C++ a better language for engineering applications

Danilo Erricolo
College of Engineering
University of Illinois at Chicago
851 South Morgan St,
Chicago, IL 60607
e-mail: derricol@eecs.uic.edu

1 Introduction

C++ is superior to older computer languages because it makes it easier to develop, maintain, update, and reuse programs. Its main advantages are the presence of pointers, dynamic data structures, objects, and the mechanisms of data hiding, inheritance and overloading. Notwithstanding all of these features, most of the people in the scientific environment still program in FORTRAN because they want to be able to use the scientific libraries that were originally written in FORTRAN. In addition, those who prefer FORTRAN are likely to think that C++ is difficult to understand and that only those who are in computer science can use it. You however need not be in computer science to take advantage of C++ and its capabilities. With this review, we attempt to convince you that C++ is not so difficult to use and that it offers tremendous advantages over FORTRAN.

Let us introduce the various features of C++. One feature of C++ is that complex numbers are included in the language. Additionally, mathematical expressions containing complex numbers have the same syntax of those for real numbers. Dynamic data structures, such as vectors, lists, trees, and graphs, are available through the Standard Template Library (STL) avoiding, therefore, all troubles of dealing with their implementation. In contrast to FORTRAN, C++ is an object oriented programming language (OOP), which means that a C++ program is built around the concept of object. An object is simply a data type composed of two parts: the internal data representation and the interface. The interface is constituted by all of the internal functions through which the object interacts with the other parts of the program; in addition, only these internal functions have access to the internal data representation. Therefore, the internal data representation is not available to the environment outside the object and this is referred to as data hiding. Because of data hiding and because each object interacts only through the functions in the interface, the structure of the program is extremely clear. Therefore, it is easier to develop, document, and upgrade the program in a shorter time overall. OOP languages provide inheritance. In practice, this means that thousands of lines of code can be avoided because instead of redefining from scratch an object that is very similar to another one already existing, one simply defines a new object that inherits the properties of the old one and just adds the new features that are required. OOP languages support operator and functions overloading. This is an extremely convenient characteristic that allows one to still write A*B to indicate the product of two matrices *A* and *B*, or to always write *print(A)* to display the content of *A* when *A* is a complex number, or a vector, or another kind of object.

To cover most of the innovative features of C++, we will consider the definition of an object, an example of operator overloading, the mechanism of inheritance and the use of the Standard Template Library.

2 Definition of an object

In C++, an object is defined by declaring the *class* (or type) to which it belongs. A class definition contains two parts indicated by the keywords *private* and *public*. The keyword *private* refers to the internal data representation that is not shared with other parts of the program, whereas *public* refers to the functions (or methods), which have access to the internal data, that are the only means to interact with the object. The advantage of having a private and a public part is that the details of each object are developed into separate units. This helps structuring the program in a clear way and it also makes it easier to locate errors. As an example, let us consider a simple definition for a class of *triangle* objects.

```
class triangle{
private:
     double x[3],y[3];
public:
triangle(double x0, double y0,
         double x1, double y1,
         double x2, double y2); // constructor
double Area() const; //returns the area of the triangle
};
```

The *private* area contains two arrays of double, *x* and *y*, to represent the coordinates of the vertices. The *public* area contains the functions *triangle()* and *Area()*. The function *triangle()* has the same name of the class and is called the constructor; its purpose is to give initial values to private data of the class. The declaration of the function *Area()* contains the keyword *const*, which means that *Area()* is a function that does not change any internal value (this is an additional information that is passed to the compiler to prevent errors). Both the constructor and *Area()* are only declared here (i.e., just the interface for the usage is provided); their definition must be provided somewhere else. The class *triangle* is used as in:

```
triangle t; // defines an instance of a triangle
double area;
area=t.area(); //returns the value of the area by executing
                 a function associated with object of the
                 class triangle.
```

