Recent advanced in Compiler

Const parameters

The ‘const’ keyword for functions and procedures is now implemented. It protects the parameter from being updated within the procedure or function.

Default parameters

Example

function MyInc(x : integer; increment : integer = 1) : integer;

The second parameter here defaults to 1. That is the programmer may use the function with one or two parameters. If only one parameter is passed the default value is used for the second parameter.

Not supported for complex parameters like arrays or records, nor for var parameters.

Skeletons

Skeletons are a major new development. There are two main thrusts of the development.

1. Allowing records to be ‘parameterized’, so that parameters to the record may be specified.
2. Extending records to include additional features like functions, procedures and properties.

In many ways these extensions are isolated from the original compiler to minimise the risk of breaking existing code. I did try extending existing records to support functions and procedures but that caused too many issues with the existing functionality.

1. Parameterization of Records

This allows ‘late defined’ constants and vars to be used when defining a record.

There are two syntaxes for using a skeleton, the definition and usage. The syntax for these two is slightly different.

The syntax for defining the constants is < [var]*name : type [= default value]* [;…] >

The angle brackets <...> identify the record as a skelton record.

Example

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen - 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
end;
```

or

```
TFIFO : record< BuffLen : word = 8>
  Data : array[ 0..BuffLen - 1]  of byte;
```

```
  ReadPtr : word;
  WritePtr : word;
end;
```

The advantage of describing it this way (rather than defining the const BuffLen externally) is that if, for example, two FIFO buffers are needed with buffers of different lengths, a single definition is required, and these can be later declared like this

```
var
  USBBuff : TFIFO<BuffLen = 64>;
  SerBuff : TFIFO<BuffLen = 8>;
```

Internally two separate record structures will be created and used, just as they would need to be the traditional way, but the benefit to the programmer is that the compiler does this work for him.

Important

A skeleton record must be completely defined, including all functions and procedures before creating any instances. The easiest way to ensure that is to get into the habit of putting all skeleton definitions into a separate unit.

Another, more complex example would be

```
TBiDiFIFO= Record< SendLen :word; RcvLen : word>
  SendBuff : TFIFO< BuffLen = SendLen >;
  RcvBuff : TFIFO< BuffLen = RcvLen >;
…
end;
```

The declaration of a var of this type might be

```
var
  SerBuff : TbiDiFIFO<SendLen = 8; RcvLen=10>;
```

The syntax for using the constants is

< ParmName = ParmValue[;...] >

Examples are shown above.

Defaults in skeletons

Defaults are permitted in skeletons too. So, for example if we decide most buffers are 8 bytes long the above example becomes

```
TFIFO : record< BuffLen : word = 8 >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
end;

var
  USBBuff : TFIFO<BuffLen = 64>;
```

```
  SerBuff : TFIFO<>;
```

‘var’ parameters

Normally parameters are constants (albeit delayed constants), but it is also possible to define a parameter as a var. There are many uses for var parameters. For example if you want to build a driver that needs to access the particular registers in the hardware, e.g. a particular (but as yet unknown) serial port registers. Note that var parameters (just like const parameters) do not take up any room in the record, but are accessible from all record functions and procedures.

Visibility of fields in records

By default all fields are **public**, that is to say, visible and modifiable by all. For skeleton records only it is now possible to hide fields so that those fields can only be modified by the record itself (through record functions, record procedures or record properties – see later) using the keyword **private**. The keyword **public** is also supported to complete the visibility possibilities.

Once a visibility mode is specified it applied to all subsequent fields until another visibility mode is specified.

Record functions and procedures

These are only supported for skeleton records.

These methods have a built in hidden parameter called ‘self’ which is of the record type.

The simple definitions of functions and procedures is how you might expect. For example

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
  function Empty: boolean;
end;

function TFIFO.Empty : boolean;  // Note that you do not repeat
the <> parameter list
begin
  return (ReadPtr = WritePtr);
end;
```

