Moving Up: C And C++

by Dave Jewell

In this final Moving Up article, Dave looks at the likelihood of 'culture-shock' for C/C++ developers moving across to Delphi!

Delphi For C Programmers

For starters, if you're one of those programmers who spends ages deliberating over which memory model to use – forget it!

Borland's Pascal dialect only gives you one memory model. All pointers within Delphi are 'far' while global variables (see my comments later) are always 'near', being stored as part of the program's `DGROUP` segment. That said, Object Pascal has a nice memory sub-allocator scheme which means you don't often have to mess around with `GlobalAlloc`, `GlobalLock` and so on.

A Pascal program is split up into one or more units (again, see later) and all routines exported from a unit are 'far' so that they can be called from another segment. When programming using Object Pascal, you'll find that you very rarely have to worry about whether or not a routine is near or far, a case in point being the `ExitProc` mechanism which will only work with far routines.

One significant problem is the lack of huge pointers, making it difficult to manipulate objects larger than 64Kb in size. Obviously, this won't be an issue once the 'flat model' 32-bit version of Delphi arrives, but until then, you'll have to make use of the `SelectorInc` variable (see the on-line help) to manually bump a pointer by 64Kb at a time. Contrary to early versions of the Delphi documentation, it's perfectly possible to use the `_hread` and `_hwrite` Windows API routines from within Object Pascal, so there are no problems with reading and writing objects that are very large, such as high resolution bitmaps, .WAV files, etc.

Object Pascal has a host of nice language features to lure the dedicated C/C++ enthusiast. It supports sets, making it very easy to test a variable against a number of possible values in one go. Nested routines make it easy to split up a large routine into a number of smaller entities in a manageable way. 'Child' routines have access to the parent's formal parameters and local variables eliminating a lot of unnecessary parameter passing.

The Delphi development system also uses 'smart linking', something which C++ vendors repeatedly tell me is in their product, but which never quite works as expected. Smart linking means that un-referenced routines, variables and string constants are quietly discarded at link time, thus minimising the size of the resulting .EXE file.

You might think that Delphi executables are pretty hefty at around 150Kb minimum, but you need to remember that these overheads arise from the OOP oriented VCL library – they're nothing to do with the underlying compiler implementation. As an example, it's perfectly possible to write a 'straight' (non-OOP) version of the infamous "Hello World" program using Delphi. Said program weighs in at under 2Kb!

A nice feature of Object Pascal is the concept of units. C/C++ programmers will be familiar with the idea of dividing up a large program into a number of separate source files, each of which implements a certain category of routines. For example, you might choose to put a number of data compression routines into a file called COMPRESS.C, or you might want to place buffered I/O classes into BUFF.CPP. Either way, it's entirely your responsibility to decide which routines and variables are "exported" by each source file. If you don't want a routine or variable to be accessible from outside the file where it's defined, you have to make it `static`. For each routine that you do want to make available to the program as a whole, you need to add it to a header file which is then included by other interested source modules.

Yes, it works, but it's not terribly convenient for one simple reason: the idea of modular, black-box programming isn't built into the C language – it's up to you to create your own black boxes by messing around with header files and using the `static` keyword.

Pascal is far superior (in my opinion!) in this respect. An Object Pascal program is made up of one or more units. Each unit is a separately compiled black box which exports routines and variables to the host program or to other units. A unit can also export type definitions and constant values, something that a C/C++ programmer would use a header file for. Internally, a Pascal unit is rigidly divided into two distinct parts: the `interface` part and the `implementation`. Anything that's defined in the `interface` is exported for use by other units, whereas the `implementation` part is completely private.

There are a couple of big advantages here. Firstly, you don't have to keep mucking about with separate header files and secondly, you can see at a glance which parts of a source file are exported and which parts form the private implementation. The VCL library is heavily based around the concept of units, as should be your own applications – where a language supports modular programming

practices and information hiding, it's a good idea to exploit these capabilities to the full.

