An introduction to generics

Michaël Van Canneyt

December 27, 2021

Abstract

Generics have been available in Free Pascal and in Delphi for quite some time now, and are increasingly used thoughout the VCL. In this article, we take a closer look at this language construct.

1 Introduction: the problem

Object Pascal is a strongly typed language. This is one of the good reasons for using Object Pascal, since it ensures that if the code compiles, it will not contain too many obvious mistakes that can happen in languages with e.g. dynamic typing, such as passing a string or an object to a function that expects an integer. The compiler will catch such an error at compile time. In dynamically typed languages, such an error will only be detected when running the code.

It also presents a drawback. In case you want to have a structure or function that needs to be implemented for several types, then you need to implement it separately for each type, even if the actual code for the function looks 100% the same.

For instance:

```
Function Min(A,B : Integer) : Integer;

begin
  If A<B then
    Result:=A
  else
    Result:=B;
end;
```

This function can be used with an integer type (in fact, all integer-like types such as byte, word etc.). But it cannot be used with a string. In Javascript, the function

```
function min (a,b) {
  if (a<b) {
    return a;
  } else {
    return b;
  }
}
```

can be used with all basic types (all types for which < is defined); you need to write it only once. In Object Pascal, you need to write the function for each type that you wish to use it with, the overloading feature of the language takes care of selecting the right version when it is used.

When implementing the above function Min(), even though the code inside the function will look 100% the same, the function must be written for each type separately.

2 The solution: Generics

If we want to reduce the number of functions that we must implement, then basically, what is required is a function in which the type of the parameters and return value is a parameter itself, so it can be specified at a later time, and let the compiler generate the function for us, with the type we specify.

Something like

```
Function Min(A,B : T) : T;

begin
  If A<B then
    Result:=A
  else
    Result:=B;
end;
```

Where `T` is unknown.

This requirement, to be able to specify the type of some fields or function parameters later on, extends to structures: `TList` from the classes unit (or `TCollection`) is a classical example.

The `TList` type can be used with pointers. Using some typecasting, it is possible to make descendents that accept objects: Given a class `TMyClass` one can make a `TMyClassList` descendent of `TList` with a default array property that has type `TMyClass`. Some overloads are needed with typecasts, but it works.

With generics, it is possible to define a list class in which the type of the elements is not yet determined, but which can be filled in later. The statement where the actual type is determined, is called *instantiating* in Delphi, or *specializing* in Free Pascal.

Roughly, this is something like:

```
TList<T> = Class
  Property Items [Aindex : Integer] : T Read GetA;
end;
```

Here, `T` would be a placeholder for a type.

Generic types (and functions in FPC) are the solution to this problem.

Generic types can be made using records (with or without methods), classes and arrays and procedural types. In Free Pascal, it is also possible to create actual generic functions or procedures.

The compiler implementation of generics in Free Pascal and Delphi is presumably quite different, which means that some things which are possible in Free Pascal, will not be possible in Delphi. Since Free Pascal aims to be Delphi compatible, the opposite should not be the case (barring bugs in Free Pascal, obviously).

3 Declaring generics

A generic type is first and foremost a type, and as such must be declared in the `Type` declaration block of a unit or a class or record.

The template types (i.e. the types that are still unknown) are indicated between single brackets < and > after the type name.

A generic type can have one or more template type identifiers, separated by a comma.

In the following, `TKey` and `TItem` are placeholders for types:

```
type
  TDictionaryItem<TKey,TItem> = record
    FKey: TKey;
    FItem: TItem;
  end;
```

The same can be done for a class:

```
type
   TDictionaryItem<TKey,TItem> = class
     FKey: TKey;
     FItem: TItem;
   end;
```

It is also possible to use generic types in other generic types:

```
type
  TDictionaryItem<TKey,TItem> = record
    FKey: TKey;
    FItem: TItem;
  end;

  TDictionary<TKey,TItem> = class
    FList : TArray<TDictionaryItem<TKey,TItem>>;
    // Alternatively:
    // FList : Array of TDictionaryItem<TKey,TItem>;
    Function Find(AKey : TKey) : TItem;
   end;
```

