Generic Programming Approach in the Implementation of Error-Correcting Codes From Algebraic Geometry

Artem Pelenitsyn

Department of Mathematics, Mechanics and Computer Science, Southern Federal University, Rostov-on-Don, Russia

apel@sfedu.ru

Abstract

We describe design decisions adopted in our software implementation of decoder for a class of algebraic geometry codes. The decisions develop methodology of generic programming and show some metaprogramming techniques which are valuable when solving similar problems from the field of error-correcting codes or more generally computational algebraic geometry. We also emphasize features of C++11 which are to increase support for generic programming in C++ language.

Categories and Subject Descriptors D.1.0 [*PROGRAMMING TECHNIQUES*]: General

Keywords generic programming, metaprogramming, C++, templates, error-correcting codes, bms-algorithm, multivariate polynomials

Introduction

The constructions of error-correcting codes (ECC) form algebraic geometry (AG) and their decoding algorithms [1] involve a number of abstract mathematical notions, which are considerably more difficult subject for software implementation than in the case of classical ECC. Taking into account the fact that the open implementations of AG-codecs are not easily found [2], [3], experience in their implementation is of particular interest. An increase in number of software libraries of the type should serve to the accumulation of the experimental data in the field of ECC.

Besides the interest from the ECC theory point of view programming solutions of a various algebraic problems (including those, originated from ECC field) rises particular issues in software engineering. One notable matter here is the expressiveness of type systems of programming languages in use. It was observed (e.g. [4, 3.1]) that the metaprogramming techniques in the C++ programming language setting allow an increase in the expressiveness of program code comparatively to widely adopted basic object-oriented and all the more procedural techniques.

Usually in algebraic problems we have to deal with some abstract "structures" which accept various domains. Consider an implementation of finite field arithmetic. It is usually differ in cases of prime fields and extensions of those. However the difference is completely insignificant to the arithmetic algorithms in finite fields and should be hidden behind (generic) interfaces, which allow to avoid code duplication preserving maximal performance. This is the core principle of generic programming formulated by Stepanov and Musser [5]. In the series of works it was noted that solution of tasks from linear algebra and graph theory using the principle could be implemented on a basis of C++ template feature [6] [7] [8]. Our paper could be regarded as a continuation of the above mentioned research but in the setting of computational algebraic geometry. [9]

The paper consists of introduction, two sections and conclusion. In the first section we give a survey of our implementation of a decoder for a class of ECC from AG. It clarifies the role of the particular metaprogramming and generic programming techniques which are described in some detail in the second section. First section has four subsections which correspond to the main units of the library described. Second section also have four subsections each of which devoted to a single technique exploited in the library.

1. A survey of implementation

The library discussed implements multivariate polynomials arithmetic to the extent needed for implementation of BMS-algorithm which constitute the main part of decoding algorithm, and those algorithms as they described in [9, chap. 10]. The announcement of the implementation of BMS-algorithm only was made [10], and in current paper we focus on the implementation details which develop generic programming methodology and metaprogramming techniques.

We used C++ programming language including some features added in its new standard (so called C++11) [11], but no compiler-specific extensions, so we believe that the implementation is rather portable. For testing purposes we compiled the source code with GCC 4.6 under GNU/Linux operating system. Taking into account that current version of Visual C++ compiler has notable support for C++11, there shouldn't be any issues in building the library under Windows family of operating systems.

The library is "headers-only" as it is usually the case with template-based C++ source code. The most important class templates which holds the major part of functionality are:

- `Point<N, OrderPolicy>`,
- `Polynomial<T>`,
- `BMSAlgorithm<SeqT, PolynomialT, OrderPolicy>`,
- `BMSDecoding<N, ECCodeParams>`

(a portion of template parameters here and below omitted for short). Besides, there are a number of supporting class and function templates. All in all the implementation take up about four thousand lines of code (including documentation inside source code). We do not stick to pure object-oriented design and use free functions following recommendations [12] which concern C++ as a multi-

paradigm language. Moreover, object-oriented approach is used to the limited extent — just as a tool for support modularity; inheritance is used only in several cases for implementation generic programming idioms; we do not use dynamic (based on virtual functions) polymorphism.

