, p. 35), any reputable computing degree program should include, the role of programming is consequential – the "essential and foundational underpinnings of its discipline" are pointed out in the first place, and immediately thereafter – "a foundation in the concepts and skills of computer programming". The study of programming includes mastering both the technologies and their scientific basis. Technologies include tools, techniques and standards enabling us to program. The scientific basis comprises the vast theoretical matter by means of which one can grasp programming. This basis should also be practical in order to explain the technologies and to render them applicable for the practicing programmer.

Teaching programming is traditionally accompanied by a number of difficulties. Despite the experience gained in more than 50 years, it is still considered quite a challenge (Caspersen and Bennedsen, 2007, p. 111), especially in terms of introductory courses (Meyer, 2003). Many researchers refer to the learning of programming as an extremely difficult activity (Bergin and Reilly, 2005; Boyle *et al.*, 2002; Gomes and Mendes, 2007; Simon *et al.*, 2006). The reasons for this situation are deeply rooted; there is still no consensus in the field even about the nature of the process of programming, let alone how to teach it. One thing is beyond doubt – that programming is a really complex mixture of art and science, hard to do and even harder to teach (Bennedsen *et al.*, 2008).

In this context, the main objectives pursued by this article can be defined as follows:

1. To identify weaknesses of the model, in which introductory programming courses in Bulgaria and in particular those at the University of Veliko Turnovo are organized.

2. Given the current educational requirements and rapid development of programming languages, to propose a new approach about how to build and what contents to include in the introductory programming courses.

3. To analyze the new features of C++ in terms of their applicability in the teaching process – both for the facilitation of beginners and for improving the quality of software which future programmers will produce.

4. To describe our positive and negative experiences in applying the new C++11 features and to give some appropriate examples and recommendations for teachers.

C++ is the programming language most frequently used in introductory and intermediate courses at Bulgarian universities. This choice is not random – C++ is a hybrid language and allows the software developer to make use of the advantages of several programming paradigms. This, on the one hand, gives freedom to teachers in selecting a teaching methodology; and, on the other hand, the requirement for all students to master at least two programming paradigms is met at an early stage of their studies (ACM/IEEE CS2008, 2008, p. 19). However, C++ is undoubtedly a difficult language for beginners. Its vast possibilities go hand in hand with a complex syntax. Therefore, taking into account the difficulties our students are faced with, the major development in C++ in recent years, as well as the modern requirements for safe, secure, effective and reliable programming, we have commenced a gradual reorganization of CS1 and CS2 programming courses at Veliko Turnovo University to facilitate beginners as much as possible without affecting the quality of education. The main impetus for this change was the new standard (ISO/IEC, 2011) for the C++ language. It features such important changes that it was inevitable to have them reflected in the curricula sooner or later.

2. On the Programming Courses at Veliko Turnovo University

The introductory programming course for the majors “Informatics” and “Computer Science” at Veliko Turnovo University is called “Programming Fundamentals”. It is built according to the classic imperative-first approach based on the C++ language. This approach introduces students to programming tracing the historical development of programming paradigms and languages – starting with the basic concepts of imperative and procedural programming, then moving on to objects. The topics studied within the introductory course include data types, control structures, functions, arrays, files, as well as techniques for program testing and debugging. The transition from procedural to object-oriented programming (paradigm shift) is the most serious challenge here. To "soften up" this transition, teachers use object-oriented languages to a great extent. Nevertheless, the introductory course accentuates the imperative aspects of the language – expressions, control structures, functions and other elements of the traditional procedural model. The techniques of object-oriented programming and design are left for a later year in the studies.

There are implementations of the imperative-first approach within two or three semesters. CC2001 (ACM/IEEE-CS, 2001) gives preference to the two-semester implementation through the courses CS111I – "Introduction to Programming", and CS112I – "Data Abstraction". The former course introduces the concepts of procedural programming, whereas the latter builds on these with the functionalities for programming in object-oriented style. This option is employed at our University with the major "Computer Science", although with altered course designations. The option to have it implemented within three semesters is the one most frequently used in Bulgaria. It presupposes three academic courses, as designated in ACM/IEEE-CS (2001), CS101I – "Programming Fundamentals"; CS102I – "The Object-Oriented Paradigm", and CS103I – "Data Structures and Algorithms" (DSA). Its advantages come from the greater number of academic hours, thus allowing to cover more topics and to ensure that students will have mastered the fundamentals before proceeding any further. The introductory courses within our major "Informatics" are based on such a three-semester model; however, the order of CS102I and CS103I is inverted. C++ is employed in all three courses. This is the case with the two-semester model with the major "Computer Science" as well.

A typical problem with the imperative-first approach is the fact that students have less time to become acquainted with object-oriented techniques compared to the objects-first model. Mastering these techniques is a basic requirement for their training. Adding to this the numerous difficulties students face while learning to program in object-oriented style, the fact that introducing this material is postponed in time is a major drawback of the approach. Therefore those universities which prepare their curricula following the imperative-first model should, in any case, include additional study of the principles of object-oriented design in higher courses. We have accomplished this through the parallel learning of the subjects "Object-Oriented Programming" (OOP) and "Software Technologies", the latter including elements of object-oriented analysis and design (OOAD). On the other hand, it is not less important for students to be well acquainted with the traditional imperative style of programming. Its constructs are an integral part of any object-oriented language and the implementation of the methods of each class.

The declarative style of programming is covered in our functional and logical programming intermediate level courses which take place in parallel within the 4th semester. Learning other mainstream programming languages such as Java, C#, as well as the various script and markup languages, is planned to take place within intermediate and advanced level elective courses.

