# Generic Programming in C++, Delphi and Java

Xu Wen-Sheng, Xue Jin-Yun
College of Computer Information Engineering,
Jiangxi Normal University, Nanchang 330027, China
Key Laboratory for Computer Science, Institute of Software,
the Chinese Academy of Sciences, Beijing 100080, China
Email: xuwen1@263.net

**ABSTRACT**

The main purpose of generic programming is to improve the reliability and reusability of software at an abstract level, and some generic program libraries, to a certain extent, have reached it successfully. While there are many high-level programming languages that have embedded with generic mechanisms to make generic development easier, there yet exist some languages without any mechanism, including Delphi and Java languages. This paper intends to make clear the standpoint that even without generic mechanism some languages still support generic programming. Followed description of the key idea of generic programming, the approaches for genericity implementation in C++, Delphi and Java are present and their analyses are given also. A simple generic program is implemented respectively for illustration.

**KEYWORDS**

Generic programming, parameterization of OO language, genericity implementation, C++, Delphi, Java

**CATEGORIES AND SUBJECT DESCRIPTORS**

D3.3 [Programming Languages]: Language Constructs and Features — classes and objects, polymorphism.

## 1. INTRODUCTION

Generic mechanism [8-10, 12-13] is a language feature that is used to improve expressivity, clarity and efficiency in developing generic program. In the early age, Ada language [3] provided generic structure, which could declare three kinds of parameters: value, type and subroutine, to support generic programming. C++ language [5-6] depicts generic program as a template, which can only accept two kinds of parameters: value and type (or class). Is there equivalence in expressivity of genericity between these two mechanisms? Java [7] is a simpler and clearer object-oriented language than C++, although they are similar in language syntax. Because of the lack of generic mechanism, Java doesn't allow program with type parameter executed directly in Java Runtime Environment (JRE), which makes Java program independent of concrete machine and language. Can we find a way to solve the genericity problem in Java? Delphi [4] is a typical structural programming language mixed with object-oriented technique derived from object Pascal, which permits subroutine type to be used as subroutine parameter with strong type checking. How about to implement generic programming in Delphi? Providing an answer for those problems will deepen understanding of generic programming with respect to a novelty software development technique and also make generic programming applied in a wider area.

This paper will progress as follows: Section 2 briefly presents the key idea of generic programming. Section 3 provides a detailed description of genericity in three languages, and Section 4 discusses the different implementations of a simple program: generic stack. Section 5 concludes.

## 2. GENERIC PROGRAMMING

Generic programming [1-2, 11] aims to obtain generic programs that embody non-traditional kinds of parametric polymorphism to produce the programs more general. In contrast to normal programs, the parameters of a generic program are often quite rich in structure. According to Ada language, for example, it permits parameters of value, type, and subroutine. These three kinds of parameters stand for major forms of program constituents that can be invoked by a name, and they also can be used as underlying structures

This research was supported by NNSF of China (Grant No. 60273092); the NGFR 973 Program of China (Grant No. 2003CCA02800).

to compose more complex kind of parameter, such as programming pattern in [6]. So mechanism for genericity implementation needs to contain these three kinds of parameters.

The construction of generic programs is built on abstract parameters that are unexecuted due to the fact that generic programs are stripped of relatively irrelevant details. In the process of instantialization, abstract parameters of generic programs would be replaced by some concrete arguments, which should be consistent with the corresponding parameters. For instance, Ada provides statement 'new' for instantialization, which can translate generic programs into related specific instances. Specific instances have none of abstract parameters, and consequently to be invoked directly as normal programs. In short, generic program consists of three interacting parts:

1. abstract generic program;
2. instantialization;
3. application of the instance.

For example, generic quick sort program coded in Ada can be defined as follows:

```
--abstract generic program: Sort
generic
    Size: Natural;
    type Item is private;
   with function "<"(x,y:Item) return Boolean;
package Sort is
   procedure quicksort(V: in out Vector);
   private
    type Vector is array (1..Size) of Item;
end Sort;
package body Sort is …
--instantialization with Integer
package IntSort is new
        Sort(Size=>10,Item=>Integer, "<"=>"<");
--application of the instance: IntSort
Vector X;
IntSort.quicksort(X);
```

This example includes all three kinds of parameters and composing parts suitable to completely exhibit the feature of generic program. Therefore in the rest of paper, we will use it for further research.