The mechanism of data hiding provided by *private* avoids all programming errors due to the change of value of variables that are not explicitly wanted. In fact, in C++ private data of a class can only be modified by introducing a function that belongs to the same class and, therefore, has access to them. In the previous example, there is no way to modify the internal data *x* and *y*; however by modifying the class *triangle* as:

```
class triangle{
private:
double x[3],y[3];
public:
triangle(double x0, double y0,
         double x1, double y1,
         double x2, double y2); // constructor
double Area() const; //returns the area of the triangle
void ChangeVertex(int i, double x, double y);//change the
coordinates of Vertex i to the new values (x,y).
};
```

the coordinates of the vertices can be explicitly changed. Notice that an attempt to modify the coordinates by direct access to the internal data as in:

```
triangle T;
…
T.x[1]=2; //illegal
```

is illegal and the compiler will issue an error. Instead, the correct way to change the coordinates is through invocation of the *public* function *ChangeVertex()*, as in:

```
T.ChangeVertex(1,2); // now the value of T::_x[1] is 2
```

(the symbol *::* is the scope resolution operator, it is used to refer explicitly to members of a class).

3 Operator overloading

A very convenient feature offered by C++ is operator overloading, which means that ordinary operators (such as +, -, /, *) can be given new meanings according to the required needs. Let us demonstrate how elegant this characteristic is, by developing a program that defines two classes of objects, a vector and a matrix, and the operator * so that one can write M*v even when M is a square matrix and v is a column vector. The program begins with:

```
#include <iostream.h>
#include <math.h>
```

which is a statement that tells the compiler to include the libraries that contain the basic input/output operation and the mathematical operations. Then we define a vector class *Vector4* and, later on, a matrix class *Matrix4*.

```
class Matrix4; // forward definition, in order to avoid a
                  circular problem
class Vector4{
private:
     double _v[4];
public:
     Vector4();
double operator[] (int i)
     { return _v[i];};
void set(int i, double z)
     { _v[i]=z;};

friend Vector4 operator *(const Matrix4& m, const Vector4&
v);
friend Vector4 operator *(const Vector4& v, const Matrix4&
m);
friend Matrix4 operator *(const Matrix4& m1, const Matrix4&
m2);

void print()
     {
     for (int i=0; i<4; i++)
          cout<<_v[i]<<"  ";
     cout<<endl;
     return;
     };
};
```

The class *Vector4* represents a vector with four components that are hidden from the environment outside the class by placing them in the private area. The public area of *Vector4* contains the minimal definitions that are needed for the sake of this example. They are the constructor *Vector4()*, the subscript operator *[]*, the *set()* function, and three different multiplication operators *.The subscript operator *[]* allows one to refer to the m-th component of the vector by writing v[m]. The *set()* function modifies the i-th component of a Vector4 object. The three different multiplication operators * give meaning to expressions such as M*v, v*M, and M*M, where M is a matrix and v a vector. Notice that the operator * is binary and that the language treats differently M*v from v*M; therefore both possibilities must be specified. The syntax of the declaration of the operator contains the keyword *friend* because the operator * needs access to the internal representation of both *Matrix4* and *Vector4* but it cannot be a member of both, hence it is declared *friend* of both. The syntax of the parameters for the operators * such as in:

```
const Matrix4& m
```

contains an ampersand *&*. The ampersand means that the parameter *m* refers to an object of the class *Matrix4* (and not to a local copy of it) and that the object referred to by *M* will not be modified due to the keyword *const*. The *print()* function is introduced to provide an easy way to display the content of a vector and it contains the instruction:

```
cout<<_v[i]<<"  ";
```

The operator << ("put to") transfers its right argument to its left argument. In this case, the content of the variable *_v[i]*, followed by an empty space, is written onto the standard output stream *cout*. Having defined the class *Vector4*, we have all the tools to complete the definition of the class *Matrix4* that was left indicated at the beginning of this example, to avoid problems of circular definitions:

```
class Matrix4{
private:
     Vector4 _m[4];
public:
     Matrix4();
     void set(int i, int j, double z)
     {
     _m[i].set(j,z);
     return;
     };

     friend Vector4 operator * (const Matrix4& m, const
     Vector4& v);
     friend Vector4 operator * (const Vector4& v, const
     Matrix4& m);
     friend Matrix4 operator * (const Matrix4& m1, const
     Matrix4& m2);
     void print()
     {
          for (int i=0; i<4; i++)
          {
               _m[i].Vector4::print();
               //executes print() of the class Vector4
               cout<<endl;
          };

          return;
     };
};
```