Note that the parameters must be repeated in the type name in the implementation:

```
Function TDictionary<TKey,TItem>.Find(AKey : TKey) : TItem;
```

It is also possible to declare a generic procedural type:

```
Type
  TCompareProcedure<T> = Function(A,B : T) : Integer;
```

Lastly, for Free Pascal, functions or procedures can also be generic, which means the following is a valid generic function definition:

```
Function Min<T> (A,B : T) : T;
```

4 Generic arrays

Generic arrays are meant to solve a type assignment compatibility problem: In Delphi, 2 variable of array types are only assignment compatible if they are the same array type.

This means that the following will not compile:

```
Type
  TA = Array of Integer;
  TB = Array of Integer;

Var
  A : TA;
  B : TB;

begin
  A:=B;
end;
```

Even though the elements of `TA` and `TB` have the same type, the arrays will not be assignment compatible.

Note: In Free Pascal, the above will compile and work correctly. This is one of the subtle differences between the 2 compilers.

The problem can be solved with a generic array. The following will compile.

```
Type
  TArray<T> = Array of t;

Var
  A : TArray<Integer>;
  B : TArray<Integer>;

begin
  A:=B;
end;
```

Inventing the generic array may be a bit superfluous, since you need only one `TArray<T>` declaration to be able to create assignment compatible arrays for all possible situations.

In fact, `TArray` is defined in the `SysUtils` unit.

```
Type
  TArray<T> = Array of t;
```

5 Instantiating generic types

When a generic type or function must actually be used using a specific type, i.e. the types for the various templates are known, it needs to be instantiated. (or specialized, in Free Pascal parlance).

This is done by specifying an actual type for the parameter:

```
type
  TStringDictionaryItem = TDictionaryItem<integer,string>;
```

Here we specially created a new type based on the generic type.

Explicitly declaring a new type is not a necessity, it can also be done when declaring a variable:

```
Var
  MyDictItem : TDictionaryItem<string,tobject>;
```

When using a generic type in another generic type, as in the following;

```
type
  TDictionaryItem<TKey,TItem> = record
    FKey: TKey;
    FItem: TItem;
  end;

  TDictionary<TKey,TItem> = class
    FList : TArray<TDictionaryItem<TKey,TItem>>;
    // Alternatively:
    // FList : Array of TDictionaryItem<TKey,TItem>;
    Function Find(AKey : TKey) : TItem;
   end;
```

the type `TDictionaryItem` is not yet instantiated in the `TDictionary` declaration. it will only be instantiated when the generic `TDictionary` type is instantiated.

Note that the compiler will generate actual code only when it encounters an instantiation (or specialization) of a generic. It will try to make sure that it includes the code for a instantiation with a certain type only once.

That means that for

```
var
  A : TDictionary<string,integer>;
  B : TDictionary<string,integer>;
```

The code for class `TDictionary<string,integer>` will be included only once in the final binary.

6 Writing generic code

The code that can appear in a generic class or type is not different from other Object Pascal code as encountered in records or classes. It works just as other code, with the caveat that the actual type of template types is unknown.

Because Object Pascal is a strongly types language, this means that some constructs or pascal statements are not possible, because the type of the template types is unknown during the declaration of the generic class.

Assume the following declaration:

```
Type
  TUtil<T> = record
    function Min(A,B : T) : T;
  end;
```

then the following code will fail to compile in Delphi:

```
function TUtil<T>.Min(A,B : T) : T;

begin
  If A<B then
    Result:=A
  else
    Result:=B;
end;
```

Note that in the implementation of the function, the type placeholders were repeated in the function header.

However, the Delphi compiler will complain on this code:

```
[dcc32 Error]  E2015 Operator not applicable to this operand type
```

The Delphi compiler does not know what to do with the comparison operator <. Note: Because Free Pascal works fundamentally different, the FPC compiler will compile this; It may, however, give an error during specialization (instantiation) of the generic type.