1.1 `Point<N>` class template

Class template `Point<N>` models a point in the N-dimensional integer lattice ($\mathbb{Z}^n$ for short). It is light wrapper around `std::array<N, int>` from C++11 which in turn incapsulates plain C array adding STL-compatible interface. The access to the sole point coordinates is done through the usual subscript operator. An important part of the `Point<N>` interface constitute a number of functions implementing various ordering on the lattice, namely `byCoordinateLess` and `totalLess` (and its synonym `operator<`). The former (free function) implements natural partial order (each coordinate of a point is less then or equal to the corresponding coordinate of the other point). The latter (member function) defines so called monomial order [9, p. 7], which itself is a particular kind of total order — this allows keeping lattice points in associative containers of C++ standard library. Also, there is a certain amount of free functions implementing a search for extremums in a point collections following various orderings, a search for point in a point collection which is less then a given point and so on. All these are necessary on the different stages of BMS-algorithm.

An implementation of `totalLess` as a member function (unlike other functions concerned with the orderings) implied by the intention to parametrize point type with monomial ordering. This is reflected in the second `Point` template parameter, `OrderPolicy`, which will be discussed more thoroughly in section 2.

1.2 `Polynomial<T>` class template

Template class `Polynomial<T>` represents univariate polynomial type, which type parameter `T` stands for the type of coefficients of those polynomials. Following fact yields an ability to get multivariate polynomials from `Polynomial<T>` type: polynomial from e.g. two variables is indistinguishable from a univariate polynomial which coefficients are themselves univariate polynomials (cf. [13, chap. IV, §1, 5]); the type for those polynomials is `Polynomial< Polynomial< T > >`.

The convenience template class `MVPolyType<n, T>` is introduced in order to facilitate the creation and usage of multivariate polynomials. First template parameter stands for a number of variables of desired polynomial type, second one is responsible for a coefficient type of those polynomials. For instance next to lines of code declares variables of the same type models polynomials from three variables with integer coefficients:

```
Polynomial< Polynomial< Polynomial<int> > > p;
MVPolyType<3, int>::type q;
```

Polynomials could be created from a special string representation built from a list of coefficients separated with a space in increasing degree order inside the square brackets. For instance, ``[2 0 1]'' corresponds to the polynomial $x^2 + 2$. Given the fact that multivariate polynomial is just an univariate polynomial with coefficients which themselves are polynomials we receive a kind of nested bracket structure, e.g. ``[[2 0 2] [1] [0 3]]'' corresponds to a polynomial $(2y^2 + 2) + x + x^2(0 + 3y)$ which is just $3x^2y+2y^2+x+2$ (accurate within possible renaming of variables).

There are a number of overloaded operators for polynomial type which are needed by decoding algorithm and in particular BMS-algorithm. They are the addition, multiplication by a scalar, multiplication by a monomial (`operator<<`) and evaluation of a polynomial for some values of variables. That is, the library holds all basic operations besides multiplication of two polynomials. Let us examine multiplication by a monomial. The following sensible convention is that monomial is given by its degree which in turn is represented as an instance of `Point<N>`.

```
Point<2> mon;
mon[0] = 3; mon[1] = 2;
p <<= mon;
```

It is worth noting that due to the static type checking here if `p` is not a polynomial from two variables, we will get compile time error.

Aggregate initialization delivers example of how C++11 facilitates solution of some typical tasks. Equivalent to the above but more concise source code using it is as follows.

```
Point<2> mon {3, 2};
p <<= mon;
```

In order to use such a syntax we have to define `Point` constructor with a single parameter of the type `std::initializer_list<int>`.

There is no "best choise" for definition of degree of multivariate polynomial. Some monomial order should be given for correct execution of BMS-algorithm and this allows to define degree. However degree processing is done quite independently from polynomials themselves. All these suggested the following decision: a polynomial degree (whatever we mean by this) is not kept inside a polynomial object; if this kind of data is necessary for a client he should keep and renew it himself.

1.3 `BMSAlgorithm` class template

Class template `BMSAlgorithm<PolynomialT>` is parameterized by a polynomial type so our implementation of BMS-algorithm is able to work with polynomials with various coefficient types and in particular with various number of variables. It could even deal with foreign polynomial types provided they obey implicit interface (or "concept") which is used by our implementation of BMS-algorithm.

1.4 `BMSDecoding` class template

`BMSDecoding` type interface is rather succinct: its constructor takes integer parameter l of code C_l [9, chap. 10], the only one public member function `decode` takes message comes from a noisy channel, which has type `BMSDecoding::FieldElemsCollection` with STL-compatible interface (in current implementation it is just a synonym for `std::vector`). The function returns a vector representing initial message.

