# Typecasting Explained: Part 1

*by Brian Long*

**Typecasting allows us to access information that is stored as one data type as if it were another type. For example, if we have a word variable (4 bytes), a typecast would allow us to treat it as if it were two bytes, or one byte, or four bytes, or a pair of characters... Similarly, if an expression evaluates to a long integer, a typecast could be employed to view the data as a pointer, or a two-word record...**

Let's take one of these examples and see how we can implement typecasting and what the alternatives are. Listing 1 is the unit for a simple program with two buttons, whose event handlers both do essentially the same thing: they set the colour of the form to a random colour, obtained by using the Windows API routine `RGB`. The value of the `Color` property (which is of type `TColor`, a VCL-defined range of long integer values) is then assigned to a local `Longint` variable curiously called `Tag`, whose whole value, and also its low and high word values, are to be written to the screen (see Figure 1).

In order to ensure the `Random` routine does generate a truly pseudo-random sequence of numbers, the on-line help tells us to call `Randomize` before we call `Random`. `Randomize` initialises the seed used by the random number generator with a value obtained from the DOS Get Time interrupt (interrupt $21, function $2C). We could call `Randomize` as one of the first actions in the project source file before `Application.Run`, or in the form's `OnCreate` handler, but here I have elected to put it in the form unit initialisation section.

The initialisation code for all units used in an application is executed before the very first line of the project source, so right at the very start of the program. You'll note that this is marked by the word `initialization` just before the final `end.` of the unit. This is a new keyword in Delphi, but you can also use the historic alternative of a `begin` in its place. The online help heartily recommends using `initialization` instead, so it appears a little odd that Borland use the old `begin` approach in all the source they supply in Delphi Client/Server.

The methods used to extract the two words from the `Longint` are the points of interest here. The extractions performed by the first button, labelled *Old Way*, don't use typecasting at all. They do it in the fashion I was taught at college, using good old-fashioned, down to earth, no messing about, plain bitwise manipulations. To get the low word from a long integer, ie the bottom 16 bits from a 32-bit value, you perform a binary `AND` operation between the long integer and $2^{16}$-1, or $FFFF – this strips off the top word, leaving just the bottom word. To get the high word, you shift the value right by 16 bits, to leave you with the desired value.

The second button obtains exactly the same results, but avoiding any bit operations. However, it does tackle the problem from two different angles. It uses two different typecasts, one a value typecast, used to get the low word, and one a variable typecast, used to get the high word.

The general format of a typecast is to specify the new type that you wish to get, followed by a pair of parentheses surrounding the original variable or expression. Let's look at the two typecasts in turn.

➤ *Listing 1 (file COLOURSU.PAS on the disk)*

```
unit Coloursu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    LowLbl: TLabel;
    HighLbl: TLabel;
    WholeLbl: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private    { Private declarations }
  public     { Public declarations }
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
var  LoWord, HiWord, Tag: Longint;
begin
  Color := RGB(Random(256), Random(256), Random(256));
  Tag := Color;
  LoWord := Tag and $FFFF;
  HiWord := Tag shr 16;
  HighLbl.Caption := '$' + IntToHex(HiWord, 4);
  LowLbl.Caption := '$' + IntToHex(LoWord, 4);
  WholeLbl.Caption := '$' + IntToHex(Tag, 8);
end;
procedure TForm1.Button2Click(Sender: TObject);
var  LoWord, HiWord, Tag: Longint;
begin
  Color := RGB(Random(256), Random(256), Random(256));
  Tag := Color;
  LoWord := Word(Tag);
  HiWord := LongRec(Tag).Hi;
  HighLbl.Caption := '$' + IntToHex(HiWord, 4);
  LowLbl.Caption := '$' + IntToHex(LoWord, 4);
  WholeLbl.Caption := '$' + IntToHex(Tag, 8);
end;
initialization
  Randomize;
end.
```

## Value Typecasts

```
LoWord := Word(Tag);
```

The size of Tag, the original variable used here, is four bytes. The size of the target type, Word, is two bytes. Because of the different sizes, this is interpreted as a value typecast. This means that before the compiler worries about what you are hoping to get out of the typecast, it evaluates the expression in the brackets to produce a value. In this case it is simply the value of a variable, but you could easily substitute an arithmetic calculation. Once a value is obtained, the compiler does what it can to treat it as the desired type. If the target type is smaller, as it is here, it merely truncates the value, knocking off any high bytes that aren't needed, so in this case we just get the low word from Tag. If the target type is bigger, it pads it out with zeros. This form of value typecast is implicitly performed by the compiler when doing certain operations. The following routine has four assignments. The first two do exactly the same thing. The second two are also equivalent:

```
procedure DoAssign(WordVal:
  Word; LongVal: Longint);
var  W: Word;
     L: Longint;
begin
  W := Word(LongVal);
  W := Val2;
  L := Longint(WordVal);
  L := Val1;
end;
```

