An introduction to Generics: containers

Michaël Van Canneyt

December 27, 2021

Abstract

Arguably one of the biggest uses of generics is for container classes: lists, collections, dictionaries. In this article we'll explain how to use generics for such classes.

1 Introduction

Every Object Pascal programmer sooner or later comes in contact with the classes unit or the contnrs unit. It contains some container classes: `TList`, `TStringList`, `TCollection`, `TObjectList` `TStack` and `TQueue`.

These are all examples of container classes, which are used to maintain lists or collections of objects. In general, they work well in daily use cases, for specialized use it is probably better to write your own.

There are 2 disadvantages to these classes:

1. They work only with pointers or classes.
2. When creating a descendent, one almost invariably ends up doing typecasts.

Let us illustrate this with an example: Assume a class `TCar`, and we want to create a list of cars : `TCarList`, where the default array property is of type `TCar`.

Nothing could be simpler:

```
uses contnrs;

Type
  TCar = Class(TObject)
    Brand : String;
    Seats : Integer;
  end;

  TCarList = Class(TObjectList)
    Function GetC (AIndex : Integer): TCar;
    Procedure SetC (AIndex : Integer; AValue : TCar);
  Public
    Property Cars[AIndex] : TCar Read GetC Write SetC;default;
  end;
```

However, the uglyness is in the implementation:

```
Function TCarList.GetC (AIndex : Integer): TCar;
```

```
begin
  Result:=TCar(Items[AIndex]);
end;
```

A typecast is necessary to cast the inherited `Items` property (of type `TObject`) to `TCar`. It also requires a call to an inherited method.

The setter is less problematic, as no typecast is needed:

```
Function TCarList.SetC (AIndex : Integer; AValue: TCar);
begin
  Items[AIndex]:=AValue;
end;
```

Ideally, it should not be necessary to have to call an inherited method.

It is also clear that `TList` or `TObjectList` cannot be used with arbitrary records or arrays.

2 Generics to the rescue

The solution to the problem outlined above would of course be that the element type for `TList` can be specified when declaring a list. This is of course exactly what generics are for.

We'll attempt to create a generic list class, and see what kind of problems pop up when trying to implement it.

In its simplest form, a list class is of course nothing more than an Array, with some methods attached to it. So, basically a list class would look like this:

```
unit list1;

interface

Type
  TList<ET> = Class(TObject)
  Private
    FItems : Array of ET;
    Function GetE(AIndex : Integer) : ET;
    Procedure SetE(AIndex : Integer; AValue : ET);
  Public
    Property Items [AIndex : Integer] : ET Read GetE Write SetE; default;
  end;

Implementation

Function TList<ET>.GetE(AIndex : Integer) : ET;

begin
  Result:=FItems[AIndex];
end;

Procedure TList<ET>.SetE(AIndex : Integer; AValue : ET);
```

```
begin
  FItems[AIndex]:=AValue;
end;

end.
```

The getter and setter do not contain any code for checking the validity of the index, for simplicity.

It is clear from the code that the access to the elements in the array is direct, and that no typecasts are necessary because the elements are kept in a dynamic array of the correct type:

```
    FItems : Array of ET;
```

To use this, one would write something like this:

```
Type
  TCar = record
    Brand : String;
    Seats : Smallint;
  end;
  TCarList = TList<TCar>;
```

Or declare a variable as:

```
Var
  Cars : TList<TCar>;
```

At the first use of this class, it would result in an access violation, as there is no way to add elements to the array: the setter can only set the items at an existing index in the array, and the length of the array is zero to begin with.

To remedy this, we add a method called `Add` to our class:

```
Procedure Add(AValue : ET);
```

With implementation:

```
Procedure TList<ET>.Add(AValue : ET);

Var
  L : Integer;

begin
  L:=Length(FItems);
  Setlength(FItems,L+1);
  FItems[L]:=AValue;
end;
```

This will not make for the fastest `TList` implementation, but it serves the purpose. The code is nothing special: if it was not for the type template `ET`, there is little to indicate that this is a generic method.

Now the generic list class becomes usable:

```
 Var
  MyCars : TList<TCar>;
  C : TCar;

begin
  MyCars:=TCarList.Create;
  try
    C.Brand:='BMW';
    C.Seats:=2;
    MyCars.Add(C);
    With MyCars[0] do
      Writeln('Brand: ',Brand,', Seats:', Seats);
  Finally
    MyCars.Free;
  end;
end.
```

This will work as expected: it will print the name of a German car brand.

3 Initializing/finalizing variables of unknown type

Adding an element to the list presents no difficulties, we hardly notice the use of generics. How about inserting an element in the list at some arbitrary position ?