2. Particular generic and metaprogramming techniques used

2.1 Recursive instantiations

The most important design decision of our library rests upon C++ template facility is recursive instantiation of `Polynomial` class template delivering multivariate polynomials. The idea of recursive instantiations is well-known design pattern [14, chap. 21], but the approaches to processing such kind of structures is still to be developed.

When recursive structure occurs, the way of its traversal is to be questioned. It could demand application of more or less sophisticated template techniques in different circumstances. We explore some of them.

Our examples below include arithmetic operators on polynomials. In these cases our implementation concerns only assignment versions of those operators due to the usage of canonical form suggested in [12, Item 27].

When template classes designed for recursive instantiations are created some efforts to facilitate their usage are usually put. In our case `MVPolyType<n,T>` class template is introduced. Its definition

gives sensible impression of a usual metaprogramming technique used in such setting.

```
template<int VarCnt, typename Coef>
struct MVPolyType {
    typedef Polynomial< // "recursive call":
        typename MVPolyType<VarCnt - 1, Coef>::
            type > type;
};
template<typename Coef>
struct MVPolyType<1, Coef> {
    typedef Polynomial<Coef> type;
};
```

It could be easily seen that recursion moves on the first template parameter, the desired number of variables. For univariate polynomials template specialization is defined, this is responsible for recursion shutdown. Type recursion is not really differ from data recursion used in basic recursive algorithms. The result of such a kind of recursion is a new type not new data. Such kind of computations done at compile time is said to be metaprogramming.

Just the same approach allows to define a constant `VAR_CNT` inside `Polynomial` which indicates the number of variables of the particular `Polynomial` instantiation. As well as the case with type synonym `CoefT` which stands for a multivariate polynomial coefficient type. For instance,

```
MVPolyType<3, int>::type::VAR_CNT
```

is equal to three and

```
MVPolyType<3, int>::type::CoefT
```

is `int`.

The code above concerns with the construction and analysis of polynomial type. Next we move to the processing of objects of the type.

Scalar multiplication implementation doesn't require any additional efforts implied by the recursive instantiation technique.

```
Polynomial operator*=(CoefT const & c) {
    for (typename StorageT::iterator it =
            data.begin(); it != data.end();
            ++it) {
        (*it) *= c;
    }
    return *this;
}
```

`Polynomial<T>` type is just a special container for `T` and it is implemented on a basis of one of standard container types which is hidden behind type synonym `StorageT`, and corresponding class field is called `data`.

Scalar multiplication is implemented as `operator*=` for polynomial type with one parameter of type `CoefT`. Multiplication is done by the call `operator*=` for each element of `data` container. If we are in the setting of one variable, then we directly call `operator*=` of coefficients type (which we assume have one). In the multivariate case (`data` holds `Polynomial<T'>` for some `T'`), we face yet another instantiation of the same operator but for polynomials which have one variable less. Note that this recursive strategy is completely embodied in the code above.

Polynomial addition implementation doesn't require any additional efforts concerned with recursive instantiation, as well. We deduce that this kind of traversal of either single (scalar multiplication) or several (addition) objects of one recursively instantiated type is rather handy. However, working with multivariate polynomials we sometimes have to simultaneously traverse objects of two different (in some sense) "recursive" types which have the same level of nestedness. Some examples of this in our library include polynomial multiplication by a monomial and addressing coefficient by a multiindex (in other words: by a multidegree of monomial which holds addressing coefficient). In either case we have to traverse an object of nested polynomial type simultaneously with an object of type `Point<N>` where N is equal to the level of nestedness of polynomial type (or to the number of variables — following our main convention).

Consider subscript operator implementation which allows us to get specific coefficient of polynomial. In particular univariate case we would like to have a version of subscript operator which accepts only one plain `int` parameter. So we implement this version in the library. Addressing coefficient of multivariate polynomial is done by the other version of the same operator, namely: `Polynomial::operator[](Point<VAR_CNT>)` (it can be seen right now that it is verified at compile time that the dimension of a point is equal to the number of variables in polynomial). Let us discuss the way the recursion should move on in this case.

