A Comparison of Generic Template Support: Ada, C++, C#, and Java™

Ben Bros gol
brosgol@adacore.com
www.adacore.com

Reliable Software Technologies – Ada-Europe 2010
Valencia, Spain

17 June 2010
Correction to Reference in Proceedings Paper

www1.adacore.com/~bros gol/ae2010/examples.html

Missing tilde
Overview

Introduction

Points of comparison

Example: generic stack in Ada, C++, C# and Java

Generic entities and parameters

Constraints on generic formal parameters

Instantiation and implementation model

Generics and Object-Oriented Programming: Covariance and Contravariance

Conclusions
Why generics?

- Abstraction from specific data structures/algorithms so that they work for a variety of types
  - Type safety, efficiency
- Seminal work: CLU, Alphard in 1970s

Concepts

- **Generic template**, a language construct (e.g., a type, class, module, or subprogram) parameterized by generic formal parameters
- **Instantiation**: compile-time expansion of template, with arguments replacing generic formals

Typical examples

- Generic container (stack, list, etc) parameterized by element type
- Generic sorting algorithm, parameterized by the element type and comparison function

Workarounds in the absence of generics

- Use of overly general type (`Object, void*`), with run-time conversions / type checks
- Macros / preprocessor

But generics are not text macros

- Generic template is checked for syntactic and semantic correctness
- Instantiation is checked (arguments must match generic formal parameters)
- Names in template resolved in scope of template definition, not template instantiation
package Ada.Text_IO is
    type File_Type is limited private;
    ...
    generic
        type Num is mod <>;
    package Modular_IO is
        procedure Put( File : File_Type; Item: Num; ...);
        ...
    end Modular_IO;
end Ada.Text_IO;

with Ada.Text_IO; use Ada.Text_IO;
procedure Carpentry_App is
    type File_Type is (Fine, Coarse, Very_Coarse);
    type Byte is mod 256;
    package Byte_IO is new Modular_IO( Byte );
    -- Byte_IO.Put uses Ada.Text_IO.File_Type, not Carpentry_App.File_Type
    B : Byte := 255;
    File_1 : Ada.Text_IO.File_Type;
    File_2 : File_Type;
begin
    ...
    Byte_IO.Put( File_1, B ); -- OK
    Byte_IO.Put( File_2, B ); -- Illegal
    ...
end Carpentry_App;
Points of Comparison

Expressiveness / basic semantics

• Which entities can be made generic?
• What kinds of formal parameters? Constraints on formal type parameters?
• Rules for instantiation / “contract model”?
• How instantiate: explicit, or implicit?
• Recursive instantiations?

Implementation model

• Expansion-based, or code sharing?
• Any run-time costs?
• When are errors detected?

Feature interactions

• Object-Oriented Programming
  ▪ Inheritance hierarchy for generic types/classes?
  ▪ Covariance/contravariance issue
• Name binding / overload resolution
Example: Generic Stack in Ada

generic
  type Element_Type is private; -- “Constraint”
package Generic_Stack_Pkg is
  type Stack(Max_Size : Natural) is limited private;
  procedure Push(S : in out Stack; Element : in Element_Type);
  procedure Pop (S : in out Stack; Element : out Element_Type);
  generic
    with procedure Display_Element(Element : in Element_Type);
  procedure Display(S : in Stack); -- Display each of the elements
private
  type Element_Array is array( Positive range <> ) of Element_Type;
  type Stack(Max_Size : Natural) is
    record
      Last : Natural := 0;
      Data : Element_Array(1..Max_Size);
    end record;
end Generic_Stack_Pkg;