## Variable Typecasts

```
HiWord := LongRec(Tag).Hi;
```

What's going on here is an interpretation of Tag as if it were a record of type LongRec as defined in the SysUtils unit. LongRec is a two word record specifically designed for use in typecasting long integers. Because we are attempting to access a structure member of the result of a typecast, in this case a record field, the compiler sees this as a variable typecast. It thereby introduces a ruling that the typecast subject (Tag) must be a variable, not an expression, and that the target type (LongRec) is the same size as the original type (Longint). A LongRec is four bytes, the same size as a Longint and so the typecast succeeds and the high word gets read.

The significance of the requirements that the typecast be against a variable and the source and target be the same size start to become clear when you realise that a variable typecast can be used on the left hand side of an assignment. So, for example, using the typecast shown above, an equally valid statement would be:

```
LongRec(Tag).Hi := 35;
```

which would write 35 into Tag's high word, leaving the low word untouched. Unlike a value typecast, the compiler does not evaluate Tag first, it generates code that assumes Tag is indeed a LongRec.

The Tag variable wasn't so named by chance, as you perhaps guessed: it was named the same as a property of the form (and indeed every other component type). The form's Tag property is a Longint, as is our current Tag variable. We can test the compiler's typecasting rules by temporarily deleting the two Tag variables from the unit in Listing 1, and trying to recompile.

You will notice that both the old-fashioned extractions compile fine with the Tag property, as does the value typecast in the second button's OnClick event handler. On a less successful note, the variable typecast does not compile. A property is not a variable, despite the fact that it can be treated like one, being assigned to and from. A property is an expression. Therefore, only value typecasts are applicable to property values.

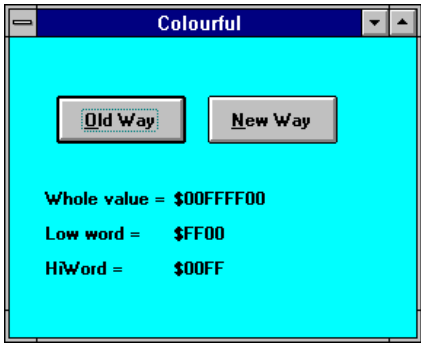
## Common Uses Of Typecasting

You may well have come across this issue before, if you've dabbled with a third generation language. Take the simple application in Listing 2 (see Figure 2). Essentially one form with three SpinEdit controls from the standard page of the component palette placed on it. One of the spin edits (Res) is read only, the other two (Op1 and Op2) have one shared event handler attached to both their OnChange events. The scheme is that as you type numbers into the latter two, the read-only one displays the result of multiplying the two numbers together. To bring the particular problem I wish to cover to light, I have used some variables in the implementation of the OnChange event handler, OpChange.

There are two word variables used to store the numbers from Op1 and Op2 (these spin edits' values are limited to 65535). The result of the multiplication is stored in a Longint variable and that variable is then displayed in the Res spin edit. When you run the program, if you type 1000 into the first spin edit and then 50 into the second, the correct result of 50000 is shown in Res. Increasing the number in Op2 keeps producing correct multiplications as you go past 60, but when you get to 66 it all falls apart, despite having a long integer to store the result in – the displayed result is 464.

However, the problem is not with the left hand side of the assignment but the right hand side. When Delphi's expression parser

➤ *Figure 1 Colour by numbers*




➤ *Figure 2 Not even running on a Pentium...*

evaluates the right hand side it scans it to see what the type of the largest individual part of it is. If there is a bracketed part of the expression, it recurses and evaluates that first – a pair of brackets mark one expression which may then be part of another expression. When it has identified the largest type present, it stores the intermediate result of the expression in storage of that type's size. In our case, the compiler places the result of the multiplication in a `Word` which is then assigned to the `Longint` variable.

Obviously, when the calculation produces a value greater than the largest `Word` value (65535), the value assigned ends up being just the lowest word of the result. Were we to have Arithmetic Overflow Checking on (either by the `{$Q+}` compiler directive, or `Options | Project | Compiler | Overflow checking`) we would be warned about this at run-time when the wraparound problem occurs.

To resolve this problem, we can use a `Longint` value typecast on one of the parts of the expression. It would be fruitless to apply it to the whole expression, as a value typecast is only applied when the expression has been evaluated, and by then the truncation has occurred. The following would be a valid way around the problem:

```
Num3 := Longint(Num1) * Num2;
```

Incidentally, this parser operation is not a bug; the same problem occurs in C and C++ compilers (at least, those that I have used).