A common question asked by C/C++ programmers moving to Delphi is "How do I create a global variable?" Put simply, you don't! The Pascal language has no concept of global variables, and with good reason! In a language which supports the free and easy use of global variables (such as C or C++), globals tend to be used indiscriminately by inexperienced programmers. Global variables introduce potential 'side-effects' into every procedure call and make it very difficult to understand large and complex programs with which you are unfamiliar. Globals go against the whole spirit of modular, structured programming.

If you really need a global variable, the correct approach is to define it inside a Pascal unit, adding it to the interface part of the unit. Only those source modules which `Use` the unit containing the variable will be able to see it: the variable doesn't automatically become available to all parts of the program.

This approach encourages you to think about which parts of your program actually need to use that variable and which don't. You'll often find that, given a little thought, a variable doesn't need to be exported from a unit at all.

Incidentally, you might think that the variables defined in the SYSTEM unit (`hInstance`, `IOResult`, etc) are 'true' variables because they appear to be available anywhere in a Delphi application. This is because the SYSTEM unit is special: it's implicitly included in the `Uses` clause of every other unit and is automatically available to the main program module.

Another useful aspect of units is the ability to perform hidden initialisation and de-initialisation (for want of a better word) of your black box routines. For example, suppose you have a palette tweaking unit which needs to figure out various system parameters such as size of the palette, number of colours supported and so forth. With Pascal, you can arrange for your initialisation code to be called before the main program starts executing. Also, if you make use of Pascal's `ExitProc` facility, (see the Delphi on-line help for details of how this works), you can arrange for each unit to get a 'good-bye kiss' when your program is unloaded. This is very useful for deleting custom bitmaps, brushes, pens and the like.

Delphi For C++ Programmers

Although Delphi's OOP capabilities are very respectable, it's got to be admitted that as far as 'high-end OOP' is concerned, Object Pascal is no match for the latest C++ dialects. For example, there's no template capability, no operator overloading and multiple inheritance (the ability to derive from more than one parent class) doesn't exist. Speaking personally, I understand the benefits of multiple inheritance, but I've just never felt the need for it myself and I know that quite a lot of other developers (C++ developers included!) feel the same way.

To some extent this reflects the different mindset that programmers adopt when using different languages. If something is alien to the philosophy of a particular language, then it's unlikely that you'll miss its absence. A good example of this is pre-processor macros. Virtually all C/C++ developers rely on macro pre-processing in one way or another, for example the message cracker macros which are built into WINDOWSX.H and allow you to write code which is portable between the Win16 and Win32 API sets. With Pascal, the concept of macro pre-processing just doesn't exist; there isn't even a pre-processor phase in the compiler!

Although this might seem like a massive limitation from a C/C++ developer's perspective, the absence of macros is completely unnoticed by your average Pascal developer. In the same way, most C/C++ programmers don't miss set operations and nested procedures – they'd only miss them if they were familiar with Pascal. It's all a question of what you're used to.

Incidentally, if you really do miss macro pre-processing, you can always run your Pascal source code through a pre-processor (such as CPP.EXE, part of the Borland C++ compiler package) before compilation.

Another feature which C++ developers will miss is the lack of implicit constructors and destructors. In C++, you can instantiate an object simply by declaring it as a global variable, in which case its constructor will be called before `WinMain` is executed and its destructor called when the program terminates. Alternatively, you can declare an object as an automatic variable inside a routine, causing the constructor and destructor to be called 'behind the scenes' when the object goes in and out of scope. This is certainly a neat feature of C++, but again, you'll find that making implicit `Create` and `Destroy` calls becomes perfectly natural after a while.

One thing that C++ developers will be familiar with is the idea of exception handling. The Delphi version of Object Pascal fully supports exception handling and you can define your own exceptions to cater for custom error situations such as wrong password, too many cooks processing broth and so forth!

Portability Between Win16 and Win32 Platforms: Doing Things The Delphi Way

A moment ago, I mentioned the lack of macro processing, and therefore – by implication – the lack of portability aids such as message cracker macros. How, then, is portability to be achieved?

The golden rule with Delphi programming is to use the VCL library wherever and whenever possible. It's the VCL application framework which insulates your application code from the specific details of whichever platform you're targeting.