3. The Object-Oriented Programming Course

3.1. *Move Semantics*

The very first element of C++11 that we used was rvalue reference type. This feature was available in the development environment that we used, Microsoft Visual Studio 2010, and its inclusion into the CS2 course in OOP in the summer semester of 2010 came naturally. By means of rvalue references, move semantics is very easily implemented. We can claim that, from a performance perspective, this is the most important new feature in C++11, therefore we shall pay special attention to it. Move semantics makes it possible for compilers to replace costly object-copying operations with less expensive "state movement" operations.

It must be admitted, however, that learning OOP using C++ involves more difficulties compared to the "pure" object-oriented languages (Smalltalk, Eiffel, Java, Simula). These difficulties come from the very philosophy underlying C++ and inherited from C, for maximum code efficiency. The price to be paid is that the programmer needs to take care himself for quite a number of matters, instead of relying entirely on the compiler. One such matter which has long been included in the OOP training course is the implementation of the methods of the so-called "Big Three" (Koenig and Moo, 2001) – destructor, copy constructor and copy assignment operator for each class, which have data members, that points to external resources or encapsulates complex data structures (the law of the big three). The way in which the latter two methods are implemented determines the long-known and successfully applied copy semantics (also called value semantics).

There are many situations where the copying of data is the correct and desired operation, but there are also other situations where doubling of resources (which is inevitable when copying) is not justifiable. Such is the case, for instance, with container reallocation, string concatenation, swap operations, etc. We illustrate such a situation to our students with the classical example of the swap of 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
}
```

When the arguments are numerals or boolean values, such a realization of `swap()` is efficient. However, if string values are being swapped (objects of `std::string`), this would imply creation of a temporary object, copying one string into it and then copying strings two more times. If the objects swapped are containers, the shortcomings of such a method become even more obvious.

The reallocation of containers, realized by means of copying, is another vivid example of inefficiency. It consists in the allocation of a new buffer in the memory, sufficient to hold the existing elements in the container as well as the new ones; call of a copy constructor for each element of the new buffer; destruction of the original elements; and release of the memory that the original buffer occupied.

The solution is to write a code for moving resources (such as dynamic memory) from one object to another, i.e., implementation of move semantics. In the instance with the swap of string values, this would mean swap of internal pointers and lengths of string objects. To realize this operation it is necessary to write a special function. To avoid doing this for each separate instance, it was first proposed as early as in 2002 to introduce new elements of the language allowing direct coding and universal usage of move semantics (Hinnant *et al.*, 2002). The means to do so are provided the by new reference type, called rvalue reference, by means of which it is possible to transfer resources from temporary objects. This has been available in the C++10 versions of popular compliers since before completion of the process of standardization of C++11. Combined with the potential of the old reference type (lvalue reference), this is all one might need for easy coding of move semantics (Hinnant *et al.*, 2008). The latter is functional precisely because it is possible to transfer resources from temporaries, for which there is no way to be referenced at another point in the program.

The development of the language is also accompanied by changes in the STL library. Even in its older versions, operations with containers and algorithms are optimized for the use of move semantics wherever possible. As a result, efficiency can be substantially improved. Test results in Hinnant (2010) show that the manipulation of large and complex STL structures with STL algorithms is more than 12 times more efficient with activated move semantics.

Copy semantics is well covered within the OOP training course in C++. This is so because one of the main learning objectives in programming in general is for the students (some of them future professional programmers) to be able to write reliable, secure and efficient code (ACM/IEEE CS2008, 2008) – which is impossible without implementation of the methods realizing copy semantics. For the same reason, one cannot ignore move semantics, either. Though students are not specifically trained how to implement it, they actually apply it using STL structures and algorithms. As early as in the introductory course, when learning strings, with the swap of two strings by the statement `std::swap(s1,s2)`, students actually use the move version of this function. Future professionals should know what the underlying mechanism is and how to design their classes so as to make use of its advantages.

In order to add move semantics to a particular class, it is necessary to define move constructor and move assignment operator for it. In some cases (ISO/IEC, 2011, pp. 278–285), the complier does so on its own, by adding implicit versions of these methods. In both situations the result is that copy and assign operations, which sources are rvalues, will automatically utilize the advantages of move semantics.

A simple example that we give our students to make it easier for them to grasp the new syntactic structures is a class containing a single data member which is a pointer to memory buffer:

```
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. Resetting is important since it prevents the destructor from freeing resources (such as dynamic memory) multiple times. The use of `std::move()` in this case is not mandatory (the pointer of integral type is copied), but we encourage it so that the students may become used to using it for activation of move semantics.

3.2. *Perfect Forwarding*

The rvalue reference type can also be used to achieve perfect forwarding, a problem of generic programming in C++ heretofore unsolved by other means. Perfect forwarding is an important C++11 technique. It allows move semantics to be automatically applied, even when the source and the destination of a move are separated by intervening function calls. The idea is for the function template to be able to pass its arguments through to another function whilst retaining the lvalue/rvalue nature of the function arguments by using `std::forward()`. It avoids excessive copying, and also frees the template author having to write multiple overloads for lvalue and rvalue references.

Common examples include constructors and setter functions that forward arguments they receive to the data members of the class they are initializing or setting, as well as standard library functions like `make_shared()`, which "perfect-forwards" its arguments to the class constructor of whatever object the to-be-created `shared_ptr` is to point to. We introduce students to this set of problems through a classical example of a generic factory function that returns a `shared_ptr` for a newly constructed generic type. Factory functions such as this are valuable for encapsulating and localizing the allocation of resources. They must accept exactly the same sets of arguments as the constructors of the type of objects constructed. We show the drawbacks of realizing it by the tools available before C++11 and, following an argumentation similar to the one in Hinnant *et al.* (2008), we come to the modern solution:

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

Another example suitable for training 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. Thus, when constructing objects, move semantics will be applied when possible.

