From Procedural to Object-Oriented Programming (foundations, distinctions, applications, training, attractive tutorial)

Valery Sklyarov

***Resumo*- Este artigo descreve as técnicas fundamentais aplicadas na análise e na programação orientada a objectos (OOD e OOP). O artigo destaca, igualmente, as capacidades da linguagem C++, descreve algumas regras para a escrita destes programas e apresenta uma aplicação de auto-estudo.**

***Abstract* - This paper discusses the basic approaches involved in the use Object-Oriented Programming (OOP) and Object-Oriented Design (OOD) for application development. It emphasises the principal distinctions between two widely-used and well-known directions in software development. At present there are many computer languages that incorporate OOP capabilities. One of the most powerful of these languages is C++. This article underlines the main innovation introduced in C++. It recommends some rules for writing C++ object-oriented programs, and considers an animated software tutorial which allows the OOP approach in general, and C++ in particular, to be learned in the fastest possible way.**

I. INTRODUCTION

Programming languages can be considered as a tools for creating different software systems. Each language supports a particular programming technology. The complexity of developing modern software systems is increasing, and there are basic limits in the ability of a particular technology to cope with this complexity. Widely-used approaches in software development are related to procedural and modular programming. The basic idea of these approaches can be represented in the following expression:

```
PROGRAM = ALGORITHM + DATA
```

Bjarne Stroustrup defines the ideas being accepted in procedural and modular programming as follows [1]: “Decide which procedures you want; use the best algorithms you can find”, “Decide which modules you want; partition the program so that data is hidden in modules”. The languages which support procedural and modular programming are especially appropriate for tasks having algorithmic features (mathematical computations, etc.). However, because of the complexity of many tasks, especially having non computation characteristics (computer-aided design, data base, user interfaces, etc.), their effectiveness is limited [2]. This compels us to consider new approaches and new technologies in software development.

The idea of Object-Oriented Programming (OOP) was implemented for the first time in the Simula-67 language. OOP was invented as a tool for dealing with the increasing complexity of software systems. We omit a proof of this assertion and refer to [2].

The basic idea of OOP can be represented in the following expression:

```
PROGRAM = OBJECTS + MESSAGES
```

Bjarne Stroustrup defines the idea fundamental in OOP as follows [1]: “Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance”.

II. OOP FOUNDATIONS AND APPLICATIONS

Let us start with foundations of Object-Oriented Technology (OOT). The basic objective of this technology is to introduce new ideas for the design of very complicated software systems. The limitations of the human capacity for dealing with complexity is well known [2]. A good approach to overcoming these limitations is decomposition. OOT directly supports object-oriented decomposition which has many distinctions from algorithmic decomposition. It deals with a set of objects interacting to perform some unique behaviour. An object is a tangible entity encapsulating data to be manipulated and methods which usually provide an external interface.

Consider an example of an object called a register. Suppose that it is a model of real digital electronic scheme. Let us assume, that our register is 10 bits in size, and that we can perform read and write operations on it. This necessitates that we invent an object comprising the following members:

```
state  (a data member);
READ   (a method or function member);
WRITE  (a method or function member).
```

After an object has been described we can suggest a simple scenario which would be something like the following:

```
write a value N to yourself;
read a value being registered.
```

To perform these simple actions we must send the messages mentioned above to the object. It should be noted that each particular language introduces its own terminology. For example in C++ methods are called member functions, and messages are operations invoking the member functions that provide the external interface, etc.

Designing a complex digital device involves using many kinds of simple registers. Suppose the registers to be considered have different sizes. So on the one hand they are slightly different, but on the other hand they have exactly the same well-defined behaviour. We therefore need to create a class of registers. Basically, classes and objects are the main innovations of OOP. They are closely related each others and cannot be considered independently. The main difference between them is the following: an object is a concrete entity existing in time and space; a class is only an abstraction [2]. Grady Booch defines class as follows [2]: “A class is a set of objects that share a common structure and a common behaviour”. Let us consider our class named for instance REGISTER, in a bit more detail. Firstly, in order to define an object of the class REGISTER we must define an instance of the class. When we are defining an object it would be useful to make some initial settings, for example to set a size for a register and an initial state. The setting the size of a register involves memory allocation (we remember, that an object is a model of a particular register in computer memory). Another words we want:

- to allocate memory for setting the size of a particular register;
- to make an initial setting.