The class *Matrix4* is built upon the class *Vector4* in the sense that the internal representation of a *Matrix4* object is a *Vector4* object, whose components are in turn *Vector4* objects. Again, there are three different definitions of the operator * because of the same reason explained previously in relation with the class *Vector4*. The implementations of the operators are:

```
Vector4 operator * (const Matrix4& m, const Vector4& v)
{//executes m*v
     Vector4 res;
     for (int i=0; i<4; i++)
     {
          res._v[i]=0;
               for (int j=0; j<4; j++)
               res._v[i]+=m._m[i]._v[j]*v._v[j];
     };
     return res;
};

Vector4 operator * (const Vector4& v, const Matrix4& m)
{//executes v*m
     Vector4 res;
     for (int i=0; i<4; i++)
     {
          res._v[i]=0;
          for (int j=0; j<4; j++)
                 res._v[i]+=m._m[j]._v[i]*v._v[j];
     };
     return res;
};

Matrix4 operator * (const Matrix4& m1,const Matrix4& m2)
{//executes m*m
     Matrix4 res;
     for (int i=0; i<4; i++)
          for (int j=0; j<4; j++)
          {
               res._m[i]._v[j]=0;
                for (int k=0; k<4; k++)

res._m[i]._v[j]+=m1._m[i]._v[k]*m2._m[k]._v[j];
          };
     return res;
};
```

Now that everything has been declared, we are ready to witness the elegance of the programming style using overloaded operators. The main part of the program is:

```
int main(){
int i,j;
Matrix4  A,M;
Vector4 v,L,R;

cout<<"Let us create a matrix A such that the element (i,j)
     is i*j+i"<<endl;

for (i=0; i<4; i++)
     for (j=0; j<4; j++)
```

```
          A.set(i,j,(i+1)*(j+1)+(i+1));

A.print();

cout<<"Let us create a vector such that the element i has
 value sin(i)"<<endl;

for (i=0; i<4; i++)
v.set(i,sin(i));

cout<<"Left product L=v*A:"<<endl;
L=v*A; //same notation used in mathematics
L.print();

cout<<"Right product R=A*v:"<<endl;
R=A*v; //same notation used in mathematics
R.print();

cout<<"Matrix product M=A*A:"<<endl;
M=A*A; //same notation used in mathematics

M.print();

return 0;
}
```

The output of the program is:

```
Let us create a matrix A such that the element (i,j) is
i*j+i
2    3    4    5
4    6    8    10
6    9    12   15
8    12   16   20
Let us create a vector such that the element i has value
sin(i)
Left product L=v*A:
9.95063  14.9259  19.9013  24.8766
Right product R=A*v:
6.8672  13.7344  20.6016  27.4688
Matrix product M=A*A:
80  120  160  200
160  240  320  400
240  360  480  600
320  480  640  800
```

4 Inheritance

C++ provides the inheritance mechanism which means that given a class B, the base class, it is possible to create a new class D, the derived class, that has in common with B the same private and public members. A derived class is usually introduced when the members (both private and public) of the base class are all needed but are not enough for a specific purpose. In this way, a derived class constitutes a refinement of the base class. Note that the definition of the derived class requires only the specification of the new features, therefore there is no need to redefine the common operations for the derived class and this saves both lines of code and time. Let us consider an example of inheritance by creating a base class *Triangle* where the private data members are the coordinates of the vertices and the public methods are the constructor, a function that returns the area of the triangle, and a function that prints its coordinates. As a derived class, we create *TriangleWithColor* that has one color attribute among its private members (in addition to those derived from the base class) and two more public functions to set and display the color attribute.

```
#include <iostream.h>
#include <math.h>

class Triangle {
private:
double _x[3],_y[3];// coordinates of the vertices numbered
from 0 to 2
public:
     Triangle(double x0, double y0, double x1, double y1,
            double x2, double y2)
     { _x[0]=x0, _x[1]=x1; _x[2]=x2;
       _y[0]=y0; _y[1]=y1; _y[2]=y2;
     }; //definition of the constructor

     double Area() const; //returns area of the triangle

     void SetVertex(int i, double X, double Y) // updates
                 the coordinates of vertex i;
     { _x[i]=X; _y[i]=Y; return;};

     void print(); // prints the coordinates
};
```