Assume we have a point `p` of a type `Point<VAR_CNT>`. In multivariate case we should address `p[0]`-th element of `data` container (it is a polynomial which have one variable less) and call `operator[]` yet again passing as a parameter some kind of slice of point `p` which holds all the components of `p` except `p[0]`. We introduce template class `Slice<Dim, Offset>` exactly for this purpose. It holds a reference to the initial point and its type "remembers" an offset which should be applied when the slice is asked for it's i-th component. Using this template we would write something like `(data[p[0]])[make_slice(p)]` for implementing the logic described above. (We certainly would have to add private member function `operator[](Slice<D, 0>)`.) However we can't carry out this rather straightforward approach due to one C++ limitation.

Template specialization is the mechanism which is used to shutdown "template recursion" (probably, metarecursion is the right term here), cf. implementation of `MVPolyType`. In our case we would have to create specialization for member function like this one: `operator[](Slice<Dim, Dim-1>)`. Though there is a limitation in C++ language [14, 12.3.3] which forbids to specialize template members of class templates (which the `operator[](Slice<Dim, Offset>)` is). Here we introduce yet another level of indirection. Instead of writing `(data[p[0]])[make_slice(p)]` we do `apply_subscript(data[p[0]], make_slice(p))` which either move recursion on or halts it by calling `operator[](int)`.

One more remark here is that it is convenient when the call to "non-existent" polynomial coefficient returns some equivalent of zero. The exact approach adopted here uses traits idea which is discussed in the next subsection. Resulting code for a subscript operator is as follows.

```
template<typename T, typename S, typename Pt>
T applySubscript(S const & el, Pt const & pt) {
    return el[pt]; // operator[](Slice) call
}
// recursion shutdown:
template<typename T, typename S, int Dim>
T applySubscript(S const & el,
                 Slice<Dim, Dim-1> const & pt {
    return el[pt[0]]; // operator[](int) call
}
template<typename T>
typename Polynomial<T>::CoefT
Polynomial<T>::operator[](
      Point<VAR_CNT> const & pt) const {
  if (pt[0] < 0 || data.size() <= pt[0])
      return CoefficientTraits<CoefT>::addId();
  else
      return
        applySubscript<Polynomial<T>::CoefT>(
            data[pt[0]],
            make_slice(pt));
```

```
}
```

Source code for `operator[](Slice)` is not shown here because it duplicates the one from `operator[](Point)` and `operator[](int)` just returns `data[pt]` instead of call to `applySubscript`.

The whole source code is of considerable length and complexity. Some other operations like monomial multiplication and evaluation of polynomial for some values of variables are implemented in the same way. It would be valuable to build some kind of generic wrapper for such kind of simultaneous traversal which would reduce boilerplate. Unfortunately we didn't come up with a solution for this yet.

2.2 Type traits and the problem of code duplication

Class template `CoefficientTraits` mentioned above is an example of type traits class [14, chap. 15] [15, 2.10]. In our case the class supplies necessary information about coefficient type to the polynomial type: additive and multiplicative identity, additive and multiplicative inverse and some others.

The traits technique is based (yet again) on template specialization mechanism. For instance, if polynomial type client instantiates it with type `Bar` for a coefficient type, and its zero (additive identity) value is kept in a static field `Bar::ZERO`, then a client should define specialization of `CoefficientTraits` with `Bar` type where `addId` member function returns `Bar::ZERO`. There is generic definition of `CoefficientTraits` which would be used in case a client doesn't provide the specialization. The definition implements default behavior (like getting additive identity through call to the constructor of `T` with no arguments).

We didn't find discussion of following issue in well known descriptions of traits technique: traits code tends to duplicate between different specializations. We describe here our solution to the one.

The library presented was tested through particular implementation of finite field arithmetic, namely the NTL library [16]. This library uses a number of different classes for implementation of different kinds of finite fields. But exact kind of a finite field is completely insignificant for the algorithms used. So we use template parameter for the polynomial coefficient types. It is indeed comfortable provided that all NTL field classes have the same implicit interface (concept of a field). But in order not to couple our library with NTL we introduce another level of indirection — `CoefficientTraits` class template which hides the usage of NTL interface in its specializations. The problem here is that basically we have to write four specializations for different classes which all have identical implementations. In order to avoid such a duplication we use enable_if and MPL libraries from the Boost collection [17].

Template series presented in enable_if library allows to trigger different template specializations of client's class and function templates under various conditions applied to template arguments. Our particular condition in this case is the membership of `CoefficientTraits` template argument to the set of NTL types. Processing collections of types ("type lists") is one of the main feature of template metaprogramming in a spirit of Alexanrescu's seminal book [15]. It was first presented in a Loki library based on ideas of [15, chap. 3] but later developed in robust Boost.MPL library ("sequence classes" [4, 5.8]).