Let's look at another example now, one using some elementary graphics by way of the `TShape` component. Listing 3 shows such an example. The intention is to draw a random shape on the form whenever a key is pressed. The width and height of the shape are random – that's easy enough. The colour is also random, we did that earlier. But how do we get a random shape to be drawn?

Bear in mind that a `TShape`'s shape is determined by its `Shape` property. This property is defined in the on-line help to be of type

```
unit Mathsu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Spin;
type
  TForm1 = class(TForm)
    Op1: TSpinEdit;
    Op2: TSpinEdit;
    Res: TSpinEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure OpChange(Sender: TObject);
  private    { Private declarations }
  public     { Public declarations }
  end;
var  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.OpChange(Sender: TObject);
var  Num1, Num2: Word;
     Num3: Longint;
begin
  if (Op1.Text = '') or (Op2.Text = '') then Exit;
  Num1 := Op1.Value;
  Num2 := Op2.Value;
  Num3 := Num1 * Num2;
  Res.Value := Num3;
end;
end.
```

➤ *Listing 2 (file MATHSU.PAS on the disk)*

```
unit Shapesu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls;
type
  TForm1 = class(TForm)
    procedure FormKeyPress(Sender: TObject; var Key: Char);
  private    { Private declarations }
  public     { Public declarations }
  end;
var  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin  { The variable with no name! }
  with TShape.Create(Self) do begin
    Left := 0;
    Top := 0;
    Width := Random(Self.ClientWidth);
    Height := Random(Self.ClientHeight);
    Brush.Color := RGB(Random(256), Random(256), Random(256));
    Shape := TShapeType(Random(6));
    Parent := Self;
  end;
  Key := #0; { Null the key to prevent any further key-press processing }
end;
initialization
  Randomize;  { For Random to be truly random }
end.
```

➤ *Listing 3 (file SHAPESU.PAS on the disk)*

`TShapeType`, which is an enumerated type which can take one of six values: `stRectangle`, `stSquare`, `stRoundRect`, `stRoundSquare`, `stEllipse` or `stCircle`. How do we effectively pick a random value from an enumerated type?

We can achieve our goal by taking advantage of a useful implementation detail. When the compiler generates an executable, an enumerated type's values boil down to being represented by sequential numbers starting at zero. So `stRectangle` would be 0 and `stCircle` would be 5. Given an understanding of this point, we can generate a random number between 0 and 5 and apply a value typecast to it to turn it into a `TShapeType`, and the compiler lets it through, as is done in the listing.

One of the more common uses of typecasts is in finding the segment and offset values of a pointer. Where it used to be common to use the `Seg` and `Ofs` functions under DOS, they aren't really reliable under Windows and so typecasts tend to be used instead. Let's have a quick look at `Seg` and `Ofs` and see what the problem is. These functions return the segment and offset of the thing passed to them, so given a `Word` variable called `W`, we could say:

```
Segment := Seg(W);
Offset := Ofs(W);
```

This returns the segment and offset parts of the address of `W`, ie its storage location. But given a pointer `P` instead, we can't get its constituent segment and offset using:

```
Segment := Seg(P);
Offset := Ofs(P);
```

because that gives us the segment and offset of the address of `P`, ie the place where `P` is stored, and not `P` itself. To get `Seg` and `Ofs` to return the required data, we need:

```
Segment := Seg(P^);
Offset := Ofs(P^);
```

This gives us the segment and offset of the address of the thing pointed to by `P`. But as I mentioned, this is not reliable in Windows. The two instructions above cause a pointer load at machine level. In protected mode, the CPU checks each pointer load to ensure the pointer is valid, or at least that its segment is. If `P` is not a valid pointer, a General Protection Fault will be generated. I tried this with a randomly chosen pointer value of $9999:$999 and got one.

In protected mode, the preferred approach to finding the segment and offset of a pointer is to apply a variable typecast to the pointer using the `PtrRec` record defined in `SysUtils`:

```
Segment := PtrRec(P).Seg;
Offset := PtrRec(P).Ofs;
```

## To Be Continued...

In the next issue we'll continue by looking at typecasting objects and also delve a little into assembler!

---

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com



*A note from the Editor...*
For more information on allocating and using selectors, and also on a wide range of subjects including the Pascal expression parser, typecasting, direct memory access etc (some of which are to be covered in Part 2 of this article), see *The Borland Pascal Problem Solver* by Brian Long, published in 1994 by Addison-Wesley, ISBN 0-201-59383-1. Admittedly, it was written with Borland Pascal in mind, but many of the concepts and ideas transfer readily across. I can recommend it.