package body Generic_Stack_Pkg is

with Generic_Stack_Pkg, Ada.Text_IO;
procedure Stack_Example is
  package Integer_Stack_Pkg is new Generic_Stack_Pkg(Integer);
  procedure Put(I : Integer) is ...
  procedure Display is new Integer_Stack_Pkg.Display(Put);
  S : Integer_Stack_Pkg.Stack(100); N : Integer = 1234;
begin
  Integer_Stack_Pkg.Push( S, Element => N);
  Integer_Stack_Pkg.Push( S, Element => "trouble" ); -- Illegal
  Integer_Stack.Display( S );
  N := Integer_Stack.Pop;
end Stack_Example;
// stack.hpp
// Inclusion model: template definition in header file
template<typename T>
class stack{
  public:
    const int MAXSIZE;
    stack(int maxsize) : MAXSIZE(maxsize), data(new T[maxsize]), last(-1){ }
    stack(int maxsize) : MAXSIZE(maxsize), data(new T[maxsize]), last(1){ }
  void push(T t){ last++; data[last] = t; }
  T pop(){ T t = data[last]; last--; return t; }
  template<void(*ref)(T)> // "void ref (T)" displays a T value
  void display(){ for (int i=0; i<=last; i++){ ref(data[i]); } }
  private:
    T* data;
    int last;
};

#include <iostream>
#include "stack.hpp"
extern void put(int i){std::cout << i << std::endl;}

int main() {
  int n = 1234;
  stack<int> s(100);
  s.push(n);
  s.push("trouble"); // Illegal
  s.display<put>();
  n = s.pop();
}
// stack.hpp
// Inclusion model: template definition in header file
template<typename T>
class stack{
    public:
        const int MAXSIZE;
        stack(int maxsize) ;
        void push(T t);
        T pop();

        template<void(*ref)(T)> // "void ref (T)" displays a T value
            void display(){ for (int i=0; i<=last; i++){ ref(data[i]); } }
    private:
        T* data;
        int last;
};
template <typename T> stack<T>::stack(int maxsize) :
    MAXSIZemaxsize), data(new T[maxsize]), last(-1){ }
template <typename T> void stack<T>::push(T t){ last++; data[last] = t; }
template <typename T> T stack<T>::pop(){ T t = data[last]; last--; return t; }

#include <iostream>
#include "stack.hpp"
extern void put(int i){std::cout << i << std::endl;}

int main() {
    int n=1234;
    stack<int> s(100);
    s.push( n );
    s.push( "trouble" ); // Illegal
    s.display<put>();
    n = s.pop();
}
public interface IDisplayable{ void Display(); }

// Need to declare a class or struct Int
// in order to ensure that the Display method is available
public struct Int : IDisplayable{
    public int value;
    public Int(int value){ this.value=value; }
    public void Display(){ System.Console.Write(value + " "); }
}

public class Stack<T> where T : IDisplayable{
    private T[] data;
    public readonly int MAXSIZE;
    private int last=-1;
    public Stack(int maxSize){ MAXSIZE = maxSize; data = new T[maxSize]; }
    public void Push(T t){ last++; data[last] = t; }
    public T Pop(){ T t = data[last]; last--; return t; }
    public void Display(){ for (int i=0; i<=last; i++) data[i].Display(); }
}

public class StackExample{
    public static void Main(){
        int n=1234;
        Stack<Int> stack = new Stack<Int>(100);
        stack.Push( new Int(n) );
        stack.Push( "trouble" ); // Illegal
        stack.Display();
        n = stack.Pop().value;
    }
}
public interface Displayable{ void display(); }

public class Int implements Displayable{ // Wrapper class that provides display method
    public int value;
    public Int(int value){ this.value=value; }
    public void display(){ System.out.print(value + " "); }
}

public class Stack<E extends Object & Displayable>{
    private E[] data;
    public final int MAXSIZE;
    private int last=-1;