## 3. GENERICITY IMPLEMENTATION IN C++, DELPHI AND JAVA

### 3.1 C++ and Genericity

C++ is a practical and industrial programming language and many features are brought together in harmony so as to support structural programming, object-oriented programming and even generic programming as well. The syntax to declare a parameterized class is extended by including an optional part of template structure that is followed the type parameters delimited with angle brackets and separated by commas, such as template<type $T_1$,…,type $T_n$>.

In C++, template is reserved words for genericity and its followed parameter list <type $T_1$,…,type $T_n$> must match the number of actual arguments. Type is restricted within two kinds: class, primitive type. If it is primitive type, the related parameter can only match a constant value, otherwise it can match any type of object.

Ada and C++ follow the style of heterogeneous translation that produces a specialized version of the code for each different instantiation. The deficiency is that type parameter is separated from its operators during declaration, which may make variable of type parameter misuse wrong operators without any detection in the compile time. It is shown in the example of Section 2 that Item and its operator "<" are defined separately. Moreover, if needed, Ada is able to define a single subroutine parameter, but C++ doesn't. The lack of subroutine parameter will make C++ less expressive than Ada.

There is a way to remedy it by the adoption of subroutine pointer. C++ supports subroutine pointer type that is similar in syntax to C language and the subroutine pointer type is differentiated by its prototype. Its variable can refer to a family of subroutines that satisfies the defined prototype. Towards the above example, operator "<" can be defined as follows:

```
template <int Size, class Item>
class Sort {
   bool (*cmp)(Item x, Item y);
   …
}
```

It is obvious that operator cmp is defined as a doubleton function pointer which would actually refer to a function after instantiation. That is,

```
bool intcmp(int x, int y) { return x<y;}
Sort<10,int> A;
A.cmp=intcmp;
```

### 3.2 Delphi and Genericity

Delphi, originated from Pascal language, is a procedural programming language mixed with object-oriented techniques. It supports object-oriented programming by introduction to class structure, which can define a subclass by suffixed a class that is delimited with parenthesis, and also can define a polymorphic method by suffixed reserved words 'Virtual'. For example, C=class(C1) function m(x: Integer, y: Integer):Integer; Virtual; end;, where C is subclass of C1 and m is polymorphic method.

On the other hand, it extends procedural programming with subroutine type, which can be differentiated each other by subroutine prototype. Subroutine type is defined by a subroutine prototype without subroutine name, for example, P=procedure(x: integer, y: integer);, where P is procedure type with two integer parameters.

Not like C++, Delphi doesn't provide any generic parameter structure for class, so it would be treated in another way. The heterogeneous translation style produces a specialized version for each instance, namely, every instance should have a dependent copy of generic program that substitutes all occurrences of generic parameter for specific one. It is called generic programming in translation. On the contrary, the homogeneous translation style generates common code for all instances, which can preserve generic code both in source level and in runtime. Its weakness is that primitive type needs to be wrapped into a class if it attempts to be treated by this common code. Owing to the lack of generic structure in Delphi, the homogeneous style is still better.

Type parameter stands for a set of classes with the common interface. On account of inclusion relation to interface between parent class and its all subclasses, type parameter can be denoted with a class that is inherited class of all possible actual argument classes. If there is no restrict on argument class, then type parameter is thought of as the topmost class, TObject. Any method in the class needs to be polymorphic, which allow instantiation by the way of the overriding method of actual argument. Because overriding method needs the same name and the same parameter list as a method declaration in parent class, it is necessary to explicitly downcast the object of parent class to actual class in method body. Now an example is given. Generic quick sort program is defined in Delphi as follows:

```
Type
  Item=class
  end;
  SrCmp=function(x:Item,y:Item):Boolean;
  Vector=array[1..100] of Item;
  Sort=class
    procedure quicksort(V: Vector);
  end;
var
  cmp: SrCmp;
  Size: Integer;
procedure Sort.quicksort(V: Vector) begin … end;
```

The above program quicksort can be reused by other units, when class Item, function variable cmp and integer variable Size have been instantiated individually to specific ones by defining a concrete subclass of Item , or assigning a function to cmp and integer value to Size. Optionally, subroutine parameter cmp could also be included in type parameter Item, which would make them encapsulated to be integral.