As an example of a derived class we have created *TriangleWithColor* that inherits both the private part and the public part from *Triangle*.

```
class TriangleWithColor:public Triangle{
private:
     char _color; // represents the color attribute with a
                      character
public:
     TriangleWithColor(double x0, double y0, double x1,
     double y1, double x2, double y2, char
     c):Triangle(x0,y0,x1,y1,x2,y2) {_color=c;};

     void SetColor(char c) //updates the color attribute
     { _color=c; return;}

     char DisplayColor() //returns the color attribute
     {return _color;};

     void print(); //prints the coordinates and the color

};
```

The definition of *TriangleWithColor()* contains one additional character attribute that represents the color of the triangle. Because of the inheritance mechanism, the same functions defined in the base class are available in the derived class unless one chooses to modify them. The functions *SetVertex()* and *Area()* are only defined in the base class and the latter is given by:

```
double Triangle::Area() const
{ return fabs(0.5*(_x[0]*_y[1]+_x[1]*_y[2]+_x[2]*_y[0]-
_x[1]*_y[0]-_x[2]*_y[1]*_x[0]*_y[2])); }
```

Referring to the definitions of the base class *Triangle* and of the derived class *TriangleWithColor* we see that (in the following example), after creating an object *T* of the base class and an object *TC* of the derived class, the instruction *TC.Area()* and *T.Area()* both refer to the same implementation of *Area()* given in the base class. Whereas for *Area()* the same behavior is acceptable for both the base class and the derived class, for the function *print()* we want to introduce a different behavior that distinguishes an object of the base class from an object of the derived class. Since it is desirable to keep the same name for the function, we simply define *print() as:*

```
void Triangle::print()
{
     cout<<"("<<_x[0]<<","<<_y[0]<<"),
("<<_x[1]<<","<<_y[1]<<"),
("<<_x[2]<<","<<_y[2]<<")"<<endl;
     return;
};

void TriangleWithColor::print()
{
     Triangle::print(); // execute print of the base class
     cout<<"color='"<<_color<<"'"<<endl;
     return;
};
```

The definition of *TrianglewithColor::print()* uses the *print()* function of the base class and displays the color attribute that is not available in the base class. The compiler determines the correct *print()* function to use according to the class to which the object belongs to. Let us refer to the following example:

```
int main(){
     cout<<"Let's create a triangle with vertices"<<endl;
     cout<<"(0,0), (2,0), (0,2)"<<endl;;
     Triangle T(0,0,2,0,0,2);

     cout<<"Let's print its area: "<<T.Area()<<endl;

     cout<<"Let's create a triangle with vertices"<<endl;
     cout<<"(0,0),   (4,0),   (0,4)   and   color   attribute
           'a'"<<endl;
     TriangleWithColor TC(0,0,4,0,0,4,'a');

     cout<<"Let's print its area:"<<TC.Area()<<endl;
          // executes Triangle::Area()

     cout<<"Let's   print   the   coordinates   of   the   first
           triangle"<<endl;;
     T.print(); //executes Triangle::print()

     cout<<"Let's   print   the   coordinates   of   the   second
triangle"<<endl;;
     TC.print(); //executes TriangleWithColor::print()

     cout<<"Let us change one vertex of the second triangle
            from (4,0) to (8,0)"<<endl;
     cout<<"and    print    the    coordinates    of    the
            triangle"<<endl;
     TC.SetVertex(2,8,0); // executes Triangle::SetVertex
     TC.print(); //executes TriangleWithColor::print()

     cout<<"Use the function print() of the base class for
           the derived object"<<endl;
     TC.Triangle::print();

     return 0;
}
```