    @SuppressWarnings("unchecked")
    public Stack(int maxSize){ MAXSIZE = maxSize; data = (E[])new Object[maxSize]; } // Can't create array of E's so we create array of Objects and do unchecked cast

    public void push(E e){ last++; data[last] = e; }
    public E pop(){ E e = data[last]; last--; return e; }
    public void display(){ for (int i=0; i<=last; i++){ data[i].display(); } }
}

class NastyStackExample{
    public static void main(String[] args){
        int n;
        Stack<Int> stack = new Stack<Int>(100);
        stack.push( new Int(n) );
        stack.push( "trouble" ); // Illegal
        stack.display();
        n = stack.pop().value;
    }
}

public class StackExample{
    public static void main(String[] args){
        int n;
        Stack<Int> stack = new Stack<Int>(100);
        stack.push( new Int(n) );
        stack.push( "trouble" ); // Illegal
        stack.display();
        n = stack.pop().value;
    }
}

But:

class NastyStackExample{
    public static void main(String[] args){
        int n;
        Stack<Int> stack = new Stack<Int>(100);
        stack.push( new Int(n) ); // OK
        Stack s = stack; // raw type
        s.push( "trouble" ); // Legal
        n = stack.pop().value; // Exception
    }
}
### Generic Entities and Parameters

#### Generic units
- Packages
- Subprograms

#### Generic formal parameters
- Types
- Subprograms
- Objects
- Instances of generic packages

#### Generic entities
- Types
  - Classes, interfaces, structs, delegates
- Methods

#### Generic formal parameters
- Types

---

#### Templates
- Classes and structs
- Functions

#### Template parameters
- Types
- Constant values *(non-type parameter)*
  - Integral, enumeration, pointer, or reference type
- Templates

---

#### Generic entities
- Classes
- Interfaces
- Methods
- Constructors

#### Generic formal parameters
- Types
Constraints on Generic Formal Types

- Numeric type categories
- Discrete types
- Array types
- Access type categories
- Derived types
- Types with assignment, “=”
  - Whether unconstrained objects allowed

No constraints

- If specific operation needed, supply as pointer-to-function generic formal

- Whether reference or value
- Whether it derives from some specific base class or interface(s)
- Whether it has an accessible no-arg constructor
  Use reflection to enforce other preconditions

- Whether it extends some specific superclass or implements some specific interface(s)
  Use reflection to enforce other preconditions

Ada

C++

C#

Java
Instantiation-Related Issues

Instantiation always explicit

**Legality**
- “Contract model”

**Expansion / code sharing**
- Almost always expansion
- Explicit instantiation provides programmer control over sharing

Instantiation implicit or explicit

**Legality**
- Constant for non-type parameter
- Argument type may lack operation used by the template (error on usage)

**Inclusion or separation model**

**Expansion / code sharing**
- Share instances with same arguments

Instantiation implicit

**Legality: parameter matching**

**Expansion / code sharing**
- All instantiations with reference types share single code body
- Instantiations with different value types have separate expansions
- All instantiations with same value type shares same code body

Instantiation implicit

**Legality: parameter matching**

**Expansion / code sharing**
- Class and interface types only
- All instantiations with same value type share one copy of statics
- Full code sharing (“type erasure”)
- In effect, generic formal parameter replaced by Object or 1st extends bound, and casts are inserted
Variance concepts

- Covariance: the ability to use a subclass where a class is required
- Contravariance: the ability to use a superclass where a class is required

Variance in the context of generic types

- Consider a generic type G<T>, parameterized by type T
- If class T2 is a subclass of / derives from T1:
  - Covariance: the ability to use a G<T2> where a G<T1> is required
  - Contravariance: the ability to use a G<T1> where a G<T2> is required

\[
\begin{align*}
G<T1> & \quad gt1 = ...; & \quad T1 & \quad \text{void M(G<T1> g1){...}} \\
G<T2> & \quad gt2 = ...; & \quad G<T1> & \quad M()\{...\} \\
gt1=gt2; & \quad \text{// Covariant} & \quad M(gt2); & \quad \text{// Covariant} \\
gt2=gt1; & \quad \text{// Contravariant} & \quad gt2 = M(); & \quad \text{// Contravariant}
\end{align*}
\]

