THE ORTHOGONALITY IN C++*

Feng-Jen Yang
Computer Science Department
Prairie View A&M University
Prairie View, TX 77446
936 261-9883
fengjen_yang@pvamu.edu

ABSTRACT

Orthogonality was first introduced to the design of programming languages in 1960s and still stands well to the test of time. A language with better orthogonality tends to be easier to understand and more intuitive to use. In this paper, I examine the orthogonality in C++ from the perspectives of type systems, expressions and flow controls.

1 INTRODUCTION

Orthogonality is originally a geometry terminology used to define a property in a given set of coordinates, by which a set of coordinates is said to be orthogonal, if changing the quantity in one coordinate does not change the projected quantities in other coordinates. While being analogized to programming language design, orthogonality means that a language construct can be used in combinations with other constructs and those combinations are making sense both syntactically and semantically. The meaning of the construct is consistent in spite of those constructs it is combined with.

Algol 68 was the first language paying special attention to orthogonality. One of its design goals was to make the language constructs as orthogonal as possible [1]. The emphasis of orthogonality also deeply influenced the design of many other Algol-like languages, such as Pascal, SNOBOL, PL/1, Ada, C++, and Java [2]. Even nowadays, this principle is still followed by most of the language related projects [3, 4, 5]. The benefits of being orthogonal are especially in favor of programmers. A language with better orthogonality tends to be easier to understand, to use, and to reason about. On the contrary, a non-orthogonal property can easily become a syntactic or semantic pitfall that programmers may fall into.

Although orthogonality does have a great impact on the design of *type systems*, *expressions* and *flow* controls, its relevant discussions are extremely limited in current programming language textbooks. The lack of comprehensive discussions on this design principle motivated this paper. I hereby focus my canvass on C++, not from a standing point of criticism, but pointing out some syntactic or semantic pitfalls caused by the lack of orthogonality. Those pitfalls, of course, can be overcome superficially by pedagogic mechanisms, such as teaching carefully and doing more practices, but, more radically, a language can always be designed more orthogonally to better benefit its followers.

2 THE NON-ORTHOGONAL ASPECTS IN C++

It is not hard to see the orthogonal aspects in C++, since the language itself is already orthogonal to a certain extent that is good enough to make it programmer friendly. For example, a program can embed conditional branches inside loops or vice versa. On the other hand, the liberty of combining program constructs is not unlimited. For example, a function can not return an array and an array can not be passed to a function by value.

Instead of restating the orthogonal aspects in C++ that most programmers are already familiar with, the subsequent discussions are focused on the illustrations of its non-orthogonal aspects to the best of my knowledge. The main purpose of this paper is to inspire some further discussions in the literature of programming languages. All of the programming examples in this paper are designed to be succinct and easy to demonstrate in class.

2.1 Non-Orthogonal Types

The first and most obvious non-orthogonal type is the class type. Although a class type can contain another class type or inherit properties from another class, its non-orthogonal properties come up when some data members are of pointer type. For a class type with pointer data members, there are necessities to overload its default constructor, explicit constructor, copy constructor, destructor and assignment operator to prevent shallow copies, side effects and memory leaks. This cumbersome obligation, as illustrated in example 1, is not only confusing novice programmers but also challenging instructors to account for. Another non-orthogonal property is caused by the special syntax of constructor and destructor. While a regular function must have a return type, the constructor and destructor just have no return type. Besides, the implicit invocations of constructor and destructors are also non-orthogonal, since regular functions are called explicitly.

Example 1: The non-orthogonal properties of class type

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
class Item
{
public:
   Item(){val = new int(0);}; //Default constructor
  Item(int initVal) {val = new int(initVal);}; //Explicit constructor
```

