Language Extensions: Variants

by Xavier Pacheco

Delphi 2.0 has a new and powerful data type called the *variant*. Variants were brought about in support of OLE Automation which makes use of them heavily. In fact, Delphi's variant data type is an encapsulation of the variant used with OLE. Delphi 2.0's implementation of variants has also proven to be useful in other areas of Delphi programming, as I will illustrate. Delphi 2.0 is the only compiled language that properly integrates variants as a *dynamic* data type at run time and as a *static* type at compile time, in that the compiler always knows that it is a variant.

Variants Change Types Dynamically

One of the main purposes of variants is to have a variable whose underlying data type cannot be determined at run time. This means that a variant can change the type to which it refers at run time. For example, the code shown in Listing 1 will compile and run properly.

Variants can support all simple data types such as integers, floating point values, strings, booleans, date and time, currency and also OLE Automation objects. Note that variants cannot refer to Object Pascal objects. Also, variants can refer to a non-homogeneous array which can vary in size and whose data elements can refer to any of the above data types, including another variant array.

The Variant Structure

The data structure defining the variant type is defined in the SYSTEMS.PAS unit and is also shown in Listing 2.

The `TVarData` structure consumes 16 bytes of memory. The first two bytes contain a word value that represents the data type to which the variant refers. Listing 3 shows the various values that may appear in the `VType` field. The next 6 bytes are unused. The remaining 8 bytes contain the actual data represented by the variant. Again, this structure maps directly to OLE's implementation of the variant type.

You'll notice from Listing 2 that the `TVarData` record is actually a variant record. Don't confuse this with the variant type. Although the variant record and variant type have similar names, they represent two totally different constructs. The `case` statement in the `TVarData` variant record indicates the type of data to which the variant refers. For example, if the `VType` field contains the value `VarInteger`, then only 4 bytes of the 8 data bytes in the variant are used to hold an integer value. Likewise, if `VType` has the value `varByte`, only 1 byte of the 8 is used to hold a byte value.

You'll notice that if `VType` contains the value `varString`, the 8 data bytes don't actually hold the string, but rather, they hold a pointer to this string. This is an important point because you can access fields of a variant directly:

```
var V: Variant;
begin
  TVarData(V).VType :=
    VInteger;
  TVarData(V).VInteger := 2;
end;
```

You must understand that in some cases this is a dangerous practice

➤ *Listing 1*

```
var V: Variant;
begin
  V := 'Delphi 2.0 is Great!'; // Variant holds a string
  V := 1;                      // Variant now holds an integer
  V := 123.34;                 // Variant now holds a floating point
  V := True;                   // Variant holds a boolean
end;
```

➤ *Listing 2*

```
TVarData = record
  VType: Word;
  Reserved1, Reserved2, Reserved3: Word;
  case Integer of
    varSmallint: (VSmallint: Smallint);
    varInteger:  (VInteger: Integer);
    varSingle:   (VSingle: Single);
    varDouble:   (VDouble: Double);
    varCurrency: (VCurrency: Currency);
    varDate:     (VDate: Double);
    varOleStr:   (VOleStr: PWideChar);
    varDispatch: (VDispatch: Pointer);
    varError:    (VError: Integer);
    varBoolean:  (VBoolean: WordBool);
    varUnknown:  (VUnknown: Pointer);
    varByte:     (VByte: Byte);
    varString:   (VString: Pointer);
    varArray:    (VArray: PVarArray);
    varByRef:    (VPointer: Pointer);
end;
```

➤ *Listing 3*

```
{ Variant type codes }
varEmpty    = $0000;
varNull     = $0001;
varSmallint = $0002;
varInteger  = $0003;
varSingle   = $0004;
varDouble   = $0005;
varCurrency = $0006;
varDate     = $0007;
varOleStr   = $0008;
varDispatch = $0009;
varError    = $000A;
varBoolean  = $000B;
varVariant  = $000C;
varUnknown  = $000D;
varByte     = $0011;
varString   = $0100;
varTypeMask = $0FFF;
varArray    = $2000;
varByRef    = $4000;
```

because it is possible to lose the reference to a string or other garbage collected entity, which will result in your application leaking memory. You'll see what I mean by the term garbage collected in the following section.