OOT supports these operations directly by introducing a special method (a member function) called a constructor.

A constructor is responsible for creating an object and initializing its data members. In our example it creates a particular register having a defined size SIZE, and sets it to a defined state INITIAL_STATE. In C++, the declaration of the class could be following:

```
class REGISTER
{   BOOLEAN *state;
    unsigned size;
public:
    REGISTER(unsigned SIZE,
         BOOLEAN *INITIAL_STATE);
    void WRITE(BOOLEAN *NEW_STATE);
    void READ(BOOLEAN *CURRENT_STATE);
    .........
};
```

As you can see a constructor has the same name as its class, and it returns no value (not even a void). Our constructor can be defined in the following form:

```
REGISTER::REGISTER(unsigned SIZE,
         BOOLEAN *INITIAL_STATE)
{
     size = SIZE;
     state = new BOOLEAN [SIZE];
     for(int i=0; i<SIZE;state[i++] =
          (INITIAL_STATE)?
          INITIAL_STATE[i]:0);
}
```

Here, BOOLEAN is a user-defined type representing a one dimensional boolean array (each bit of the array corresponds to a related bit of a real register), `new` is a C++ operator (and a C++ keyword) providing dynamic storage allocation. We will omit other explanations of C/C++ instructions and library functions (the reader can find them for example in [3], or in any C/C++ programming guide and library reference).

Let us consider our register in a bit more detail. In most instances it should probably be set to zero (00...0) initially. This leads us to the idea of a constructor which has so called default parameters. Consider the following constructor declaration (inside the class):

```
REGISTER(unsigned SIZE,
         BOOLEAN *INITIAL_STATE = NULL);
```

Now we can define an object in two possible ways (we have assumed that SIZE = 10):

```
REGISTER register1(10);
REGISTER register2(10,INITIAL_STATE);
```

In the first call of the constructor, the second parameter is NULL by default. In the second call the second parameter is a pointer to a particular initial state.

Once an object has been created it occupies computer memory. To free memory that was allocated we must destroy the object. To do that, the OOT provides a special method (member function) called a destructor. A destructor has the same name as the class, but preceded by a tilde (~). As we have already mentioned it destroys the object being constructed. Consider the following possible declaration and definition of a destructor for our example.

```
class REGISTER
{ ........
public:
    REGISTER(...);
    ~REGISTER(void)
    {   delete [size] state;   }
........
};
```

Here `delete` is a C++ operator (C++ key word) which provides dynamic storage deallocation. Our (and any other) destructor has no arguments.

Let`s consider our class named REGISTER further. As we mentioned above it represents a set of objects which exhibit some well-defined behaviour. A concrete object comprises data members and function members. Data members can be considered as properties of an object and therefore external access to the data members can be strongly restricted. OOT provides member access control which make it possible to hide both data members and function members within class, so that access to them from outside the class is limited. For instance, the C++ language introduces member access control attributes (key words), which are: `public`, `protected` and `private`. All members of a class are `private` by default. `Public` (`protected`, `private`) members must be declared in the public (protected, private) section of a class. The significance of the attributes is: `public` members can be used both inside and outside of a class without any restrictions; `protected` members can be used inside a class, by `friend` functions (see below) and inside derived classes (see below); `private` members can be used inside a class and by `friend` functions (`friend` is a key word of C++). Different sections (public, protected, private) can be repeated any number of times and in any sequence. In the example we considered above, we had one implicitly declared private section (by default) and one explicitly declared public section.

Let us look at an extended version of our example. Suppose we want to introduce a shift register that is a variety of register. Consider the previous REGISTER class declaration. We remember that the WRITE and the READ are member functions of class REGISTER. We need exactly the same functions for a shift register. In addition, our shift register involves a new function performing the shift operation. In other words it is worth-while to build a hierarchy of classes which might be as follows: base class named REGISTER - derived class named SHIFT_REGISTER. The concept of OOT which emphasizes a hierarchical structure is called inheritance. Basically inheritance denotes a relationship between classes. It makes it possible that one class shares the behaviour and/or structure of one or more other classes. Let us continue with our example and design a base class named REGISTER and a derived class named SHIFT_REGISTER. According to the previous explanation, the class SHIFT_REGISTER will inherit members of the class REGISTER. In addition the new class will introduce some new members, for example, a member function named SHIFT. It leads us to the following declaration:

```
class SHIFT_REGISTER : public REGISTER
// declaring the derived class SHIFT_REGISTER
// we must enumerate
// the base classes in a comma-delimited list
// followed after
// colon (:). In the example there exist a
// single base class
// named REGISTER
      { .......
      public:
         SHIFT_REGISTER(unsigned SIZE,
               BOOLEAN *INITIAL_STATE) :
            REGISTER(SIZE,INITIAL_STATE)   { }
        .......
         void SHIFT(unsigned number);
       .......
};
```