The only allowed operation for identifiers of type T, at this point, is an assignment, i.e. the following will compile:

```
function TUtil<T>.Min(A,B : T) : T;

begin
  Result:=B;
end;
```

The compiler can check that the `Result` and `B` identifiers have the same type, even though the actual type is unknown. So this is valid code.

There are more things that are not allowed, for instance typecasts.

```
function TUtil<T>.Min(A,B : T) : T;
begin
  If Integer(A)<Integer(B) then
    Result:=A
  else
    Result:=B;
end;
```

The Delphi compiler will refuse this, because it cannot determine whether the typecast is valid for the template type `T`.

Similarly, the Delphi compiler will complain if an attempt is made to use methods or fields of the type T:

```
function TUtil<T>.Min(A,B : T) : T;
begin
  If A.AsInteger<B.AsInteger then
    Result:=A
  else
    Result:=B;
end;
```

The compiler cannot resolve `AsInteger`, so it will complain:

```
[dcc32 Error] E2003 Undeclared identifier: 'AsInteger'
```

7 Helping the compiler out: Generic Type restrictions

Some of the problems caused by the fact that the template types are unknown, can be remedied with type constraints:

The compiler can be told that a template type must at least be of a certain class or interface. Any attempt to instantiate a generic class with a template type that is not a descendent of this class or does not implement the specified interface, will result in a compiler error.

By doing this, the compiler can for instance verify code that accesses methods or fields of the template.

Specifying a type constraint is done in the same manner as specifying a type for a parameter in a function or procedure:

```
Type
  TUtil<T : TMyClass> = record
    function Min(A,B : T) : T;
  end;
```

This means that the `TUtil<T>` can only be instantiated with `TMyClass` or a descendent of `TMyClass`.

Using this, the following code will work:

```
Type
  TMyClass = Class
    Function AsInteger : Integer; virtual; abstract;
  end;

  TUtil<T : TMyClass> = record
    function Min(A,B : T) : T;
  end;

  TMyUtilRec = TUtil<TMyClass>;

function TUtil<T>.Min(A,B : T) : T;
begin
  If A.AsInteger<B.AsInteger then
    Result:=A
  else
    Result:=B;
end;
```

This works because the compiler can now verify that `A.AsInteger` is valid, since `T` is guaranteed to be of type `TMyClass`.

Note that

```
Type
  TMyOtherClass = Class
    Function AsInteger : Integer;
  end;

  TMyUtilRec = TUtil<TMyOtherClass>;
```

Will not work, the compiler will complain:

```
[dcc32 Error]  E2515 Type parameter 'T' is not compatible with type 'TMyClass'
```

Even though `TMyOtherClass` has a method `AsInteger`.

Type restrictions follow the same rules as parameter declarations, they can be grouped:

```
Type
  TUtil<T1,T2 : TMyClass> = record
    function Min(A,B : T2) : T1;
  end;
```

and of course in the case of multiple template types, they can have different constraints:

```
Type
  TUtil<T1 : TMyClass; T2 : ISomeInterface> = record
    function Min(A,B : T2) : T1;
  end;
```

Additionally, it is possible to specify multiple constraints for the same type:

```
Type
  TUtil<T : TMyClass, ISomeInterface> = record
    function Min(A,B : T) : T;
  end;
```

This means that `T` must be both a descendent of `TMyClass` and implement the `ISomeInterface` interface when instantiating the `TUtil` generic.

This means the following code is valid for the generic record:

```
Type
  TMyClass = Class
    Function AsInteger : Integer;
  end;

  ISomeInterface = Interface
    Function AsString : String;
  end;

  TUtil2<T : TMyClass, ISomeInterface> = record
    function Min(A,B : T) : T;
  end;

function TUtil2<T>.Min(A,B : T) : T;
begin
  If (A.AsInteger<B.AsInteger) or (A.AsString<B.AsString) then
    Result:=A
  else
    Result:=B;
end;
```

It is possible to specify simply that a template type must be a class:

```
Type
  TUtil2<T : class> = record
```

```
    function Min(A,B : T) : T;
  end;

function TUtil2<T>.Min(A,B : T) : T;
begin
  If B.InheritsFrom(A) then
    Result:=A
  else
    Result:=B;
end;
```

The same can be done for a record.