### 3.3 Java and Genericity

Java aims to provide a pure and rigid object-oriented programming language with the ability to easily support contributed computing and flexibly develop programs in both large and small size, and its syntax is similar to C++ but without template structure. In order to extend Java with genericity, there are several proposals such as PolyJ, Classloader, GJ and etc. PolyJ extends typedef structure in class that can relate virtual type directly to their actual type, but the main deficiency is its poor compatibility. Classloader depends on the modification of class load method in Java Virtual Machine (JVM) for identifying type parameter, which preserves compatibility in the load time. GJ based on template in C++ provides a generic structure that supported F-Bounded polymorphism, and GJ's compiler is coded by Java itself, which can bring compatibility ahead to the compile time.

However, the best compatibility can be obtained via direct programming in Java without any extra structure, and then what is the most needed is to present a guide on generic programming that can be used in designing compiler of genericity. In [14-15], an implementation approach is put forward for this purpose. Here we only introduce its main idea in brief.

Type parameter is the common interface to a family of specified arguments. If type parameter is represented by abstract class or interface, then the specified arguments are

Table 1. Comparison of Generic Implementation between C++, Delphi and Java

| | C++ | Delphi | Java |
|---|---|---|---|
| Style of programming | OOP+SP* | SP+OOP | OOP |
| Style of implementation | Heterogeneous | Homogeneous | Homogeneous |
| Generic structure | Template | No | No |
| Treatment to primitive type | Direct support | Manual wrapping | Wrapper type |
| Need of type casting | No | Yes | Yes |
| Need of polymorphic method | No | No | Yes |
| Need of inheritance | No | Yes | Yes |
| Providing match checking | No | Yes | Yes |
| Reliability of program | Low | Medium | High |

Note: OOP is abbreviation for Object-Oriented Programming, and SP is for Structural Programming.

considered as its subclass, and the match between them is guaranteed by existing checking on inheritance. Next is to subroutine parameter. Due to nonsupport of subroutine type or pointer, both subroutine and value parameter are to be a part of type parameter class instead, who can be instantiated respectively by the overridden method and constructor in subclass. As to the problem of primitive types mentioned above, Java provides wrapper as substituted classes for them, such as Integer for int, Character for char and so on.

Against its favorite compatibility, the deficiency is that this approach needs to set up a complex hierarchy of classes hard understandable to less experienced programmer, and the possible solution to it is to present a simpler notation for abbreviation and to design a tool for translation. According to this approach, the above example can be designed as following:

```
abstract class Item {
    public static int Size=10;
    public abstract boolean cmp(Item x);
}
class Sort {
    public static void quicksort(Item[] V) {…}
}
class IntItem extends Item {
    int v;
    public boolean cmp(Item x)
        { return this.v<((IntItem)x).v;}
    public IntItem(int x) { v=x;}
}
class test {
    IntItem V[2]={new IntItem(0), new IntItem(1)};
    Sort.quicksort(V);
}
```

where IntItem served as actual argument is subclass of Item, and other parameters, such as Size and cmp, are incorporated with Item as a unified parameter.

### 3.4 Comparison

There exists a viewpoint that generic programming is just to design on topmost class, which means that you can use all kinds of classes as actual arguments to get the most general program. However, it mistakes the key idea of genericity. Genericity is to abstract common code for a family of similar programs, but not for any kinds of programs. Otherwise it would be meaningless. So it gives us the reason why to use the above relatively complex structures to describe a generic program instead to use seemingly simpler ones.

The above three implementations of genericity is dependent of different language features. For instance, C++ provides template structure and subroutine pointer; and Delphi provides object-oriented techniques and subroutine type; meanwhile Java provides pure class hierarchy and referential class variable. These construct the foundation for generic programming. Table 1 gives the details.

## 4. A SIMPLE EXAMPLE: GENERIC STACK

Generic stack is a kind of abstract stack whose element type does not given directly but by a type parameter. If needed, it can be reused to construct various stacks, such as integer stack, float stack or some self-defined type stack and so on. The Figure 1 shows the implementations in three languages.