When you declare a derived class you can use the access specifier (`private`, `protected`, or `public`) in front of the class in the base list. Access specifiers allow you to alter the inherited access attributes for members of the derived class (see, for example [4]). If a base class has a constructor defined explicitly with one or more arguments, any derived class must have a constructor as well. In the example a constructor for the class SHIFT_REGISTER is declared as:

```
SHIFT_REGISTER(unsigned SIZE,
    BOOLEAN *INITIAL_STATE) :
         REGISTER(SIZE,INITIAL_STATE)   { }
```

Suppose that the READ function is exactly the same for both (base and derived) classes. However the WRITE function is slightly different for the derived class. The C++ language allows us to perform a redefinition (it is called function overriding) of a function in this case. We want to use a base class version of the function WRITE for our class REGISTER and a derived class version of the function WRITE for our class SHIFT_REGISTER. Let us consider the following definitions:

```
REGISTER reg(...), *pointer1_to_register,
          *pointer2_to_register;
SHIFT_REGISTER shift_register(...);
```

One rule in C++ says that any variable defines as a pointer to a base object (see, for instance, pointer1_to_register, pointer2_to_register) may also be used as a pointer to a derived object. The following statements will be valid in our example:

```
pointer1_to_register = &reg;
pointer2_to_register = &shift_register;
```

Now using pointer2_to_register gives access to all members of the object shift_register, inherited from the class REGISTER. Consider the following expression:

```
pointer2_to_register -> WRITE(...);
```

Which version of the WRITE will be called? If pointer2_to_register was defined as a pointer to the base class, then the WRITE member function of the base class will be called. And how can we invoke the derived class version? The answer lies in a `virtual` function

declaration (`virtual` is C++ key word). Virtual functions make it possible to resolve the overloading problem (polymorphism) during the course of program execution (it is called late binding), rather that at compile time (it is called early binding). They represent functions declared with the specifier `virtual` in a base class and subsequently redefined in one or more derived classes with their names, types of returned values and the number and types of arguments, being unchanged.

Suppose we want to add two extra classes named LEFT_SHIFT_REGISTER and RIGHT_SHIFT_REGISTER to be inherited from class SHIFT_REGISTER. Let us consider the SHIFT member function that is common to both new classes. In our representation the SHIFT_REGISTER class makes sense only as the intermediate base of classes LEFT_SHIFT_REGISTER and RIGHT_SHIFT_REGISTER derived from it. Really, registers that perform a pure shift operation do not exist, because this operation is either a left shift or a right shift. To declare operations like these, C++ provides pure virtual functions. A virtual function is made pure by setting it equal to zero. If a class contains at least one pure virtual function it is called an abstract class. It is impossible to create an object for such a class. It may only be used as a base class for the definition of derived classes which can inherit its members.

So now we could consider the following declarations:

```
class SHIFT_REGISTER : public REGISTER
{  ........
public:
  virtual void SHIFT (unsigned number) = 0;
       ........
};
class LEFT_SHIFT_REGISTER : public
     SHIFT_REGISTER
{ ........
public:
     void SHIFT (unsigned number)
     {   shifting left by number bit
positions     }
    ........
};
class RIGHT_SHIFT_REGISTER : public
     SHIFT_REGISTER
{  ........
public:
     void SHIFT (unsigned number)
     {   shifting right by number bit
positions   }
    ........
};
```

Let`s look at the statements:

```
(register state) <<= number;
(register state) >>= number;
```