Variants Are Garbage Collected

By this I mean that Delphi automatically handles the allocation and de-allocation of memory required for a variant type. For example, examine the code which assigns a string to a variant variable below:

```
procedure ShowVariant;
var V: Variant
begin
  V := 'This is a long string';
  ShowMessage(V);
end;
```

What actually happens here is that Delphi first allocates the memory for the string and then refers the variant to that string by pointing to it with the data bytes of the variant. When the variant leaves scope, that is, the procedure ends and returns to the code that called it, the string associated with the variant is automatically freed. Delphi does this by implicitly inserting a `try..finally` block in the procedure as shown in Listing 4.

This same implicit release of memory occurs when you assign a different data type to the variant, for example, examine the two code snippets in Listing 5.

Here again, Delphi must allocate the memory for the string before making the assignment. Then, before assigning another value or data type to the variant, the memory being used up by the string must be released.

If you understand what happens in these illustrations, you will see why it is not recommended that you manipulate fields of the `TVarData` record directly as shown below:

```
procedure ChageVariant;
var
  V: Variant
begin
  V := 'This is a long string';
  TVarData(V).VType :=
    varInteger;
  TVarData(V).VInteger := 32;
  V := 34;
end;
```

Although this appears to be safe, in fact it is not because it results in the memory used by the string being lost. As a general rule, don't access the `TVarData` fields directly, or at least be absolutely sure that you know exactly what you're doing.

```
procedure ShowVariant;
var V: Variant
begin
  { First allocate memory for the string and make the assignment }
  try
    V := 'This is a long string';
    ShowMessage(V);
  finally
    { Now free the memory associated with the variant }
  end;
end;
```

➤ *Listing 4*

```
procedure ChangeVariant;
var V: Variant
begin
  V := 'This is a long string';
  V := 34;
end;
{This code causes Delphi to implicitly insert a try..finally statement
 as shown below: }
procedure ChangeVariant;
var V: Variant
begin
  { First allocate memory for the string and make the assignment }
  try
    V := 'This is a long string';
  finally
    { Now free the memory associated with the variant before making
      the assignment below }
  end;
  V := 34;
end;
```

➤ *Listing 5*

```
S := String(V);   // S will contain the string '1.6'
I := Integer(V);  // I is rounded to the nearest integer value, in this case - 2
B := Boolean(V);  // B contains false if V contains 0, otherwise B is true
D := Double(V);   // D contains the value 1.6
```

➤ *Listing 6*

```
var V1, V2, V3: Variant;
begin
  V1 := '100';  // A String type
  V2 := '50';   // A String type
  V3 := 200;    // An Integer type
  V1 := V1 + V2 + V3;
end;
```

➤ *Listing 7*

Typecasting Variants

You can explicitly typecast expressions to type variant. For example the expression `Variant(X)` results in a variant type whose type code corresponds to the result of the expression `X` which must be an integer, real, currency, string, character or boolean type.

You can also typecast a variant to that of a simple data type. For example, given the assignment `V := 1.6;` where `V` is a variable of type variant, the expressions in

Listing 6 will have the results shown. These results are dictated by certain type conversion rules applicable to variants. These rules are defined in detail in Delphi 2.0's *Object Pascal Language Guide*.

By the way, in the above example, it is not necessary to typecast the variant to another data type to make the assignment. This code would work just as well:

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

What happens here is that the conversions to the target data types are made through an implicit typecast. However, because these conversions are made at run time, there is much more code logic attached to this method. If you are sure of the type which a variant contains, then you are better off explicitly typecasting it to that type in order to speed up the operation. This is especially true if the variant is being used in an expression, which I'll discuss next.