3.3. *Further Development of the Course*

In 2010 a novelty in the OOP course was only the implementation of move semantics and perfect forwarding, through rvalue references. In 2011 we added three new C++11 features, also available in VC10 – type deduction from initializer (`auto`), `decltype` and the new suffix return type syntax. The following example combines the use of all three:

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

The return type of template function `sum()` is defined correctly with various types of arguments.

Since 2012, we have been focusing on variadic templates as well, since these can greatly simplify the writing of type-safe code with variable numbers of arguments. In the training course we show the classical examples of general type-safe `printf()` function and simple tuple class, as well as several examples of our own.

During the last couple of years our students would choose to attend the elective Graph Algorithms course, which is held parallel to the OOP course in the 4th semester. In this elective course, C++11 is also used, which is a great advantage since students gain more experience and witness some actual application of what they have learned.

From next academic year students who have been taught using the new approach and who have mastered quite a few of the C++11 techniques will enter the OOP course. They will be more experienced with classes and objects. In such case it will be left for the CS2 course to examine in depth the object-oriented concepts and mechanisms – mainly, the details of implementing the inheritance (single, multiple, virtual) and polymorphism through abstract classes and virtual methods. Our idea is to turn the OOP course into an advanced C++ programming course, teaching in depth the fine points of defining and using classes and code optimization. Of the C++11 elements, we will also include here initializer-list constructors, defaulted and deleted functions, inherited constructors, in-class member initializers, as well as more details regarding variadic templates. A great part of the course will be dedicated to a more thorough familiarization with the standard library, and more specifically, the new unordered containers, algorithms, included therein, as well as the new `std::move_iterator`, which dereferences to an rvalue reference, and the new iterator increment and decrement operations. This course will also entail a more detailed learning of the extremely important lambda expressions. We will also continue developing skills and raising the level of knowledge on concurrent programming, chiefly in the direction of sharing data between threads; detecting data races; promises, futures and async; tasks; mutexes, locks and monitors.

We have a good opportunity to form inter-disciplinary connections with the functional and logical programming courses running in parallel within the 4th semester. This enables us to experiment in the future with solving the same problems both with the tools of pure declarative languages and with C++11 and Castor library (Naik, 2010).

Since our bachelor programs lack a dedicated OOAD course, whereas practice shows that the topics within the “Software technologies” course are not sufficient for a detailed knowledge of the principles of design, we will include in the CS2 OOP course some theoretical topics related to the peculiarities of the object model – the finding of key abstractions, relationships among objects and classes, planning of interactions between objects, etc. We believe that these are substantial for acquiring programming skills.

The verification and validation of software is an important subject on the Computer Science and Informatics curricula (Todorova and Kanev, 2012). Three out of 31 core hours of the Software Engineering knowledge area are envisaged for this topic (ACM/IEEE CS2008, 2008). In our curriculum, this area is covered by several academic subjects; however, particularly those topics that are related to verification and validation concern, to a greater extent, ready-made software systems and are, in a sense, isolated from the stage of programming. Therefore we decided to emphasize these topics within the OOP course from next year on, following the educational framework model as proposed in Todorova and Kanev (2012).

Although we have a separate course in Visual Programming, we plan on including the construction of several GUI applications as well, thus introducing earlier the obligatory familiarization with event-driven programming.

4. The Programming Fundamentals Course

It is not by chance that pedagogical research is focused precisely on introductory courses. Problems there are most numerous and they are reflected in the further training and even the professional realization of students. Therefore we shall give more attention to the reorganization of our introductory Programming Fundamentals course. This course underwent the most serious change. We decided to reorganize it form classical imperative-first approach to multiparadigm programming with early exposure to objects, templates and threads.

4.1. *Rationale for Change*

Several considerations motivated us to make such a major change:

1. In contemporary higher education in computer science, only a few programming paradigms are taught and the object-oriented one is still dominant. Less attention is paid to functional and logical programming. Each of these paradigms is taught separately from the others (within a separate course) and thus students are unable to see the correlations between them and the ways to use them in conjunction, which, according to Van Roy *et al.* (2003), will subsequently affect the competence of programmers and the quality of their work. The underlying idea of Van Roy *et al.* (2003) is that programming should be taught from the viewpoint of conceptions, and not based on an individual language or paradigm. The advantage is that students have a deeper and better-grounded vision of the design of their programs, of their correctness and complexity. Some authors, for instance (Gewali and Minor, 2006), support the thesis that computer science students should be taught within more than one programming paradigm from the first year of their studies, arguing that the natural method for decomposing a problem is into sub-problems according to the necessary conceptions for their solving. For such teaching to be possible, either a multiparadigm language should be used (such as Leda, for instance, which has elements of a functional, object-oriented and logical language), or a multitude of languages should be taught that can be used together for implementation of the overall project (for instance, with Microsoft Visual Studio). In recent years, the C++ language has advanced greatly. According to its creator, Bjarne Stroustrup, its latest version C++11 (ISO/IEC, 2011) feels like a new language (Kalev, 2011). We can already define it as a multiparadigm language, since it directly supports procedural, object-oriented, generic, concurrent and functional (template metaprogramming, lambda expressions, tuples, etc.) programming. By means of Castor library (Naik, 2010), we can even add logical programming to these paradigms. So it is entirely possible to learn concepts of more than two paradigms within the introductory course, without having to use several languages or an unpopular multiparadigm language such as Leda or Oz.