Since state is a boolean array we cannot use standard >>= and <<= C/C++ operators. However the C++ language lets you redefine the actions of most standard operators. The compiler distinguishes the various functions by noting the context of the call. In other words it can check the number and types of the operands. The keyword `operator`, followed by the operator symbol, is used to define the function name. Let us assume that we want to override the operator <<=. An example of how you can do this is as follows:

```
class LEFT_SHIFT_REGISTER  :  public
     SHIFT_REGISTER
{ ........
public:
       ........
      BOOLEAN* operator <<= (unsigned number)
       {
          for(unsigned i=0;i<size-number;i++)
             state[i] = state[i+number];
          for(i=size-number;i<size;i++)
             state[i] = 0;
          return state;
       ........
       }
};
```

It should be mentioned that we intend to use the member state which is declared in the base class. To enable this member to be accessed in a derived class, it should be assigned the attribute `protected` (you must avoid the attribute `public` for data members of a class). Now the following member function will work correctly:

```
void SHIFT(unsigned number)
{   *this <<= number;   }
```

Here a new C++ pointer named `this` is invoked (`this` is C++ key word). It is a local variable of the class which does not need to be declared. It is a pointer to the object containing the function being executed and is available in the body of any nonstatic member function. So the return statement:

```
class_type* func(...)
{........
   return this;
  ........ }
```

returns a pointer (for instance &obj) to the object of type class_type, in which the function named func was declared. The return statement:

```
class_type& func(...)
{ ........
   return *this;
```

```
        ........  }
```

returns the object (for instance obj) of type class_type, in which function named func was declared. The construction type& lets us create a reference type closely related to a pointer type. In the following statements:

```
LEFT_SHIFT_REGISTER lsr(10);
LEFT_SHIFT_REGISTER &ref_lsr = lsr;
```

the lvalue ref_lsr is an alias for lsr. Any operation on ref_lsr has precisely the same effect as that operation would on lsr.

Consider the expression `*this <<= number` which appeared above. Basically, an operator function must either be a nonstatic member function, or have at least one argument of the class type. Our nonstatic class member operator function has two arguments which are:

- an implicitly defined argument, of type LEFT_SHIFT_REGISTER;
- an explicitly defined argument named number, of type unsigned integer.

The expression `*this` lets refer to an object of a class LEFT_SHIFT_REGISTER vie its implicitly defined pointer this (remember that `this` is a pointer and `*this` is an object).

Now you can use expressions:

```
object_name.SHIFT(number);
pointer_to_object->SHIFT(number);
```

Suppose you want to use another expression which will look something like the following:

```
object_name <<= number;
```

In this case you have to change the operator<<=(...) function as follows:

```
LEFT_SHIFT_REGISTER& operator <<= (unsigned
number)
{ ... (see considered above statements) ...
       return *this;
}
```

At last if you want to consider the following statement:

```
object_name << number;
```

you have to overload the predefined bitwise left shift "<<" operator:

```
void operator << (unsigned number)
{ ... (see considered above statements,
       excepting "return ... ") ...  }
```

Let us consider another possible task. We wish to compare states for two objects which have different types. It could be states of a left shift register and a right shift register. We want to invoke operations like Equal To, Not Equal To, etc. and perform them in a function named comp_two_reg. Remember that state has the attribute protected and therefore is a hidden class member. So how can we access it? The answer lies in the `friend` function declaration (`friend` is C++ key word). Using `friend` declarations, C++ provides a mechanism for access to private (protected) members of a class from functions which are not members of that class. This is enabled when the functions have the specifier `friend`. For the example we are considering, we should declare the function comp_two_reg in LEFT_SHIFT_REGISTER and RIGHT_SHIFT_REGISTER with the specifier `friend`. For the general case, the necessary statements are outlined below.

```
class LEFT_SHIFT_REGISTER : public
     SHIFT_REGISTER
{........
friend void
     comp_two_reg(LEFT_SHIFT_REGISTER *lsr,
     RIGHT_SHIFT_REGISTER *rsr);
     // we assume that our function has
     // returned value of type void
};
class RIGHT_SHIFT_REGISTER : public
     SHIFT_REGISTER
{ .........
friend void
     comp_two_reg(LEFT_SHIFT_REGISTER *lsr,
     RIGHT_SHIFT_REGISTER *rsr);
     ........
};
void comp_two_reg(LEFT_SHIFT_REGISTER
     *lsr, RIGHT_SHIFT_REGISTER *rsr)
     {if((lsr->size)!=(rsr->size))
        cout  << "states can not be compared"
              << endl;
        for(unsigned i=0;i<lsr->size;i++)
           if (lsr->state[i]!=rsr->state[i])
           {   cout << "states are not equal"
                   << endl;
               return;
           }
         cout << "states are equal" << endl;
}
```

Remember that `friend` functions are not member functions and so they do not have the pointer `this`.