Variants In Expressions

You can use variants in expressions with the following operators: `+, =, *, /, div, mod, shl, shr, and, or, xor, not, =, <>, <, >, <=, >=`. When using variants in expressions, Delphi knows how to perform the operations based on the contents of the variant. For example, if two variants `V1` and `V2` contain integers, the expression `V1 + V2;` results in the addition of the two integers. However if `V1` and `V2` contain strings, then the result is a concatenation of the two strings. Now, what happens if `V1` and `V2` contain two different data types? Delphi uses certain promotion rules in order to perform the operation. For example, if `V1` contains the string `'4.5'` and `V2` contains a floating point number, `V1` will be converted to a floating point and then added to `V2`. The code in Listing 7 illustrates this.

Based on what I just mentioned about promotion rules, it would seem at that first glance that the code in Listing 7 would result in `V1` having the value 350 as an integer. However, if you take a closer look you'll see that this is not the case. Because the order of precedence is from left to right, the first equation that gets executed is `V1 + V2`. Since these two variants refer to strings, a string concatenation is performed resulting in the string '10050'. That result is then added to the integer value held by the variant `V3`. Since `V3` is an integer, the result `'10050'` is converted to an integer and added to `V3`'s value giving an end result of 10250.

Delphi 2.0 promotes the variants to the highest type in the equation in order to successfully carry out the calculation. However, when an operation is attempted on two variants of which Delphi 2.0 cannot make any sense, an *Invalid variant type conversion* exception is raised. The code below illustrates this:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'hello';
  {next line raises exception}
  V1 := V1 / V2;
end;
```

As stated earlier, it is sometimes a good idea to explicitly typecast a variant to a specific data type if you know what that type is and if it is used in an expression. Consider the following line of code:

```
V4 := V1 * V2 / V3;
```

Before a result can be generated for this equation, each operation goes through a run time function that goes through several gyrations to determine the compatibility of the types which the variants represent. Then the conversions are made to the appropriate data types. This results in a large amount of overhead and increased compiled code size. A better solution is obviously not to use variants. However, when necessary, you can also explicitly typecast the variants so that the data types are resolved at compile time:

```
V4 := Integer(V1) *
  Double(V2) / Integer(V3);
```

Keep in mind that this assumes you know the data types which the variants represent.

Null And UnAssigned

There are two special types of variants that I need to briefly discuss. The first is the `varEmpty` which means that the variant has not yet been assigned a value. This is the initial value of the variant as it comes into scope. `varNull` is different to `varEmpty` in that it actually represents a value. This is primarily true of field values of a database table. It is possible for a field to contain a `NULL` value. You'll see later in this article how the `TTable` component uses variants to reference its fields.

Another difference is that attempting to perform an equation with a variant containing a `varEmpty` value will result in an *Invalid variant operation* exception. The same is not true of variants containing a `varNull` value. However, when a variant involved in an equation contains a `NULL` value, that value will propagate to the result. Therefore, the result is always `NULL`.

Efficiency Concerns

It may be tempting to use variants instead of the conventional data types since they seem to offer so much flexibility. However, this will increase the size of your code and make it run slower. Additionally, it will make your code more difficult to maintain. Variants are useful in many situations. In fact, the developers of Delphi use variants with the database components because of the flexibility they offer. Generally speaking, however, you should use the conventional data types instead of variants. Only in situations where the flexibility of the variant outweighs the performance advantage of the conventional method should you resort to using variants.

Variant Arrays

Earlier I stated that a variant can refer to a non-homogeneous array.

Therefore this syntax is valid:

```
var V: Variant;
    I: Integer;
begin
  I := V[1];
end;
```

Now, although the code above will compile, you'll get an exception because the variant array has not yet been created. There are several ways to create a variant array. One way is that the array is obtained from an OLE server function:

```
V := SomeOleServer.GetSomeArray;
```

Additionally, there are a couple of variant array support functions that allow you to create a variant array. These are `VarArrayCreate` and `VarArrayOf`. The function `VarArrayCreate` is defined as:

```
function VarArrayCreate(
  const Bounds:
  array of Integer;
  VarType: Integer): Variant;
```

To use it, you pass in the array bounds for the array you want to create and a variant type code for the type of the array elements. For example, the following returns a variant array of integers and assigns values to the array items:

```
var V: Variant;
begin
  {Create a 4-dimension array}
  V := VarArrayCreate([1, 4],
    varInteger);
  V[1] := 1;
  V[2] := 2;
  V[3] := 3;
  V[4] := 4;
end;
```

Assigning `varVariant` as the type code allows you to create a variant array of variants so that the elements of the array can differ. Also, you can create a multi-dimensional array by passing in the additional bounds required. For example, the code below creates an array with the bounds `[1..4, 1..5]`:

```
V := VarArrayCreate([1, 4, 1,
  5], varInteger);
```

```
procedure VarArrayRedim(var A: Variant; HighBound: Integer);
function VarArrayDimCount(const A: Variant): Integer;
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
function VarArrayLock(const A: Variant): Pointer;
procedure VarArrayUnlock(const A: Variant);
function VarArrayRef(const A: Variant): Variant;
function VarIsArray(const A: Variant): Boolean;
```

➤ *Listing 8*

```
procedure VarClear(var V: Variant);
procedure VarCopy(var Dest: Variant; const Source: Variant);
procedure VarCast(
  var Dest: Variant; const Source: Variant; VarType: Integer);
function VarType(const V: Variant): Integer;
function VarAsType(const V: Variant; VarType: Integer): Variant;
function VarIsEmpty(const V: Variant): Boolean;
function VarIsNull(const V: Variant): Boolean;
function VarToStr(const V: Variant): string;
function VarFromDateTime(DateTime: TDateTime): Variant;
function VarToDateTime(const V: Variant): TDateTime;
```

➤ *Listing 9*

The `VarArrayOf` function is defined:

```
function VarArrayOf(
  const Values:
  array of Variant): Variant;
```

This function returns a one dimensional array whose elements are given in the `Values` parameter. The example below creates a variant array of 3 with integer, string and floating point values:

```
V := VarArrayOf([1,'Delphi',2.2]);
```

Variant Array Support Functions

There are several other variant array support functions. These functions are defined in the `System` unit and are also shown in Listing 8.

The `VarArrayReDim` function allows you to resize the upper bound of the rightmost dimension of a variant array. The function `VarArrayDimCount` returns the number of dimensions in a variant array. `VarArrayLowBound` and `VarArrayHighBound` return the lower and upper bounds of an array respectively. `VarArrayLock` and `VarArrayUnlock` are two special functions which I'll go into more detail in a moment.

`VarArrayRef` is a special function that was added in a later release of Delphi 2.0. this function resolved a problem that existed in passing variant arrays to OLE Servers. The problem was that when you had a statement such as:

```
Server.PassVariantArray(VA);
```

the array is passed not as a variant array, but rather as a variant by reference. If the server expected a variant array a type mismatch occurs. There was no way to pass a variant array. `VarArrayRef` takes care of this situation. It takes a variant containing a variant array and returns the expected type. Its syntax is:

```
Server.PassVariantArray(
  VarArrayRef(VA));
```

`VarIsArray` returns true if the variant parameter passed to it is a variant array, otherwise false.

Initializing A Large Array

Variant arrays are important in OLE Automation. You can use variant arrays to pass binary data to an OLE Automation object. Examine the line below:

```
V := VarArrayCreate(
  [1, 10000], VarByte);
```

This line creates a variant array of 10,000 bytes. Suppose you have another array (non-variant) declared of the same size and you want to copy the contents of this non-variant array to the variant

array. Normally, you can only do this by looping through the elements and assigning them to the elements of the variant array as shown below:

```
begin
  V := VarArrayCreate([1,
10000], VarByte);
  for i := 1 to 10000 do
    V[i] := A[i];
end;
```

The problem with this code is that it creates too much overhead just to initialize the variant array elements. This is due to the assignments to the array elements having to go through the run time logic to determine compatibility and so forth. To avoid this translation of data, you should use the VarArrayLock and VarArrayUnlock functions.

VarArrayLock returns a pointer to the array data and locks the array so that it cannot be resized while it is locked. VarArrayUnLock unlocks an array locked with VarArrayLock. Once the array is locked, you can use a more efficient means to initialize the data by using, for example, the Move procedure with the pointer to the array's data. The code below performs the initialization of the variant array show above, but in a much more efficient manner:

```
begin
  V := VarArrayCreate(
    [1, 10000], VarByte);
  P := VarArrayLock(V);
  try
    Move(A, P^, 10000);
  finally
    VarArrayUnlock(V);
  end;
end;
```

Database Use Of Variants

One of the areas where Delphi uses variant types is with the FieldValues property of a TDataSet. This property is declared as:

```
property FieldValues[
  const FieldName: string]:
  Variant; default;
```

FieldValues returns the value of a field specified by FieldName as a variant. So, you can obtain the value of a field "Last Name" with the following line:

```
LN := Table1.FieldValues["LastName"];
```

Also, since FieldValues is the default property for the TDataSet, you can use the following syntax:

```
LN := Table1['LastName'];
```

Since this is the likely way that you'll be accessing the field values of tables, this shows why it is a good idea for you to understand how to work with variants. One thing to keep in mind is that often a particular field will contain a NULL value. Attempting to assign a NULL to a variable will result in an exception. So, if in the above line the LastName field contains NULL you'll get an exception. Therefore, if you're not sure whether an field may be NULL you should use the VarToStr function:

```
LN := VarToStr(Table1['LastName']);
```

What this does is check to see if the variant passed to it is NULL. If so, it returns an empty string, otherwise, it returns the string value.

Another area of importance is the Locate function of a TDataSet, which is declared as:

```
function Locate(const
  KeyFields: string;
   const KeyValues: Variant;
   Options: TLocateOptions):
   Boolean;
```

KeyFields lists one or more fields separated by a semicolon on which a search is to be performed. KeyValues is a variant or variant array specifying the field value or values to match in the search. A typical way to perform a search on multiple fields is to use the VarArrayOf function to construct the variant array to pass to Locate. An example is shown below:

```
if not Table1.Locate(
  'LastName;FirstName',
  VarArrayOf(['Fisher',
  'Pete']),[]) then
  ShowMessage('Not Found');
```

Variant Use In OLE Automation

Since this is not an article on OLE Automation I won't go into depth on this topic. Basically, variants can contain a reference to an OLE Automation object. This means that you can call methods and get/set properties of the object through the variant. Therefore, the following lines of code would be valid provided that the variant refers to an OLE object:

```
V.SomeFunc;
V.SomeProp := 2;
```

As variants referring to strings are garbage collected, so are variants that refer to OLE Automation objects. For example, take a look at the code below:

```
V := CreateOleObject(
  'Word.Basic');
V.FileNew('Normal');
V := 22;
```

The first line creates a variant reference to an OLE Automation server, specifically Microsoft Word. The second line adds a new file to the MS Word environment. The third line assigns an integer value to the variant. Before this integer assignment is made, the reference to the OLE Automation object, any memory and resources are freed implicitly. This is part of the default garbage collection handled by Delphi. OLE is really a separate topic, or many separate topics, in itself. Therefore, I just wanted to briefly mention how variants are used with this technology.

Supporting Functions

There are several other support functions for variants that you can use. These function are declared in the System unit and are also shown in Listing 9.

The VarClear procedure clears a variant and sets the TVarData.VType field to varEmpty. VarCopy copies the Source variant to the Dest variant. The VarCast procedure converts a variant to a specified type and stores that result into another variant. VarType returns one of the type

codes listed in Listing 3 on a specified variant. `VarAsType` has the same functionality as `VarCast`. `VarIsEmpty` returns true if the type code on a specified variant is `varEmpty`. `VarIsNull` indicates whether or not a variant contains a `NULL` value. `VarToStr` was described previously. `VarFromDateTime` returns a variant which contains a given `TDateTime` value. `VarToDateTime` returns the `TDateTime` value contained in a variant.

Conclusion

Variants are a powerful data type that can add flexibility to your applications. Also, with the increase in OLE's popularity, it is beneficial for you to know how to perform operations on variant data types as well as issues regarding variant usage. Finally, my thanks to Steve Teixeira for his help in reviewing this article.

Xavier Pacheco is a Field Consulting Engineer with Borland International and co-author of *Delphi 2.0 Developer's Guide*. You can reach him by email at xpacheco@wpo.borland.com or on CompuServe at 76711,666