2. The Interim Revision of CS 2001 states that, in the "Programming Fundamentals" area of knowledge, it is important "to address security concerns, so that students become aware of the need to program in such a way that they do not create security loopholes" (ACM/IEEE CS2008, 2008, p. 17). C++11 is definitely more secure than its previous versions. There are security-related features that can easily be taught to beginners without overtaxing them. Such are, for instance, uniform initialization which prevents narrowing conversions, new null pointer literal nullptr and strongly typed enumerations.

3. In the OOP course we find the typical design mistakes characteristic of those students who had been taught employing the imperative-first approach, chiefly a drive towards centralized control and a tendency for the whole functionality of the program to be unified into one class, ignoring the use of OOP's key mechanisms – inheritance and polymorphism. To deal with these problems, we deemed it wiser to familiarize our students at an earlier stage if not with the design, at least with the use of classes, objects and STL templates.

4. Application of the imperative-first approach invariably leads to heightened attention to language syntax at the expense of the programming problem solving technique. We believe that, in the introductory programming class, students need to master the capacity of the language to program at a high level first, in order not to lose the thread of problem solving. This is greatly facilitated by the new C++11 features.

5. Some colleagues consider C++ a rather complex language for beginners, therefore it is necessary to justify our choice to stick to this language for the introductory course instead of replacing it with the popular Java, C# or any modern script language:

C++ is commonly learned at Bulgarian universities as a first language. This is the case at some prominent departments, too, such as the University of Cambridge Engineering Department (CUED). Until a few years ago, students with substantial background in mathematics came to the universities from the specialized secondary schools, with well-developed capacity for abstract thinking. They had some experience of learning programming seriously, as a part of their specialized training. The syntax of C++ is not a problem for such students (not necessarily meaning that they have no problems at all). In recent years the reforms in secondary and higher education resulted in the fact that quite a few students studying computer majors come from humanitarian, language and even sports schools. Such students do not have any experience with programming, as a rule, but it is our belief that they are precisely the ones to be facilitated by some new features of C++11 (for instance, deducing the type of variable from its initializer expression, uniform initialization, range-based for loop). Even if they do not have any preliminary training, our students are expected to at least have the mindset to take in abstract notions.

The goals of training in the majors "Informatics" and "Computer Science" include preparing students for their realization as programmers, and C++ is a language widely used in the branch; there is no evidence that this situation is about to change soon. This language is more flexible than other languages because it can be used to create a wide range of apps – from fun and exciting games, to high-performance scientific software, to device drivers, embedded programs, and Windows client apps. In addition, abstraction mechanisms in C++11 are greatly improved ("abstraction" here meaning not only classes and objects).

Another important consideration to continue using C++ is that replacing it, for instance, with Java, C# or Python in the introductory course would affect many other courses and would practically call for changes in the entire curriculum of the specialty. Any change of such scale needs to pass a burdensome administrative procedure.

Last but not least, we stake on the good mastery of C++ due to the efficiency of this language. Languages like Java and C# are good when programmer productivity is important, but they show their limitations when power and performance are paramount. Nowadays, as Stroustrup points out, efficiency is not just running fast or running bigger programs, it's also running using less resources. Using less resources means less energy consumption, and this contributes to the fight against global warming.

4.2. *Characteristics of the New Qpproach*

Our approach differs from both the traditional bottom-up (imperative-first) approach and the popular objects-first approach. Although we introduce objects at an early stage, the introductory course does not teach in depth the concepts and mechanisms of OOP – these are left for the subsequent intermediate course. The difference with the imperative-first approach that we employed so far is that students already use standard-library components (`string, vector`) from day one, as is the case with Stroustrup (2008). These components are actually template classes, i.e., subject to generic and object-oriented programming. We have taken into account Stroustrup's recommendations (Stroustrup, 2009) to focus on how to solve programming problems, not language-specific details. Language features are introduced as a set of tools for problem solving. We structured the course of lectures mainly based on the textbooks (Stroustrup, 2008; Horton, 2010; Horstmann, 2010), reworking the examples so as to use all C++11 features facilitating beginners. The assignments for labs and individual work are taken chiefly from the books of problems (Todorova *et al.*, 2008b) and (Todorova *et al.*, 2008a), and we also edit the solutions so as to use with priority the STL algorithms and C++11 elements.

Although we use STL from day one, our course also presents early on built-in data types (`int, double, char, bool`), statements, expressions, imperative control structures and I/O operations. The introduction of pointers and arrays is postponed until much later, only after having introduced and used `std::array`, classes, functions, parameter passing by value and by reference. In conjunction with arrays, we also introduce tuples as one more C++11 option to unify data, borrowed from functional programming. We use STL algorithms for the solving of problems related to searching, sorting, etc. It is precisely here that we also include lambda expressions – elements of functional programming again. We also show students how some problems can be solved using the techniques of parallel programming. Detailed learning of algorithms is left for the DSA course which takes place in the second semester and also belongs to the introductory courses group. This course, being directly related to Programming Fundamentals, is also currently being reorganized and the students who have studied C++11 in the first semester now develop further their skills for using the new features.

As to the object-oriented paradigm, the emphasis is entirely on the use of ready-made classes and objects (often based on templates) and the creation of relatively simple user-defined types (classes), for grouping of data rather than for demonstration of complex behavior.

Learning function overloading we come to the creation of simple function and class templates as well (we include generic programming).

Concurrent programming is an extremely vast field. It is still considered an advanced topic, and with good reason, but by means of standard-library `<thread>` and `<future>` headers we can include some examples of parallel execution. Furthermore, there is already some experience with teaching parallel programming even to nine and ten year-olds (Gregg *et al.*, 2012) of course, not in C++ but in a specially designed language for this purpose. What we teach is fork/join (barrier) model, passing arguments to threads and sharing data between threads, working with tasks (`async()`, `future`, `promise`). Our examples include parallel search in arrays, calculation of Fibonacci numbers, operations with matrices.