The output of the program is:

```
Let's create a triangle with vertices
(0,0), (2,0), (0,2)
```

```
Let's print its area: 2
Let's create a triangle with vertices
(0,0), (4,0), (0,4) and color attribute 'a'
Let's print its area:8
Let's print the coordinates of the first triangle
(0,0), (2,0), (0,2)
Let's print the coordinates of the second triangle
(0,0), (4,0), (0,4)
color='a'
Let us change one vertex of the second triangle from (4,0)
to (8,0)
and print the coordinates of the triangle
(0,0), (4,0), (8,0)
color='a'
Use the function print() of the base class for the derived
object
(0,0), (4,0), (8,0)
```

T is an object of the base class, whereas *TC* belongs to the derived class. Invocation of *Area()* results into executing the same code for both objects. However, *print()* behaves differently when called for *T* and *TC* because these objects belong to different classes. The function *SetVertex()* is only defined in the base class and it can be applied to the derived object. Notice that to use a base class function when a function with the same name is provided into a derived class, the base class function must be invoked explicitly, such as in:

```
TC.Triangle::print().
```

The inheritance mechanism is also used to create class hierarchies, however this topic goes beyond the purpose of this review.

5 Standard Template Library

Another advantage of C++ is the standard template library that provides support for numerical computation (complex numbers plus vectors with arithmetic operations), dynamic data structures (vector, list, and map) and algorithms for their use (such as general traversals, sorts, and merges). These predefined data structures are easily tailored to suit the needs of a programmer; in this way, one avoids to reinvent the wheel and creates code that is more portable because it is built upon a standard of the language rather than a particular version of it. We will show some features of the STL by developing a program that creates a sorted linked list of elements, i.e. a dynamic data structure that contains elements of the same type (or class) that are ordered in a sequence. Even though lists are among the simplest dynamic data structures, they are important because they may be used to represent complex data structures such as trees or graphs. We will first consider the creation of a list and we will describe how to order it afterwards.

In order to create a list, two components are required: the class of the elements and the operations on the list, such as insertion, access, and removal of an element. Even though all lists are similar in terms of the operations on them, most languages that support lists require to redefine all list operations every time a new list is created (i.e. any time the class of the elements changes). In C++, one advantage of using the STL is that the definition of a list simply requires the declaration of the class of the elements. Referring to the sample program of Section 3, the declaration of a linked list of elements of the class *Vector4* simply requires the use of the template class *list<>* as shown in the following:

```
#include<list> // includes the appropriate library
typedef list<Vector4> VectorList;
```

where the keyword *typedef* means that *VectorList* is a synonym for the class *list<Vector4>*. Therefore, a variable that represents a list of *Vector4* elements is defined as:

```
VectorList List;
```

Because *VectorList* comes from a template, all list operations are already defined. In particular the operation of inserting an element is done using the *insert()* function according to:

```
List.insert(position,element)
```

where *position* is the location (inside *List*) before which *element* is inserted. The function *insert()* serves the purpose to populate the list with elements, but to create a sorted list we must introduce a way to order the elements. Since the elements of a list are ordered according to the numerical value of an attribute, then we extend the class *Vector4* to contain a new function, called *norm()* that returns the norm of a vector. The function *norm()* is defined as:

```
class Vector 4{
private
…
public:
…
     double norm() const
     {
          double res=0;

          for (int i=0; i<4; i++)
               res+=pow(_v[i],2.0);
          return sqrt(res);
     };
…
};
```

The numerical attribute is then used by one of the algorithm of the STL to create a sorted list of elements. In this example, we consider the algorithm *lower_bound()*, whose prototype is:

```
lower_bound(start,stop,attr,rule)
```

lower_bound() returns the location before which an element with attribute *attr* must be inserted into a list that begins at *start* and ends at *stop*. In addition, *lower_bound()* operates under the assumption that the list is already sorted according to *rule*. In the example that we are going to consider, *attr* is the norm of the *Vector4* object to be inserted and *rule* is an operator that compares the norm of the element to be inserted with the norm of the current element under exam in the list. If the *Vector4* objects are sorted in ascending order of their norm, *rule* is expressed by introducing a class *LessThan* as it follows:

```
class LessThan{
public:
     bool operator() (const Vector4& Vect,const double&
Norm) const
     { return Vect.norm()<Norm;};
};
```