```
  Item(const Item& anItem) //Copy Constructor
  {val = new int(*anItem.val);};
  void operator=(const Item& rhs) //Assignment operator
  {if (this != &rhs) *val = *rhs.val; };
  ~Item(){delete val;}; //Destructor
  void Set(int newVal) {*val = newVal;}; //Change value
  void Write(){ cout << *val << endl; }; //Write value
private:
  int* val;  //A data member of pointer type
};
int main()
{
  Item x,    //Call the default constructor
       y(3), //Call the explicit constructor
       z(y); //Call the copy constructor
  x = y; //Call the assignment operator
  x.Write(); //Display the value in item x
  y.Write(); //Display the value in item x
  y.Write(); //Display the value in item x
  return 0;  //Call the destructor
}
```

The second non-orthogonal type is the enumeration type. Semantically its enumerated symbols are stored as integers and an enumeration variable can be used for integral purposes via some implicit type coercions. Those type coercions are not only fulfilling their integral purposes but also revealing their non-orthogonal properties. When an enumeration variable is used in integral computations, its type will be automatically converted into integer, and, consequently, no longer compatible with the original enumeration type. So, as illustrated in example 2, when an enumeration variable is used as the loop index, it is necessary to have an explicit type casting at the update of the index. Otherwise, by incrementing the index variable its type is implicitly converted into integer and not able to be assigned back to the variable itself. Another non-orthogonal property of enumeration type comes up while outputting an enumeration variable. In example 2, during the execution of the *for* loop, the type of the enumeration variable is, again, converted into integer. So the actual outputs will be *numeric* values from *0* to *4* instead of *symbolic* values from *Mon* to *Fri*.

Example 2: The non-orthogonal properties of enumeration type

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
enum WeekDay {Mon, Tue, Wed, Thu, Fri}; //An enumeration type
int main()
{
  WeekDay d; //An enumeration variable
  for (d = Mon; d <= Fri; d = WeekDay(d+1)) //Casting the updated index
    cout << d << " "; //Output 0 to 4 instead of Mon to Fri
  return 0; //Successfully done
}
```

The third non-orthogonal type is the array type. As illustrated in example 3, there

is no assignment operator allowing an array to be assigned to another array directly. To semantically assign an array to another, it is necessary to have an explicit loop to perform the deep copy of array elements one by one. Another non-orthogonal property of array type is because of the fact that an array name is used to hold the base address that in turns pointing to its first element. The current C++ syntax only allows an array to be passed to a function by reference. Passing an array to a function by value and returning an array from a function are prohibited.

Example 3: The non-orthogonal properties of array type

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
int main()
{
  int a[3], b[3] = {1, 2, 3}, i;
   for (i=0; i<3; i++) a[i] = b[i];   //Assign array b to array a
    return 0; //Successfully done
}
```

The fourth non-orthogonal type is the string type. Although it is not a built in C++ type, it is declared as a class type in the string library and has been popularly used in daily programming. So, it does worth a mention in the context of orthogonality. As illustrated in example 4, when a *string object* is working with an *assignment* operator, the programmer would never feel any difference between a *string object* and a *C string*. Its non-orthogonal property comes up when a string object is passed to the open function of a file stream. Due to the fact that open function is strictly working with *C string* only, an explicit conversion from a *string object* into a *C string* is required.

Example 4: the non-orthogonal properties of string type

```
#include <iostream>  //For standard I/O
#include <string>    //For string type
#include <fstream>   //For file I/O
using namespace std; //Reserve library objects for standard uses
int main()
{
  string str;   //A string object
  ofstream ofs; //An output file stream
  str = "data.txt";    //Used like a C string
  ofs.open(str.c_str());//Converted to C string for file opening
  if (ofs.fail())
  {
    cout << "File Open Failure !" << endl;
    exit(1); //End execution immediately
  }
  ofs << 38 << endl; //Write to output file
  ofs.close(); //Close the output file
  return 0;  //Successfully done
}
```

2.2 Non-Orthogonal Expressions

The most obvious non-orthogonal expressions are those involving the increment and decrement operators. As illustrated in examples 5 and 6, since the *prefix* and *postfix* formats result in different precedence of evaluations, their overall results are also different. The implicit changes of precedence among the increment, decrement and multiplication can result in very confusing semantics.

Example 5: The non-orthogonal properties of the postfix and postfix increment

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
int main()
{
  int i=3, x;
  x = i++ * 3; //Perform * first and then ++
  cout << x << endl;  //Display 9
  cout << i << endl;  //Display 4
  x = ++i * 3; //Perform ++ first and then *
  cout << x << endl;  //Display 15
  cout << i << endl;  //Display 5
  return 0;  //Successfully done
}
```