Since time is not enough, and students do not have the necessary theoretical training, we demonstrate the possibilities for a logical style of programming in C++ through the Castor library only within lectures.

4.3. *C++11 Features That We Have Included in the Introductory Course*

The first C++11 element in this course that our students are confronted with is uniform initialization. C++ offers several ways of initializing an object depending on its type and the initialization context. When misused, the error can be surprising and the error messages obscure. It can be hard to remember the rules for initialization and to choose the best way. The C++11 solution is to allow {}-initializer lists for all initializations. Another advantage of uniform initialization is that it prevents narrowing conversions, i.e., conversions that would cause loss of data. However, in the course of working with the students, we found that it is not always suitable to use unified syntax. When it is a matter of initializing variables of fundamental types, syntax by assignment is quite more intuitive, for instance

```
double s{0.0};
```

looks more confusing for students than the traditional

```
double s = 0.0;
```

To avoid any misconception due to the use of the = sign, the difference between initialization and assignment needs to be explained. The uniform initialization is very suitable when working with containers. If we want to initialize, for instance, a vector of 5 integers, without an initializer list we would have to write 5 calls to the `push_back()` method. In C++11 this can be done much more easily, concisely and efficiently:

```
vector<int> v{5, 8, 13, 21, 34};
```

Another feature that we consider a great facilitation for beginners and we use from the very beginning is automatic type deduction (`auto` keyword). It is useful for two reasons. First, most obviously it's a convenience that lets us avoid repeating a type name that we already stated and the compiler already knows. If we want to traverse the vector of the above example by the means of C++98, we need to write the following:

```
for(vector<int>::const_iterator p = v.begin(); p! = v.end(); ++p)
```

whereas in C++11 this is done as follows:

```
for(auto p = begin(v); p! = end(v); ++p)
```

The other situation in which it is convenient to use auto is when the type is either hard to know exactly or hard to write. We avoid such situations in the introductory course since they are related to writing more difficult to explain to beginners templates. We confine ourselves to automatic type deduction of a variable initialized by a result from the execution of an STL algorithm, for instance:

```
auto res = all_of(begin(v), end(v), [](int x){return x > 0;});
```

It should be stressed that using `auto` doesn't change the code's meaning. The code is still statically typed, and the type of every expression is already crisp and clear; the language just no longer forces us to redundantly restate the type's name.

A C++11 feature, which is related to auto but we avoid it in the introductory course, is `decltype`. It is applicable chiefly in generic programming. If we just need the type for a variable that we are about to initialize, auto is often a simpler choice. `decltype` is more appropriate in those cases where we need the type of something that is not a variable, such as, for instance, the return type of template function.

A great facilitation for novice programmers, especially when working with containers, is range-based for loop, often defined as a Python-like feature, although nowadays almost every programming language has a convenient way to write a for loop over a range of values. Here is how we can traverse the vector from the above example with its help:

```
for(auto x:v) cout << x << '';
```

Besides STL containers, with the new range-based for loop strings, arrays, initializer lists and anything for which we define `begin()` and `end()` can be iterated over, and the values can not only be read but also changed.

One of the most exciting features of C++11 is ability to create 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 in the introductory course of study mainly as predicates when working with containers and algorithms (as in the example above). We treat raw arrays in the same way:

```
int A[10]{5, 4, -3, 2, -1, -7, 8};
int n = 0;
for_each(begin(A), end(A), [&n](int x){if(x > 0) ++n;});
sort(begin(A), end(A), [](int a, int b){return abs(a) < abs(b);});
```

We teach lambdas because they make the code more elegant and faster and make the existing STL algorithms 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 to have students get used to using them at an early stage.

Many developers have used concurrency in C++ before, but always through a third-party library – often directly exposing OS APIs. Now threads are finally part of the language (C++11). They are a convenient way to process multiple event streams, and the dominant way to take advantage of multiple cores for a single application, so we have included these in the introductory course at the stage when we implement simple algorithms on regular arrays and STL containers.

As already mentioned, learning pointers is left for a late stage of the introductory course. Initially, our idea was to completely replace traditional (raw) pointers with new C++11 smart pointers declared in the <memory> header. What attracted us was the opportunity to have an object that contains a pointer and behaves like a pointer, but automatically releases its memory when certain conditions are met. That would be a great facilitation for the beginners. However, this turned out to be ill-suited and we confined ourselves to raw pointers only. It was not an easy task to explain the difference between the three types of smart pointers and to teach students the skills to use these properly. We left smart pointers for the OOP course. The only change is that our students always use the new literal `nullptr` for denoting null pointer value, because the literal 0 or the macro NULL are ambiguous (they could be either an integer or a pointer).

Another new feature of C++11 which is directly related to uniform initialization and has also been included in our CS1 course is class template `initializer_list`. We do not use it for defining initializer-list constructors (this is also left for the OOP course), but only for functions that can assume a various number of arguments, for instance:

```
double average(initializer_list<int> args){
auto sum = 0.0; auto n = 0;
for(auto x:args){sum+ = x; ++n; }
return sum/n; }
```

This function finds the arithmetic mean of the arguments it is called by. The restriction is that they need to be of the same type (in this case, `int`).

Of the new container classes, we use a lot `std::array` and `std::forward_list`. We even introduce the concept of array by `std::array` and only then do we demonstrate regular arrays. For the new container classes, we also employ the new algorithms that mimic the set theory operations `all_of, any_of, none_of`, together with the new category of `copy_n` algorithms and minimum/maximum operations.