Neither covariance nor contravariance for generics are safe in general

- Example: generic container class Sack<T>, and classes Animal, Fish, Shark
- Assigning a Sack<Fish> to a Sack<Animal> or to a Sack<Shark> could violate type safety

But in some contexts, one or the other may be useful and safe

- Covariance OK if you can only output from gt1 (as a G<T1>) after assigning from gt2
- Contravariance OK if you can only input into gt2 (as a G<T2>) after assigning from gt1
Covariance

Assignment (for references to objects that are instances of generic types) is not covariant

```java
zoo = aquarium; // Covariant?
// If the above were permitted, then the following would be a problem:

Elephant jumbo = new Elephant();
zoo.AddElement(jumbo); // OK for zoo, but not for aquarium
```

Analogous problem on parameter passing
Assignment could be covariant if there is no way to input Animal objects into zoo afterwards
• For example: reference zoo through an interface that can output but not input elements
  • Thus you can remove elements (as Animal) but not add Animal objects
Assignment (for references to objects from instances of generic types) is not contravariant

```javascript
sharktank = aquarium; // Contravariant?
// Obviously unsafe, since could reference a general Fish as a Shark
```

Assignment could be contravariant if there is no way to output sharktank elements as Shark
- For example: reference sharktank through an interface that can input but not output elements
- Thus you can add Shark objects but not reference existing elements
Generic covariance and contravariance are new in C# 4

```csharp
interface IInputable<in T>{
    void AddElement(T t);
}

interface IOutputable<out T>{
    T RemoveElement();
}

class Sack<T>:IInputable<T>, IOutputable<T>{
    public void AddElement(T t){...}
    public T RemoveElement(){...}
}

IOutputable<Animal> outzoo;
IInputable<Shark> insharktank;

Sack<Animal> zoo = new Sack<Animal>();
zoo.AddElement(new Elephant());
zoo.AddElement(new Donkey());

Sack<Fish> aquarium = new Sack<Fish>();
aquarium.AddElement(new Fish());
aquarium.AddElement(new Fish());

zoo = aquarium;    // Illegal
outzoo = aquarium;    // Covariance
// Can't use outzoo to insert Animals
Animal a = outzoo.RemoveElement();    // OK

Sack<Shark> sharktank;
sharktank = zoo;    // Illegal
insharktank = zoo;    // Contravariance
// Can't use insharktank to remove Sharks
insharktank.AddElement( new Shark() );    // OK
```

Generic covariance and contravariance are new in C# 4
Object-Oriented Programming and Generics

**Generic packages may form hierarchy**
- Type in child can derive from tagged type in parent

**Covariance and contravariance**
- Issue does not arise

**Ada**

- Class templates may form inheritance hierarchy
  - Covariance and contravariance are not supported

**C++**

- **Generic types may form inheritance hierarchy**
- **Covariance and contravariance**
  - For interfaces, delegates
  - Covariance: “out” type used as method result
  - Contravariance: “in” type used as value parameter
  - Works only for reference types

**C#**

**Java**

- **Generic types may form inheritance hierarchy**
- **Covariance and contravariance**
  - Methods can use “wildcards” for parameters, result types
  - $G<? extends T>$ allows $G<U>$ where $U$ is a subclass of $T$ (covariance)
  - $G<? super T>$ allows $G<U>$ where $U$ is a superclass of $T$ (contravariance)
**Ada**
- Separation of class into type + module
- Early error detection (“contract model”)
- Most general set of generic formal parameters, constraints on generic formal types

**C++**
- Deferred error detection
- Flexibility, “metaprogramming”
- Separate compilation issues

**C#**
- Partial solution to code sharing
- Covariance, contravariance for interfaces and delegates
- Instantiation = IL expansion

**Java**
- Type erasure / upwards compatibility
  - Anomalies such as shared static data, inability to construct arrays of formal generic type
- Code sharing
- Covariance, contravariance through “wild-cards”