To check, we add a method called `Insert` to our class:

```
Procedure Insert(AIndex : Integer; AValue : ET);
```

With implementation

```
Procedure TList<ET>.Insert(AIndex : Integer; AValue : ET);

Var
  L : Integer;

begin
  L:=Length(FItems);
  Setlength(FItems,L+1);
  Move(FItems[AIndex],FItems[Aindex+1],SizeOf(ET)*(L-Aindex));
  FItems[AIndex]:=AValue;
end;
```

Again, nothing special in this code, except the use of `SizeOf`: as can be seen, it can be used with template types in generics: when specializing, the compiler knows the size of the type used to specialize the list with.

The opposite method of insert, `Delete`, requires a bit more care:

```
Procedure Delete(AIndex : Integer);
```

With implementation:

```
Procedure TList<ET>.Delete(Aindex : integer);
```

```
Var
  L : Integer;

begin
  FItems[Aindex]:=Default(ET);
  L:=Length(FItems);
  Move(FItems[AIndex+1],FItems[Aindex],SizeOf(ET)*(L-1-Aindex));
  Dec(L);
  FillChar(FItems[L], SizeOf(ET), 0);
  SetLength(FItems,L);
end;
```

There are 2 tricky parts in this code.

The first line is needed to make sure that the item to be deleted is properly finalized: if we would omit this line, managed types in `ET` such as strings (or `ET` itself, if it is a managed type) would end up having wrong reference counts, resulting in memory leaks or access violations.

For this, the `Default` compiler intrinsic is used. This function can be represented as:

```
Function Default(T : AType) : AType;
```

it produces an empty value for the type `T` that is passed to it. In essence, it returns

```
  Result:=AType(FillChar(Result,SizeOf(AType),0));
```

By assigning an empty value to `FItems[AIndex]`, we ensure that it is properly cleared. Only then we move all the other values in the array to a lower position in the array.

For the last element in the array, we do not want to properly clear it. Instead, we just want to zero it out, so no finalization can occur when the array is cleared or resized. For this, we need the

```
  FillChar(FItems[L], SizeOf(ET), 0);
```

statement.

4 IndexOf and Sort: Comparing elements

Often, one needs to find the position of an element in the list (e.g. in order to remove it using delete). Finding the position is usually handled with `IndexOf`:

```
function IndexOf(const AValue: ET): Integer;
```

A naive implementation would be:

```
Function TList<ET>.IndexOf(Const AValue : ET) : Integer;

begin
  Result:=Length(FItems)-1;
  While (Result>=0) and Not (AValue=FItems[Result]) do
    Dec(Result);
end;
```

However, the compiler will complain about the comparison, telling us that it doesn't know how to compare cars:

```
list5.pas(30,36) Error: Operator is not overloaded: "TCar" = "TCar"
```

In Free Pascal, the solution is to implement an operator overload of the "=" operatior for the `TCar` type. In Delphi, this is not sufficient. Instead, a comparer interface is needed. The comparer interface is defined in the Generics.Defaults unit (also supplied with Free Pascal):

```
IComparer<T> = interface
  function Compare(constref Left, Right: T): Integer; overload;
end;

TComparer<T> = class(TInterfacedObject, IComparer<T>)
public
  class function Default: IComparer<T>; static;
  function Compare(constref ALeft, ARight: T): Integer; virtual; abstract; overlo
end;
```

The `Default` class function of `TComparer` will create a comparer interface that checks equality of 2 elements of the type `T`. For brevity, 2 class functions were omitted from the `TComparer` class, we'll get back to them further in the text.

Using this interface, we can implement our `IndexOf` function to compare 2 cars:

```
 Function TList<ET>.IndexOf(Const AValue : ET) : Integer;

Var
  C : IComparer<ET>;

begin
  C:=TComparer<ET>.Default;
  Result:=Length(FItems)-1;
  While (Result>=0) and (C.Compare(AValue,FItems[Result])<>0) do
    Dec(Result);
end;
```

The following will then work:

```
program cars5;

uses generics.defaults,list5;

Type
  TCar = record
    Brand : String;
    Seats : Smallint;
  end;
  TCarList = TList<TCar>;

Var
  MyCars : TList<TCar>;
  C1,C2 : TCar;
```

```
begin
  MyCars:=TCarList.Create;
  try
    C1.Brand:='BMW';
    C1.Seats:=2;
    MyCars.Add(C1);
    C2.Brand:='Peugeot';
    C2.Seats:=4;
    MyCars.Add(C2);
    MyCars.Delete(0);
    Writeln('Peugeot : ',MyCars.IndexOf(C2));
    Writeln('BMW     : ',MyCars.IndexOf(C1));
  Finally
    MyCars.Free;
  end;
end.
```

It will nicely print

```
Peugeot : 0
BMW     : -1
```

It first sight, all is well implemented, but unfortunately, this is not completely the case.

For example, the following will not work:

```
MyCars.Add(C2);
MyCars.Delete(0);
UniqueString(C2.Brand);
Writeln('Peugeot : ',MyCars.IndexOf(C2));
Writeln('BMW     : ',MyCars.IndexOf(C1));
```

this code prints

```
Peugeot : -1
BMW     : -1
```

The code reports that no car brand `Peugeot` can be found !