Needless to say, a class and record constraint cannot be combined, because a type cannot be a class and a record at the same time.

8 Back to the original problem

Are we now any closer to our original problem, the `Min` function usable with any type ?

Yes and no. In Free Pascal, the following simply works:

```
Type
  Trec = Record
    A : Integer;
    // Define the < operator for TRec.
    Class Operator LessThan (A,B : TRec) : Boolean;
  end;

// Implement it.
Class Operator TRec.LessThan (A,B : TRec) : Boolean;

begin
  Result:=A.A<B.A;
end;

// Now our generic function
Function Min<T> (A,B : T) : T;

begin
  If A<B then
    Result:=A
  else
    Result:=B;
end;

// Finally we demonstrate everything:
Var
  A,B : Integer;
  C,D : Trec;

begin
  A:=1;
  B:=2;
  A:=Min<Integer>(A,B);
  Writeln(A);
  C.A:=4;
  D.A:=3;
  C:=Min<TRec>(C,D);
  Writeln(C.A);
end.
```

This is because the Free Pascal compiler operates different from Delphi. When it instantiates the function `Min<TRec>`, it "replays" the code for `Min` using `TRec` as the type. During replay, it encounters `A<B`. Since it knows the operator `LessThan` for the type `TRec` it can compile the code. The above will not work for all types, since e.g. for classes, operators cannot be defined.

In Delphi, some extra work is necessary but in essence the above can be achieved as well; The following will also work in Free Pascal.

The first problem is that Delphi does not support generic functions. This can be remedied by using a static function in a record.

```
Type
  TUtils = Record
    Class Function Min(A,B : T ) : T; static;
  end;

Class Function TUtils.Min(A,B : T) : T;

begin
  if A<B then
    Result:=A
  else
    Result:=B;
end;
```

But this still does not compile, since the compiler still cannot handle the `A<B`.

The solution is using a template interface - which is defined in the `System.Generics.Defaults` unit:

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

This interface introduces a function that should return a negative result if `Left<Right`, a positive result if `Left>Right` and zero if `Left=Right`.

Using this, we can rewrite the function:

```
  TUtils2 = Record
    Class Function Min<T>(A,B : T; Cmp : IComparer<T>) : T; static;
  end;
```

With implementation

```
Class Function TUtils.Min<T>(A,B : T; Cmp : IComparer<T> ) : T;
```

```
begin
  if (Cmp.Compare(A,B)<0) then
    Result:=A
  else
    Result:=B;
end;
```

Now, to create and pass on a `IComparer` interface for all basic types would be tedious and a lot of work.

The `TComparer` class (defined in the same unit) will create such an interface for all basic types automatically, allowing us to use our new function as:

```
  A:=TUtils2.Min<Integer>(A,B,TComparer<Integer>.Default);
```

It is still somewhat annoying that the comparer interface must be specified each time. This can be solved - for basic types - with the following definition of our record:

```
Type
  TUtils<T> = Record
    Class Var C : IComparer<T>;
    Class Constructor Create;
    Class Function Min(A,B : T ) : T; static;
  end;

Class Constructor TUtils<T>.Create;

begin
  C:=TComparer<T>.Default;
end;
```

The class constructor is called once for each instantiation of this generic record, and will store a correct `IComparer` interface for each record.

This means we can do

```
 A:=TUtils<Integer>.Min(A,B);
```

This comes pretty close to what we actually wanted to achieve.

9 Conclusion

Clearly, for the simple case demonstrated here, it is questionable whether it is worth using generics. A simple set of overloaded functions may well be sufficient. However, for more complicated cases the use of generic types can offer many advantages. This will be demonstrated in a future contribution.