First we define a sequence of NTL type in terms of `boost::mpl::vector`.

```
typedef boost::mpl::vector<
        NTL::GF2E, NTL::ZZ_pE,
        NTL::GF2, NTL::ZZ_p>
    NtlFieldTypes;
```

Then, in order to exploit enable_if facilities we have to introduce one extra template parameter in `CoefficientTraits`.

```
template<typename CoefT, typename Enable=void>
struct CoefficientTraits {
    /* generic implementation */
};
```

Finally, we supply the only one specialization which will trigger when `CoefficientTraits` gets any NTL type as an argument.

```
template<typename T>
struct CoefficientTraits<
        CoefT,
        typename boost::enable_if<
            boost::mpl::contains<NtlFieldTypes,
                CoefT>
            >::type > {
    // NTL types' specific implementation ...
};
```

2.3 Policy classes

Another generic programming technique, used to parameterize point type by a monomial order, is based on policy classes [15, chap. 1]. Policy is a consistent collection of actions concerned with type being designed and orthogonal to the rest of type functionality. Exact point ordering in integer lattice is a good candidate to extraction in a policy class. The main technical approach here is to add one extra template parameter and inherit from it. The scheme is as follows.

```
typename T, template <typename> class Policy =
    DefaultPolicy>
class MyClass : public Policy<T> { /* ... */ };
```

In classical work [15] public inheritance is suggested. In this way we get interface of policy class mixed in the interface of the main class (`MyClass` in the example above). In our library the inheritance is private instead, and point interface features use base class ones internally. Consider the implementation of a member function which serves for a point increment.

```
Point<Dim, OrderPolicy>&
Point<Dim, OrderPolicy>::operator++(){
    inc(data); // OrderPolicy::inc
    return *this;
}
```

`Point` type is publicly inherits from the `OrderPolicy` which have `inc` member function.

The only issue here is that policy class have to "know" some implementation details of `Point` type, namely the type of `data` container. The latter is passed as a template argument to the `OrderPolicy`. Resulting code is as follows.

```
template<
        int Dim, // point dimension
        template <typename PointImpl> class
            OrderPolicy =
            GradedAntilexMonomialOrder>
    class Point : OrderPolicy< std::array<int,
        Dim> >
{ /* ... */ };
```

2.4 C++11 and functional style

The most interesting tasks arise while designing components at a comparably low level (including those using external API like the case with NTL library). The way of implementation higher level entities which use those low level components is mostly straightforward and done in the usual STL-like style which presents one of the solid incarnation of generic programming as defined by A. Stepanov. It is worth mentioning that the invention of lambda function in the language adds a lot to facilitation of this particular style which have a big deal with functional programming world.

As was noted by B. Stroustrup in his report on HOPL [18, 4.1.4], "The STL and generic programming in general owes a ... debt to functional programming... [but C++ function objects] is not the most elegant expression of high-order ideas". So lambdas come to provide more elegant way to deal with generic programming in C++. Let us consider one example of this taken from the library presented.

In the decoding phase the one of the main parameters is the message r transmitted through the noisy channel. It is just an n-dimensional vector over some finite field. Define syndrom as a map which given the monomial m yields an inner product of r and a vector, which is the result of evaluation m at n points of some algebraic curve (not to be confused with integer lattice points discussed above). Syndrom sequence is the result of calculating syndrom at a collection of monomials $\{m_i\}$ which constitute a basis in some important linear space. The code calculating syndrom sequence is as follows.

```
using namespace std::placeholders;
auto syndromComponentAtBasisElem =
  // declare lambda function:
  [this,&received](BasisElem const & be) ->
      typename SyndromeType::value_type
  {
    auto tit =
      boost::make_transform_iterator(
        this->curvePoints.begin(),
        std::bind(
          computeMonomAtPoint<Field, BasisElem,
          CurvePoint>,
          be,
          _1));
    return typename SyndromeType::value_type(
      be,
      std::inner_product(
        received.begin(), received.end(),
        tit, FieldElemTraits<Field>::addId()));
  };
std::transform(basis.begin(), basis.end(),
    std::inserter(syn, syn.begin()),
    syndromComponentAtBasisElem);
```