Let us build a more complicated model of the shift register. For example we want to obtain information about a fixed state named state_fixed which is being defined independently of a particular object. For these purposes a new storage class specifier, `static`, can be used. Consider the following declaration of the state_fixed LEFT_SHIFT_REGISTER static data member:

```
static BOOLEAN *state_fixed;
```

The member state_fixed is called a `static` member. Static members have different properties from nonstatic members. With nonstatic members, a distinct copy exists for each object of the class. With static members only

single copy exists, shared by all objects of the class. It is allowed to initialize static data members outside of a class (if they are accessible), even a particular object has not been defined yet, for instance:

```
LEFT_SHIFT_REGISTER::state_fixed = NULL;
or
LEFT_SHIFT_REGISTER::state_fixed =
     new BOOLEAN[25];
```

The basic use for static members is to keep track of data that is common to all objects of a class. Because there is only a single copy of a `static` function for many objects of the same class, it does not have the pointer `this`, and therefore can only access nonstatic members by explicitly specifying a concrete object with the . or -> selection operator. For example we can consider an alternative way making it possible to set a fixed state. In this case we are keeping our fixed state in a `static` function named, for instance, S_FIXED. This function would be defined as the following:

```
// the first statement represents the
// function declaration within the class
static void S_FIXED(LEFT_SHIFT_REGISTER *lr);
// the following statements show the possible
// function definition
void LEFT_SHIFT_REGISTER::S_FIXED(
     LEFT_SHIFT_REGISTER *lr)
     {   BOOLEAN fixed_state[] =
          { 1,0,1,1,...,0 };
// accessing to class data members explicitly
// using their names (pointers), for example,
// lr -> size; or lr -> state[i];
}
```

The final step of our example is devoted to memory of type register. Suppose we want to consider arrays of registers, which can be:

an array of left shift registers;
an array of right shift registers.

In other words we intend to build containers of registers. They might represent a logical model of a physical scheme located on a single chip, which is designed for general application. Basically our task is aimed at constructing a family of related classes which provide an array of left shift registers and an array of right shift registers. This can be done within the boundaries of OOT using so called templates (`template` is C++ key word). These allow you to define a pattern for either class definitions, or a family of related functions, by setting the data type itself as a parameter. Borland C++ container classes such as stacks and arrays are good example of using templates [4].

Suppose l_or_r is a type which can be either

LEFT_SHIFT_REGISTER

or

RIGHT_SHIFT_REGISTER.

A declaration “template<class l_or_r>“ says that l_or_r is a type name. A class `template` specifiers how individual classes can be constructed. For our example the individual classes might be:

the class of left shift registers;
the class of right shift registers.

As a result our class `template` would be look something like the following:

```
template<class l_or_r> class ARRAY:
     public LEFT_SHIFT_REGISTER,
     public RIGHT_SHIFT_REGISTER
{  l_or_r **data;
   unsigned array_size;
 public:
   ARRAY(unsigned ARRAY_SIZE,
         unsigned SIZE,
         BOOLEAN *INITIAL_STATE = NULL);
   ~ARRAY(void);
   l_or_r& operator [] (unsigned x)
      { return *data[x]; }
};
template<class l_or_r> ARRAY<l_or_r>::
ARRAY(unsigned ARRAY_SIZE,
     unsigned SIZE,
     BOOLEAN *INITIAL_STATE) :

LEFT_SHIFT_REGISTER(SIZE,INITIAL_STATE),

RIGHT_SHIFT_REGISTER(SIZE,INITIAL_STATE)
     {  data = new l_or_r* [ARRAY_SIZE];
        for(int i=0;i<ARRAY_SIZE;i++)
          data[i] = new l_or_r(SIZE);
        array_size = ARRAY_SIZE;
     }
     template<class l_or_r>
     ARRAY<l_or_r>::~ARRAY()
     {  for(int i=0;i<array_size;i++)
          delete data[i];
        delete [] data;          }
```

We can continue a refinement of our example. Another varieties of devices (counters, decoders, etc.) could be investigated. So a new hierarchical structure could be built. In turns our devices might be considered as a building blocks for more complicated digital schemes, such as microprocessors, microcontrollers and microcomputers. They could be further developed in modern computers and computer systems. In any possible level, object models are applicable in a habitual and natural form. Starting from very simple devices, being considered above, we can develop our application to solve, for example, different tasks of logical simulation and digital synthesis (see, for instance [5]).