When teaching enumerations, we focus only on the new scoped and strongly typed enums (`enum class`), since they solve a number of problems – they do not implicitly convert to `int`; they do not export their enumerators to the surrounding scope, which prevents name clashes; the underlying type of an enum can be specified, which makes forward declarations possible.

An important new feature in C++11 are tuples – heterogeneous, fixed-size collections of values. They are generalization of pairs. Tuples have characteristics of both structs and arrays. Like structs, the tuple elements can be of different types. Like arrays, the elements can be accessed via indexing. Tuples are used whenever we want a heterogeneous list of elements at compile time but do not want to define a named class to hold them. Tuples make it much easier to return multiple values from a function. We use these in the introductory course also for unusual purposes, such as, for instance, to rotate the values of three variables:

```
tie(a, b, c) = make_tuple(b, c, a);
```

Should we do this in the conventional way, we would need 4 assignments and one additional variable.

5. The Data Structures and Algorithms Course

We started making changes to this course only recently, in the summer semester of 2011–2012 academic year, when those students who had been taught by the new approach in the Programming Fundamentals course in the previous semester entered it. We preserved the general orientation of the course towards low level implementation of structures and algorithms. We also preserved the topics examined and their sequence – starting with the concept of recursion, sorting algorithms and overview of algorithmic strategies brute-force, divide-and-conquer, backtracking and greedy, then moving on to dynamic realization of fundamental abstract data structures – stack, queue, linked lists, trees, graphs and hash tables.

When learning sorting algorithms, we used to compare the efficiency of our implementations with the library ones, `sort()` and `qsort()`. To generate series of random values with various distribution we used the features `uniform_int_distribution` and `normal_distribution`, provided by C++11 via the new header <`random`>. To measure the running times of algorithms to the nearest nanosecond we also used features of the new standard of header <`chrono`>. Surprisingly for the students, test results were almost always in favour of library realizations of algorithms; however, this motivated them to look for other options to optimize their solutions. The same applies to the realization of the studied data structures as well.

We introduced classes and objects only after the DSA course; therefore we presented the realization of abstract data structures through pure C-style structs (without member functions), global variables and raw pointers. After the reorganization of the introductory course, we now do these realizations with the means of OOP, albeit not including its fundamental mechanisms of inheritance and polymorphism. We confine ourselves to ensuring a better grasp of encapsulation and information hiding, classes and separation of behavior and implementation, as taught in the introductory course. In contrast, striving to develop students' skills of writing a safe and secure code at an earlier stage, we have put forward in the DSA course some topics of object-oriented exception handling which we previously used to dwell on in the OOP course.

All functions realizing operations with the structures are written in such a way so as to make use of the advantages of exception handling. Here we do not utilize the full possibilities of this mechanism (for instance, integrating user-defined exception classes in the exception system hierarchy) and do not explain everything in detail (e.g., stack unwinding), but we only teach students how to find critical code sections so as to place them into protected `try` sections, how to catch all errors that may arise and how to signal any "emergency situation" by throwing an exception.

Our plan is to have students of the next OOP course improve, through adding a new functionality, the classes they made for representing abstract structures. This can be achieved only after students have acquired the relevant knowledge. For example, when learning operators overloading, the class representing linked list can be expanded with operators for stream output, index and conversion to `std::vector`. Upon completion of the OOP course, when students will already have thorough knowledge of templates, containers and iterators, this class may assume the behavior of a real STL container, even better than `std::forward_list`.

6. Problems Found and Recommendations to Teachers

As was expected, the most problems occurred with the Programming Fundamentals course. The first problem that we confronted in this course was the choice of a suitable development environment and compiler. C++11 support is still at the experimental stage and there is practically no compiler providing all facilities. Currently the compiler which supports the most C++11 features is GNU GCC, but there is no suitable for beginners development environments that use it. From next year on, we hope to use a stable 2012 version of traditional for us Visual Studio.

As has already been mentioned, the background in informatics and in programming in particular, of those students who enroll in our specialties varies quite a bit. We found that those of them who had studied programming by the classical approach in secondary school would expect this to continue at the university, and they would consciously or unconsciously resist the new approach. They find it unnatural to use library containers and algorithms before having learned all imperative and procedural constructions.

With those students who had no prior training, on the other hand, a problem occurred with motivating them to write a version of their own of an algorithm for which they have already used a ready-made library function (for instance, `std::count_if`), as well as to work with raw arrays, considering that `std::array` provides the same functionality.

At the beginning, we used auto declarations almost everywhere where it was possible, but subsequently, when testing, we found that the students face difficulties with determining the type of variables, and this indicates future problems with constructing algorithms. One frequently occurring mistake was the attempt to unify the declarations of variables of different types:

```
auto n{10}, s{0.0}; //error—auto must always deduce to the same type
for(auto i = 0; i < n; ++i)
```

That is why we gave up using auto in the trivial cases at least, such as defining variables of fundamental types, as in the above example, and we prefer using explicit type declaration:

```
int n = 10;
double sum = 0.0;
for(int i = 0; i < n; ++i)
```

We wasted quite a lot of time in Programming Fundamentals class explaining all the things related to lambda expressions – 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 ->). There is no way to ignore lambda expressions in the introductory course, since they are a very important mechanism and it is good to have them introduced at an early stage; however, we distributed the matter related thereto among the three courses, the greatest part remaining for the OOP course.