Less_Than() contains a boolean operator that returns true whenever the current element in the list is smaller than the norm of the element to be inserted. Now, let us introduce a few elements into the list, while preserving its ordering. Use of the *lower_bound()* algorithm requires that

```
#include<algorithm>
```

be put at the beginning of the program, to include the appropriate library. In addition, we add the statement:

```
using namespace std;
```

where *namespace* refers to a mechanism provided by C++ to group together related data. In this case, *std* stands for the standard library and it allows one to write, for example, *lower_bound()* instead of *std::lower_bound()*. The main body of the program starts with the insertion of the first element:

```
cout<<"Let us create a list that is sorted in ascending
order"<<endl;
List.insert(List.begin(),v); // this is the first element
```

where *v* is the same *Vector4* object of the sample program of Section 3. To create two more *Vector4* objects we do:

```
Vector4 v2,v3;
cout<<"Let us create a vector such that the element i has
value 2*sin(i)"<<endl;
for (i=0; i<4; i++)
     v2.set(i,2*sin(i));
v2.print();

cout<<"Let us create a vector such that the element i has
value 4*sin(i)"<<endl;
for (i=0; i<4; i++)
     v3.set(i,4*sin(i));
v3.print();
```

and we insert them into the appropriate position using *lower_bound()* as in:

```
VectorList::iterator it;
it=lower_bound(List.begin(),List.end(),v2.norm(),LessThan()
); // find the position where v2 should be inserted
List.insert(it,v2);
it=lower_bound(List.begin(),List.end(),v3.norm(),LessThan()
); // find the position where v3 should be inserted
List.insert(it,v3);
```

Notice that we have introduced an iterator *it*, i.e. a pointer that goes through the elements of the list. An iterator is defined by adding *::iterator* after the name of the class of the list. The list now contains three elements in ascending order; we can check its content by doing:

```
int k=0;
cout<<"Let us check the content of the list:"<<endl;
for (it=List.begin(); it!=List.end(); it++)
{
     cout<<"  Element "<<++k<<": ";
     (*it).print(); // display element
     cout<<"             Norm = "<<(*it).norm()<<endl;
               // print its norm
};
```

We can also find a particular element by doing:

```
cout<<"Let us find the first element with norm less than
3"<<endl;
it=lower_bound(List.begin(),List.end(),3,LessThan());
     (*it).print(); // display element
```

Finally, the output of the program is:

```
Let us create a list that is sorted in ascending order
Let us check the content of the list:
  Element 1: 0  0.841471  0.909297  0.14112
             Norm = 1.24692
  Element 2: 0  1.68294  1.81859  0.28224
             Norm = 2.49384
  Element 3: 0  3.36588  3.63719  0.56448
             Norm = 4.98768
Let us find the first element with norm less than 3
0  3.36588  3.63719  0.56448
```

6 Conclusions

In this review, we have shown the features of C++ that make it superior in comparison with other languages. Use of data hiding and objects are an important way to improve the quality of the software that is produced because the final product is easier to develop, maintain, document, and upgrade. Because of the overloading mechanism, the elegance of some notations, such as in the example of the product among vectors and matrices, is now available even when writing a computer program. In addition, the standard template library provides a huge set of templates that save a great deal of time in the development of software. Here only the features of C++ that are relevant to engineering applications have been briefly introduced with some examples. The examples have been verified using the Microsoft Visual C++©, version 6.0. For a formal discussion of C++, the most authoritative source is the book by its creator [1]; however, many will find it easier to refer to textbook such as [2].

7 References

1. B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 1997.
2. S. B. Lippman, J. Lajoie, *C++ Primer, 3rd edition*, Addison Wesley, 1998.