Example 6: The non-orthogonal properties of the prefix and postfix decrement

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
int main()
{
  int i=3, x;
  x = i-- * 3; //Perform * first and then --
  cout << x << endl;  //Display 9
  cout << i << endl;  //Display 2
  x = --i * 3; //Perform -- first and then *
  cout << x << endl;  //Display 3
  cout << i << endl;  //Display 1
  return 0;  //Successfully done
}
```

2.3 Non-Orthogonal Flow Controls

The first non-orthogonal flow control happens when an *arithmetic expression* is used as a *logical expression* in conditional branches. The special semantic that any *nonzero* value is considered as logical *true* and only *zero* is considered as logical *false* can makes some conditional branches very confusing. As illustrated in example 7, the original intentions of the program are to test whether *x* is between *1* and *3* and then test whether *x* is equal to *0*, but the programmer accidentally uses mathematical notations to represent the interval and equality and result in a program that is syntactically correct but semantically wrong. Within the first *if* statement, the evaluation of *1<x<3* can be parenthesized into *((1<x)<3)*.

Since the current value of *x* is *0*, the *(1<x)* is evaluated into logical *false* and the entire expression becomes *(false < 3)*. The special semantic that considering logical *false* as *0* will make the overall expression to be evaluated into logical *true*. Within the second *if* statement, instead of checking the equality, the *x=0* is treated as assigning *0* to *x* which in turns has a numerical result of *0*. The special semantic that considering *0* as logical *false* will make the overall expression to be evaluated into logical *false*.

Example 7: The non-orthogonal properties of conditional branches

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
int main()
{
  int x=0;
  if ( 1<x<3 ) //Evaluated as logical true
    cout << "Yes" << endl;
  else
    cout << "No" << endl;
  if ( x=0 ) //Evaluated as logical false
    cout << "Yes" << endl;
  else
    cout << "No" << endl;
  return 0; //Successfully done
}
```

The other non-orthogonal flow control happens in the context of calling virtual functions between a super class and its subclass. The regular manner of calling a function is specified by the function name, the number of parameters, and the types of parameters. To this end, the two *write* functions in example 8 make no difference. The special syntax that a *reference parameter* of the super class type can accept an incoming argument to be *either* the super class object *or* its subclass object makes the calling context very confusing. In this example, the type of the argument being passed to the *print* function is the real key to determine the calling of either the *write function in class A* or the *write function in class B*.

Example 8: The non-orthogonal properties of virtual functions

```
#include <iostream>  //For standard I/O
using namespace std; //Reserve library objects for standard uses
class A  //Declare the super class
{
public:
  A(){ x=0; };   //Initialize x only
  virtual void Write(){ cout << "x=" << x << endl; }; //Write x only
private:
  int x; //The super class data
};
class B : public A   //Declare the subclass
{
public:
  B(){ y=0; }; //Initialize both x and y
  virtual void Write() //Write x and y
```

```
 { A::Write(); cout << "y=" << y << endl; };
private:
  int y;  //The subclass data
};
void Print(A& aORb) //A reference parameter of the super class type
{
  aORb.Write(); //Call the Write() in A or B
  cout << endl;
}
int main()
{
  A a;  //A super class object
  B b;  //A subclass object
  Print(a); //Call a.Write() in Print()
  Print(b); //Call b.Write() in Print()
  return 0; //Successfully done
}
```

3 CONCLUSION

Orthogonality is commonly perceived as a desirable property of programming languages. In this paper, I illustrated some non-orthogonal aspects of C++ that involve extra obligations to programmers and seriously affect the expressiveness of the language. To this end, I am expecting the next generation of programming languages to be more orthogonal and able to read human mind better. The downside of this expectation could be shifting too much work to the design of compilers or interpreters. Nevertheless, considering the dramatic and nonstop improvement on both software and hardware technologies, I am optimistic with this expectation.

4 REFERENCES

[1] Lindsey, C.H., A History of Algol 68, *ACM SIGPLAN Notices*, 28(3), 97-132, 1993.

[2] Malik, M.A., Evolution of the High Level Programming Languages: a Critical Perspective, *ACM SIGPLAN Notices*, 33(12), 72-80, 1998.

[3] Zhang, C. and Jacobsen, H., Resolving Feature Convolution in Middleware Systems, *ACM SIGPLAN Notices*, 39(10), 188 – 205, 2004.

[4] Dantas, D.S., Walker D., Washburn G. and Weirich, S. 2005, PolyAML: A Polymorphic Aspect-Oriented Functional Programming Language, *ACM SIGPLAN Notices*, 40(9), 306-319, 2005.

[5] Fisher, K., Mandelbaum, Y. and Walker, D., The Next 700 Data Description Languages, *ACM SIGPLAN Notices*, 41(1), 2-15, 2006.