One thing that we wanted to teach in the introductory course but gave up immediately was smart pointers – `shared_ptr, unique_ptr` and `weak_ptr`. All students find them rather difficult to comprehend, and they require a deeper knowledge of generic and object-oriented programming. Nevertheless, we use smart pointers within the DSA course (again, not without problems), but the major part of all details related thereto a left for the OOP course as well. Even there, it was not an easy task to explain the difference between the three types of smart pointers and to teach students the skills to use these properly. It was due to similar considerations that everything related to variadic templates was left for that course.

With regard to C++11 concurrency, our students cope relatively well. This is due to the simpler problems that they need to solve. However debugging and tracing execution of concurrent programs is problematic. Specifically we found that it is difficult for them to distinguish between the concepts "task" and "thread", and this is important for understanding the model of parallelism. Teachers need to stress the fact that asynchronous tasks are those smaller portions of the original algorithm only linked by the data they produce or consume, whereas threads are units of execution administrated by the runtime environment. The two concepts are linked in the sense that tasks are run on threads.

The least amount of changes related to C++11 were made in the DSA course; however, it has been completely reorganized in its part concerning the realization of abstract data structures (the transition from C-style structures to classes and smart pointers). It was our intention to demonstrate, as early as this year, some more up-to-date concurrent sorting algorithms and concurrent graph and tree traversals, but we were unable to prepare good realizations using only the means described in the standard (ISO/IEC, 2011) and we postponed this for next year. The main problems in this course were again the traditional ones – understanding the more complex algorithms (such as heap sort) and using pointers in recursive data structures. To cope with these problems, graphic representations and animations helped a lot. We expected that the usage of classes instead of structs would raise the level of abstraction, but this was not the case – on the contrary, after they had some experience in using ready-made classes and constructing simpler ones in the introductory course, students would only naturally adopt the idea to construct abstract data structures with the means of OOP. A specific problem that we encountered in the DSA course was related to pointers again. It concerns the ambiguous behavior demonstrated by smart pointers. Thanks to the overloaded operators *, -> and =, we use these both as raw pointers and as objects (through their member functions `get(), reset()` and `use_count())`. This proved to be confusing for some students since the subject matter concerning operator overloading is planned for the OOP course and additional clarifications were necessary.

With regard to the problems in the OOP course, students cope relatively well with move semantics and perfect forwarding. At the beginning, understanding the operation of the `std::move()` and `std::forward()` functions was a bit problematic, therefore we would demonstrate activation of move semantics by means of explicit type conversion to rvalue reference. It proved to be a successful practice to demonstrate new possibilities by expanding the same classes we had already used to demonstrate only copy semantics. A good example should include a class containing data members pointing to dynamic resources. User-defined "smart" array and string class with overloaded operations (+, + =, =) are suitable for this. Demonstrating a realization of move semantics in a hereditary hierarchy and composition of classes is also a must.

The distributed character of an object-oriented program always makes it problematic for students to understand precisely what the code they have written does. To see which method when is executed, a step-by-step running of the program and even intermediate printouts are needed.

The concept of recursion is among those traditionally problematic for the students to grasp. This was reflected in our work with variadic templates, where it is a distinctive feature. Therefore we decided to confine ourselves to less complex examples such as the following template finding 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 a similar realization performed earlier with `initializer_list` and we emphasize the difference in calling these templates, as well as the option for a variadic template to accept arguments of various types within one list.

7. Conclusion

Summarizing our experience in teaching C++11 the following conclusions can be made:

1. When reorganizing our training courses, we strive to avoid the common negative tendency for the C++ programming courses to stress the syntactic peculiarities of the language instead of developing algorithmic thinking.
2. Intensive programming courses overwhelm those students who have no previous experience and this is demotivating for them. Experienced students, on the contrary, are under the delusion that they know more than they really do and cannot overcome any bad habits they may have. Therefore our main idea was, by means of the new features C++11 offers for programming on a high level, to focus on the techniques for programming and problem solving, dedicating proper attention to other important aspects of the process of software development as well, related to analysis, design and testing.
3. Programming languages develop. Under user pressure, new features are constantly being incorporated. This invariably complicates them. New facilities presuppose new programming techniques and even new paradigms which need to find their place within the training process. The ability to choose an appropriate paradigm or an appropriate mix of paradigms for solving a given problem is essential.
4. Studying concepts of more than two programming paradigms at an early stage makes students better problem solvers. In this vein, language features have to be introduced as a set of tools for problem solving.
5. C++ remains a heavy, industrial language but it is entirely possible to teach C++11 to beginners. You only need to choose such a subset of its constructs as would not dishearten those students, and with which syntax would not distract their attention from the basic ideas; that is, we use the language as a vehicle for introducing programming concepts.
6. We believe that C++11 provides constructs that are easy enough to use. There are some, however, that are not so easy to learn even at the intermediate level, but then they are extremely important (e.g., move semantics, lambda expressions, variadic templates).
7. The teaching of modern C++ must include elements of functional programming like lambda expressions, tuples, etc. Functional programming is so important because of its relationship with other substantial paradigm – the concurrent programming.
8. Nowadays C++11 by means of standard-library `<thread>` and `<future>` headers brings rich support for threading to the language. Asynchronous tasks enable a lightweight programming model to parallelize execution. This model might suffice in particular scenarios, but if we need deeper handling and control of the execution of threads, C++11 comes with the thread class. Despite being a more complex programming model, threads offer better methods for synchronization and coordination, and therefore necessarily must be present in the curriculum.
9. Generic and object-oriented programming are so closely related in C++ that it is good idea to start teaching them simultaneously still in the introductory course. Our practice, however, shows that there are generic programming concepts, like variadic templates, which are too complex for beginners (require good understanding

of recursion) and should be studied at a later stage in the OOP course, where the advanced object-oriented concepts are thought.