III. CONCLUSIONS

We have discussed a simple example demonstrating how OOT in general, and the C++ language in particular, can be used to represent logical models of digital devices. Briefly, the consequences resulting from what we have discussed are the following.

1. The key concepts of OOP are: encapsulation (class declaration, objects definition, protecting class members, defining object access rules); inheritance (single inheritance, multiple inheritance, abstract classes); polymorphism (function overloading, operator overloading, virtual functions, templates).

2. OOP defines techniques that provide for [3]: describing an object structure, or `class` (in C++ classes also can be represented by structures and unions with slightly different rules for accessing members); describing methods or member functions for the manipulation of objects using unique object inheritance principles; protecting object members and defining the rules for accessing objects; passing messages between objects; eliminating or at least reducing global data.

3. OOP directly supports hierarchical ordering which can be considered as one of the most powerful tools for managing complexity. Using inheritance, object-oriented program can be organized as a set of trees or directed acyclic graphs of classes [1]. The physical building block in object-oriented languages is the module, comprising a set of classes instead of subprograms as in procedural languages. In large systems the object model scales up [2]. Clusters of abstractions can be built in layers on top of one another [2].

4. As followed from point 3, an object model is closely related to finite automata models [5]. It is also mentioned in [2, see p. 90]. Let us return to our example. On the one hand a register is an object. On the another hand it can be considered as tiny, independent machine referring to finite automata theory being developed for a long time. Digital schemes, containing digital devices like registers, counters, decoders, etc., can be described as a set of classes using the basic OOP principles mentioned above. However they can be also considered as a network of machines from the perspective of finite automata theory. Moreover it is an approach which can be used to design parameterized matrix digital schemes [5]. At present templates are developing in nearly the same direction.

5. These are a brief description of new C++ basic key words:

"`class`" which is used to represent a general structure of a set of objects (see also "`struct`" and "`union`" key words [3,4]);

"`delete`", "`new`" which are used to dynamically allocate and deallocate memory;

"`friend`" which is used to design non-member functions accessing private and protected class members;

"`operator`" which is used to overload most of the predefined operators in C++;

"`private`", "`protected`", "`public`" which are used to set predefined rules for accessing class members. You can assign member attributes (private, protected, public) when you declare class members, and when you specify the base classes for derived classes;

"`template`" which is used to build parameterized types - in other words to set the type itself as a parameter;

"`this`" which is used to represent a hidden pointer which is unique for each object, and points (addresses) to the object itself in computer memory;

"`virtual`" which is used to support overloading during execution (to provide late binding).

C++ introduces also some additional key words that are: "`catch`", "`throw`", "`try`" (exception handling, see, for example, [1,4]), "`inline`" (to make function as inline function, see, for instance [3]).

Finally, there is a comprehensive, interactive, animated, graphical tutorial program on C++ and object-oriented concepts [3]. You can refer to [3, p 5-7] to understand how to use the accompanying disk containing the tutorial. The teaching program makes learning C++ as easy as possible. There are also many C++ examples on the tutorial disk.

REFERENCES

[1] Bjarne Stroustrup. The C++ programming language. Second Edition, Addison-Wesley Publishing Company, 1994, 691 p.

[2] Grady Booch. Object-Oriented Analysis and Design. Second Edition. The Benjamin/Cummings Publishing Company, Inc.,1994, 589 p.

[3] Valery Sklyarov. The Revolutionary Guide to Turbo C++. Birmingham, WROX, 1992, 352 p.

[4] Borland C++. Programmer`s Guide, Borland International, Inc., 1993, 326 p.

[5] V.A.Sklyarov. Matrix LSI Finite Automata Synthesis. Minsk,Science and Technique, 1984, 372 p.