We import the names from the standard (since C++11) namespace `std::placeholders`, this allows us to use expressions like `_1` when `binding` function parameters (partial application is yet another feature of functional programming). Then it come lambda function definition. The reference to the function is kept in a variable `syndromComponentAtBasisElem` for later usage. Note that its type is left implicit through the `auto` keyword. The lambda function takes one argument of type `BasisElement` which is just a monomial and return a pair which consists of its argument `be` and syndrom value at this monomial. Transform iterator is an adaptor for its first argument, the actual iterator through curve point collection. Tha adaptor applies a function given in second argument to the elements of the underlying collection. The function here is a result of binding basis element argument (which we already have from a lambda's parameter) of a function which evaluates given basis element at a given curve point. The desired inner product is calculated with a standard algorithm as is the case with resulting syndrom sequence produced by `transform` algorithm.

At the very beginning of definition of lambda function it could be seen the list of objects which are captured by the lambda: `[this, &received]`. This capturing is a well known phenomenon in functional world which is called "closure" there. However in most functional languages explicit list of captured objects is not needed. In C++language it is a part of solution of so called upwards funarg problem [19] concerned with the following issue: captured object could possibly live longer than the stack frame of function where they were allocated. Clearly, the most challenging cases of the problem arise in the languages without garbage collection [20] like C++.

Conclusion

In the paper we tried to demonstrate application of various generic and metaprogramming techniques used for solving the problems arose in designing a codec for error-correcting codes from algebraic geometry. We haven't yet measured the performance of library implementation obtained. Yet we see further applications of generic programming techniques (e.g. expression templates) and C++11 (e.g. adding move semantics) in the library.

References

[1] Høholdt T., van Lint J.H., Pellikaan R. Algebraic geometry codes Handbook of Coding Theory, V.S.Pless, W.C. Huffman, and R.A. Brualdi, Eds. Amsterdam, The Netherlands: Elsevier, pp. 871–961. 1998.

[2] Madelung Y. Implementation of a decoding algorithm for AG-codes from the Hermitian curve, Rept. IT 93–137, Inst. Circuit Theory Telecom., Tech. Univ. of Denmark, Lyngby, Denmark, Apr. 1993.

[3] Mayevskiy A., Pelenitsyn A. Software Implementation of Algebraic-Geometry Codec using Sakata algorithm, Izvestia Yufu (Southern Federal University Bulletin), Technology Sciences, No. 8, pp. 196–198. 2008. (in Russian.)

[4] Abrahams D., Gurtovoy A.. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley. 2004.

[5] Musser D. A., Stepanov A. A. Generic Programming. Proceeding of International Symposium on Symbolic and Algebraic Computation, volume 358 of Lecture Notes in Computer Science, pp. 13–25, Rome, Italy. 1988.

[6] Siek J. G., Lumsdaine A. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In International Symposium on Computing in Object-Oriented Parallel Environments. 1998.

[7] Siek J. G., Lee L.-Q., and Lumsdaine A. The generic graph component library. In Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 399–414. ACM Press. 1999.

[8] Veldhuizen T. L. Arrays in Blitz++. Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98), vol. 1505 of Lecture Notes in Computer Science. Springer-Verlag. 1998.

[9] Cox D.A., Little J.B., OShea D.B. Using Algebraic Geometry, Second Edition. Springer. 2005.

[10] Pelenitsyn A. On Implementation of n-Dimensional BMS-algorithm Using Generic Programming. Transactions of Scientific School of I.B. Simonenko, pp. 197–203. 2010. (In Russian.)

[11] International Standards Organization: Programming Languages — C++. International Standard ISO/IEC 14882:2011.

[12] Sutter H., Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Boston: Addison-Wesley Professional. 2004.

[13] Nicolas Bourbaki. Elements of Mathematics, Algebra II, Chapters 4–7. Springer. 1990.

[14] David Vandevoorde and Nicolai M. Josuttis. C++ Templates. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2002.

[15] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2001.

[16] Shoup V. NTL: A Library for doing Number Theory. URL: `http://shoup.net/ntl/`

[17] Boost C++ Libraries. URL: `http://www.boost.org/`

[18] Stroustrup B. Evolving a language in and for the real world: C++ 1991-2006. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). ACM, New York, NY, USA, 4-1-4-59. 2007.

[19] Moses J. The Function of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem. MIT AI Memo 199. 1970.

[20] Appel A. W., Shao Z. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures // Princeton CS Tech Report TR-450-94. 1994.