An example that uses the parameter (delayed constant) BuffLen is

```
TFIFO : record< BuffLen : word >
  Data : array[ 0..BuffLen – 1]  of byte;
  ReadPtr : word;
  WritePtr : word;
  function Full: boolean;
end;

function TFIFO.Full: boolean;
var
  iTest : word;
```

```
begin
  iTest := WritePtr + 1;
  if iTest = BuffLen then
    iTest := 0;
  endif;
  return( iTest = iReadPtr );
end;
```

Default values in record functions and procedures

Record functions and procedures may have defaults, but those defaults must only be defined inside the record definition, so for example:

```
type
  TMyRec : Record<>
    function XX( v : byte = 7) : byte;
  end;

the implementation is written as

function TmyRec.XX( v : byte ) :byte;
begin
  …
end;
```

Note that the default value is not repeated in the implementation. This makes it easier to change the default value later, because it only need to be changed in one place.

Record Properties

A little like normal properties a record property has a getter and a setter (**read** and/or **write**). However, unlike normal properties the setter and getter must be defined within the record, either as a field or as a function or procedure with the correct form.

For example:

```
TFIFO : record< BuffLen : word >
  private
    fData : array[ 0..BuffLen – 1]  of byte;
    fReadPtr : word;
    fWritePtr : word;
    function GetNext: byte;
    procedure SetNext( Value : byte );
  public
    property Next : byte read GetNext write SetNext;
    property ReadPtr : word read fReadPtr; // don’t allow
uncontrolled write to this field
end;
```

Notice how one property definition accesses a function and a procedure, while the other access a field directly. Notice also how everything except the properties are hidden from the programmer so uncontrolled access is forbidden.

The definitions of the functions might be as follows:

```
function TFIFO.GetNext: byte;
var
   Result : byte;
begin
   if fReadPtr <> fWritePtr then
      Result := fData[ fReadPtr ];
      fReadPtr := fReadPtr + 1;
      if fReadPtr = BuffLen then
   else
     return( 0 );
   endif;
end;

procedure TFIFO.SetNext( Value : byte );
var
  iNext : word;
begin
   iNext := fWritePtr + 1;
   if iNext <> fReadPtr then // buffer not full
     Data[ fWritePtr ] := Value;
     fWritePtr := iNext;
  endif;
end;
```

Array properties are also permitted, but in that case both setter and getter must be methods (procedure and function respectively).

Other Notes:

1. **Private** is more or less equivalent to Delphi **strict protected**.

2. Chevrons are only required when creating record definitions and creating a record. They are not required for any function or procedure coding.

3. Skeleton records must be completely defined, including all functions and procedures *before* creating any instances of the records.

Constants and types in record functions and procedures

Because there may be multiple physical copies of a record function or procedure, you are not permitted local constants within a function. Use record skeleton constants (late defined constants) or global constants instead. Local types are also currently not implemented. Local vars are of course permitted.

Making Record Functions and procedures efficient.

Because multiple copied of a record function or procedure exists it makes sense to keep the definitions as short as possible, perhaps by calling an external non-record function to do most of the work. This is good practice anyway.

Restrictions on Record functions and procedures

Certain types of constructs are prohibited in record functions and procedures:

goto (and also labels)
ASM

but of course record functions can call normal functions and procedures that do contain these constructs.

What could possibly go wrong?

Despite their power there are issues with skeletons. Error messages could be in places that are not intuitively obvious or indeed very helpful. For example suppose we have a TFIFO function

```
function TFIFO.GetPos9 : byte;
begin
  return( fData[9]);
end;

Then later we define

var
  iBuff1 : TFIFO<Bufflen = 10>;
  iBuff2: TFIFO<BuffLen=8>;
```

Then fData[9] is valid for iBuff1, but not for iBuff2, so the error can only appear on the line defining iBuff2. Narrowing it down from there might not be easy.