10. Although there is no sufficient data for statistical conclusions, the results shown by students trained implementing the new approach were not worse than those demonstrated by their colleagues trained by the traditional approach.

References

ACM/AIS/IEEE-CS Joint Task Force for Computing Curricula 2005 (2006). Computing Curricula 2005. The Overview Report. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, ACM, New York.

ACM/IEEE CS2008 Review Taskforce (2008). Computer Science Curriculum 2008: An Interim Revision of CS 2001. Technical report, ACM/IEEE CS.

ACM/IEEE-CS Joint Curriculum Task Force (2001). Computing Curricula 2001. *Journal on Educational Resources in Computing*, 1, 3es, ACM New York.

Bennedsen, J., Caspersen, M., Kölling, M. (2008). *Reflections on the Teaching of Programming. Methods and Implementations*. Springer.

Bergin, S., Reilly, R. (2005). Programming: factors that influence success. In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, pp. 411–415.

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.

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*, Atlanta, Georgia, USA, pp. 111–122.

Gewali, L., Minor, J. (2006). Multi-paradigm approach for teaching programming. In: *Proceedings of the 2006 International Conference on Frontiers in Education: Computer Science & Computer Engineering (FECS)*, Las Vegas, Nevada, USA, pp. 141–146.

Gomes, A., Mendes, A. (2007). Learning to program – difficulties and solutions. In: *Proceedings of the 2007 International Convergence on Engineering Education*, Coimbra, Portugal.

Gregg, C., Tychonievich, L., Cohoon, J., Hazelwood, K. (2012). EcoSim: a language and experience teaching parallel programming in elementary school. In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, ACM, New York, pp. 51–56.

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

Hinnant, H., Dimov, P. Abrahams. D. (2002). A proposal to add move semantics support to the C++ language. Technical Report N1377=02-0035, ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming language C++.
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm`.

Hinnant, H., Stroustrup, Bj., Kozicki, Br. (2008). *A Brief Introduction to Rvalue References*. The C++ Source (a web publication).
`http://www.artima.com/cppsource/rvalue.html`.

Horstmann, C. (2010). *C++ for Everyone*, 2nd edn., Wiley.

Horton, I. (2010). *Ivor Horton's Beginning Visual C++*. Wrox.

ISO/IEC (2011). International Standard ISO/IEC 14882:2011(E). *Information Technology – Programming Languages – C++*, 3rd edn.

Kalev, D. (2011). *The Biggest Changes in C++11 (and Why You Should Care)*. Software Quality Connection.
`http://www.softwarequalityconnection.com/2011/06/the-biggest-changes-in-c11-and-why-you-should-care`.

Koenig, A., Moo, B. (2001). C++ made easier: the rule of three. *Dr. Dobb's Journal*.
`http://www.drdobbs.com/cpp/184401400`.

Meyer, B. (2003). The outside-in method of teaching introductory programming, in perspective of system informatics. In: Broy, M., Zamulin, A. In: *Proceedings of Fifth Andrei Ershov Memorial Conference*, Akademgorodok, Novosibirsk. *Lecture Notes in Computer Science*, 2890, Springer-Verlag, pp. 66–78.

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

Simon, S., Fincher, S., Robins, A., Baker, B, Box, I., Cutts, Q., de Raadt., M., Haden, P., Hamer, J., Hamilton, M., Lister, R., Petre, M., Sutton, K., Tolhurst, D., Tutty, J. (2006). Predictors of success in a first programming course. In: *ACM International Conference Proceeding Series; Proceedings of the 8th Australian conference on Computing Education*, Hobart, Tasmania, Australia, pp. 189–196.

Stroustrup, Bj. (2008). *Programming – Principles and Practice Using C++*. Addison-Wesley.

Stroustrup, Bj. (2009). Programming in an undergraduate CS curriculum. In: *Proceedings of the 14th Western Canadian Conference on Computing Education*, Burnaby, BC, Canada, pp. 82–89.

Todorova, M., Armianov, P., Georgiev, K. (2008a). *Workbook of Exercises on Programming in C++. Part Two – Object-Oriented Programming*. Techno Logica, Sofia.

Todorova, M., Armianov, P., Petkova, D., Georgiev, K. (2008b). *Workbook of Exercises on Programming in C++. Part One – Introduction in Programming*. Techno Logica, Sofia.

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

Van Roy, P., Armstrong, J., Flatt, M., Magnusson, B. (2003). The role of language paradigms in teaching programming. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ACM, New York, pp. 269–270.

I. Donchev is a graduate of St. Cyril and St. Methodius University of Veliko Turnovo, Bulgaria. He received his MCs degree in informatics in 1996 and his PhD in pedagogy of teaching informatics in 2009. Currently I. Donchev is principal lecturer at the Department of Information Technologies at the same university. His research is focused on the methodology of teaching programming and instructional design. He is a member of the John Atanasoff Society of Automatics and Informatics (SAI).

Programavimo kalbos C++11 taikymo patirtis

Ivaylo DONCHEV

C++ programavimo kalba yra viena iš daugiausiai naudojamų kalbų Bulgarijos universitetų įvadiniuose ir aukštesnio lygio programavimo kursuose. Pastaruoju metu ši kalba buvo tobulinama, todėl joje naudojamos abstrakcijos tapo dar lankstesnės ir plačiau pritaikomos. Šie pokyčiai susiję su naujo standarto C++11 atsiradimu. Neįmanoma visų atsiradusių pasikeitimų tiesiogiai perkelti į mokymo kursus, todėl buvo atnaujintos ne tik mokymo programos, bet ir peržiūrėti bei perorganizuoti programavimo kursai.