## 5. CONCLUSIONS

In this paper, generic programming can be implemented in three different languages. What generic programming needs is non-traditional kinds of parameter to behave polymorphicly. For this reason, C++ has template structure to support type parameter, but it doesn't support subroutine parameter; Delphi and Java don't provide any generic

Figure 1. Generic stack implementations in C++, Delphi and Java

| C++ | Delphi | Java |
|---|---|---|
| template<class T><br>class stack {<br>  struct Node {<br>  T v; //use template variable T<br>  Node *next;<br>  };<br>  Node*head=NULL;<br>  void push(T e) {<br>  Node *p=new Node();<br>  p->v=e;<br>  p->next=head;<br>  head=p;<br>  }<br>  void pop() {<br>  Node *p=head;<br>  if(head!=NULL) {<br>  head=head->next;<br>  delete p;<br>  }<br>  }<br>}<br>void main() {<br>  stack<int> s;<br>  s.push(3); s.pop();<br>} | type<br>PNode=^Node;<br>Node=record<br>  v:variant;  //use variant for no limited<br>  next:^Node;<br>end;<br>stack=class<br>  head:PNode=nil;<br>  procedure push(e:variant);<br>  procedure pop();<br>end;<br>var<br>  s:stack;<br>procedure stack.push(e:variant);<br>p: Node;<br>begin<br>  new(p); p^.v:=e;<br>  p^.next:=head;<br>  head:=p;<br>end;<br>procedure stack.pop();<br>p: Node;<br>begin<br>  if(head<>NULL) begin<br>  p:=head; head:=head^.next;<br>  dispose(p);<br>  end;<br>end;<br>begin<br>  s.push(3);s.pop();<br>end. | class stack {<br>  class Node {<br>  Object v;  //use Object for no limited<br>  Node next;<br>  };<br>  Node head=null;<br>  void push(Object e) {<br>  Node p=new Node();<br>  p.v=e; p.next=head;<br>  head=p;<br>  }<br>  void pop() {<br>  if(head!=null)<br>  head=head.next;<br>  }<br>}<br>class test {<br>  public static void main(String args[]) {<br>  stack s=new stack();<br>  s.push(new Integer(3));<br>  s.pop();<br>  }<br>} |

mechanism to support them. It shows the point that even without generic mechanism some languages can utilize their existing features to implement generic programming.

Through this paper, we also introduce some progresses on implementation of generic programming, and exerts ourselves to clarify the key idea of genericity regardless its confusing appearance. Based on it, we would make our further research for more challenging problems in generic programming.


## ACKNOWLEDGEMENT

We would like to thank all the anonymous referees for their helpful comments and suggestions.


## REFERENCES


[1] D. Musser, A. Stepanov. Generic Programming. In Proceedings of the 1st International Joint Conference of ISSAC-88 and AAECC-6. LNCS 358:13~25, Springer-Verlag, 1989

[2] R. Backhouse, P. Jansson, J. Jeuring and L. Meertens. Generic Programming: An Introduction. In S. Swierstra, ed., Advanced Functional Programming, vol. 1608 of LNCS: 28~115. Springer-Verlag, 1999.

[3] John Barnes. Ada 95 Rationale: The languages, the Standard Libraries. LNCS 1247, Springer-Verlag, 1997.

[4] X. Pacheco, S. Teixeira. Borland Delphi 6 Developer's Guide (1st Edition). Published by Sams, 2001.

[5] P. Plauger, A. Stepanov, Meng Lee, D. Musser. The C++ Standard Template Library. Published by Prentice-Hall, 2000.

[6] Andrei Alexandrescu. Moden C++ Design (1st Edition). Addison-Wesley, 2001.

[7] Bruce Eckel. Thinking In Java (3rd Edition). Published by Prentice-Hall, 2002.

[8] K. Thorup. Genericity in Java with Virtual Types. In Proceedings of the 1997 European Conference on Object-Oriented Programming, 1997.

[9] R. Cartwright, G. Steele. Compatible Genericity with Runtime Types for the Java Programming Language. In Proceedings of 13th Annual ACM Conference on OOPSLA, vol. 33: 201~215, 1998.

[10] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language, OOPSLA, 1998.

[11] L. Cardelli, P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys, Vol. 17(4):471-522, 1985.

[12] Maria Barron, Ryan Stansifer. A Comparison of Generics in Java and C#. In Proceedings of 41st ACM SE Regional Conference, 2003.

[13] Keith Turner. Catching more errors at compile time with Generic Java: A proposal to add parametric types to the Java language. Software Engineer, IBM, 2001.

[14] Xue Jinyun, Li Yunqing, and Yang Qinghong. Several New Patterns of Reusable Program Parts. Journal of Computer Research and Development, 1993, 1: 52~57(in Chinese with English Abstract).

[15] Xu Wensheng. The Implementation Approach Research of Generic Programming in Java Language[M.A. Thesis]. Nanchang: Jixangxi Normal University, 2002(in Chinese with English Abstract).