The reason is that the default comparer will compare 2 records byte by byte. The first field of `TCar` (Brand) is a string, which in essence is a pointer to an array of characters. This means that comparing the records byte by byte will compare 2 pointers. The `UniqueString` makes sure the pointer to the characters `Peugeot` is unique, hence the default comparer is returning false, even though the value of the string is the same.

To solve this properly, we need in essence to provide a proper comparer. This can be done with the `TComparer.Construct` class function, which we omitted from the earlier declaration:

```
Type
  TComparisonFunc<T> = function(constref Left, Right: T): Integer;

  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>; static;
    function Compare(constref ALeft, ARight: T): Integer; virtual; abstract; over
    class function Construct(const AComparison: TComparisonFunc<T>): IComparer<T>
  end;
```

`Construct` takes a function which compares 2 variables of type `T`, and constructs an `IComparer` interface for the type `T`. (the same can be done with a method instead of a plain function)

Thus, if we write a function

```
Function CompareCars(constref C1,C2 : TCar) : integer;
begin
  if C1.Brand<C2.Brand then
    Result:=-1
  else if C1.Brand>C2.Brand then
    Result:=1;
  else
    Result:=C1.Seats-C2.Seats;
end;
```

We can construct a comparer as follows:

```
  CC : IComparer<TCar>;

begin
  CC:=TComparer<TCar>.Construct(@CompareCars);
end;
```

. We still need somehow to pass the comparer to the `IndexOf` function. One way of doing this would be to pass it to the `IndexOf` function:

```
function IndexOf(const AValue: ET; AComparer : IComparer<ET>): Integer;
```

But this is cumbersome to call. Also, looking ahead it is reasonable to assume that we would add a `Sort` method to the list, and that will also need a comparer interface. So, it is better to add it as a field to the class, and pass it on in a constructor:

```
Type
  TList<ET> = Class(TObject)
  Private
    FComparer : IComparer<ET>;
  Public
    Constructor Create; overload;
    Constructor Create(AComparer : IComparer<ET>); overload;
  end;
```

(other fields, methods and properties have been emitted for brevity). The implementation of the constructors is quite simple:

```
Constructor TList<ET>.Create(AComparer : IComparer<ET>);

begin
  FComparer:=AComparer;
end;

Constructor TList<ET>.Create;
```

```
begin
  FComparer:=TComparer<ET>.Default;
end;
```

The result of this code (using 2 overloaded constructors) is that we can construct a list with a default comparer (suitable for simple types such as integers, enumerators etc), or we can pass on a comparer of our own, as shown in the following example:

```
 program cars6;

uses generics.defaults,list6;

Type
  TCar = record
    Brand : String;
    Seats : Smallint;
  end;
  TCarList = TList<TCar>;

Function CompareCars(constref C1,C2 : TCar) : integer;
begin
  if C1.Brand<C2.Brand then
    Result:=-1
  else if C1.Brand>C2.Brand then
    Result:=1
  else
    Result:=C1.Seats-C2.Seats;
end;

Var
  MyCars : TList<TCar>;
  C1,C2 : TCar;
  CC : IComparer<TCar>;


begin
  CC:=TComparer<TCar>.Construct(@CompareCars);
  MyCars:=TCarList.Create(CC);
  try
    C1.Brand:='BMW';
    C1.Seats:=2;
    MyCars.Add(C1);
    C2.Brand:='Peugeot';
    C2.Seats:=4;
    MyCars.Add(C2);
    MyCars.Delete(0);
    Writeln('Peugeot : ',MyCars.IndexOf(C2));
    Writeln('BMW     : ',MyCars.IndexOf(C1));
    C1.Brand:='Peugeot';
    C1.Seats:=4;
    UniqueString(C1.Brand);
    Writeln('Peugeot (2) : ',MyCars.IndexOf(C1));
  Finally
```

```
    MyCars.Free;
  end;
end.
```

A nice side effect of using a comparer is that we can construct 2 different lists, once with the compare above, and for instance once with a case insensitive comparer:

```
Function CompareCarsCI(constref C1,C2 : TCar) : integer;
begin
  Result:=CompareText(C1.Brand,C2.Brand);
  If Result=0 then
    Result:=C1.Seats-C2.Seats;
end;
```

Using this comparer, the `IndexOf` function will be case insensitive.

5 Conclusion

In this article, we discussed a use case for generics which is quite common: creating a container class. In implementing a sample container class, we encountered 2 common problems; initializing a variable of unknown type, and comparing 2 variables of unknown type. The solution to the first problem led us to the use of the `Default` function, the solution of the second problem led us to the use of the `IComparer` interface and the `TComparer` class. Luckily, one does not need to implement such container classes manually: they are implemented in the `generics.collections` unit.