For example, suppose you wanted to have some code which was executed whenever a particular form was loaded. In conventional API terms, you'd use the `WM_INITDIALOG` message to perform

such processing. Under Delphi, you'd use the `OnCreate` event. Similarly, imagine that you wanted to arrange things so that, initially, a list box appeared with the first item in the list box selected. Again, experienced Windows developers would do this by sending a `LB_SETCURSEL` message to the list box control. Although it is possible to do things this way with Delphi (you can get the API window handle of any windowed control from its `Handle` property), it's much better to do things the Delphi way and simply set the `ItemIndex` property of the control to zero, like this:

```
MyListBox.ItemIndex := 0;
```

You'll find that using the VCL library is not only more portable than hitting the API, but it also results in source code that's much more readable and concise. This is a point that I can't emphasise strongly enough – if you insist on doing things the way you've always done 'em, then you'll end up making a rod for your own back: you'll have a lot of grief when it comes to moving your application across to the 32-bit version of Delphi!

How then, are you supposed to find out which method to call in a particular situation? The Delphi on-line help documentation is, frankly, nothing to write home about. The Object Pascal and VCL Reference Manuals are now available, in Acrobat format for free on CompuServe and in paper at a small cost direct from Borland.

I've found that the best way of getting round the shortcomings in the documentation is to purchase the source code to the VCL library (of course, if you've bought the Client/Server version of Delphi, then you'll already have this source) and use a fast text-file scanning program to search for strings in the library source.

Personally, I use TS.EXE (Text Search), a very fast search utility that's provided as part of the Norton Utilities. TS can take a wildcard file description on the command line, allowing you to do something like this:

```
TS *.PAS "LB_GETCOUNT"
```

You may have another favourite tool, of course. Using this kind of setup, it's very easy to quickly track down methods of interest.

For example, imagine that you want to find out how many items there are in a listbox – a pretty obvious thing to wish to do! If you use the on-line help to look at the various methods and properties of the `TListBox` class, you'll be disappointed: there's nothing obvious there. To count the number of items, you actually have to look at the `Count` sub-property of the `Items` property of the listbox, like this:

```
NumItems :=
  MyListBox.Items.Count;
```

This isn't especially obvious for the beginning Delphi developer. However, if you have some knowledge of the Windows API, you can use the Norton Text Search program (or similar) to rapidly scan the source code for any occurrence of `LB_GETCOUNT`, the Windows message which must be invoked at some point in order to get hold of the wanted information. You can then work your way back up the VCL hierarchy until you've got the needed method.

Of course, there will be times when the API is the only way to do some particular job. Fortunately, Object Pascal does support conditional compilation and you could therefore do something like this:

```
{$IFDEF WIN32}
– Win32 specifics go here –
{$ELSE}
– Win16 specifics go here –
{$ENDIF}
```

This is the approach used by the developers of the VCL library. In my opinion, it's best to localise this sort of thing inside a single unit which presents a consistent, platform-independent interface to the outside world. That way, you only need to modify one source file when adding Win32 capabilities to your application.

Porting C Code To Pascal

You're not going to have much joy in porting MFC code or OWL code to Delphi because of the ompletely different framework architectures. However, if you've made a rigid distinction between how your application *looks* and what your application *does*, then you should be able to move quite a lot of code across. To put this another way, suppose you're writing an engineering program: you'd typically use the application framework (be it, MFC, OWL or VCL) to implement the sexy user interface features of your application, but the real nuts-and-bolts number crunching code should be completely divorced from the user interface.

If you've abided by this sort of approach, then you should be able to port your low-level number crunching code to Delphi by the simple expedient of wrapping it up into a DLL and calling the DLL from your VCL front-end code *[Dr Bob's HeadConv Delphi Expert, included on the disk with this issue in shareware form, can help you here by creating a Delphi import unit from the C header file. Editor]*.

Alternatively, there are a number of tools around which will help you translate C code into the equivalent Pascal code. One of these is CtoP from Knowledge Software Limited. The program doesn't attempt to perform a 100% conversion but does take care of much of the spade-work for you. It's designed to take Unix System V C as input and will produce Turbo Pascal output. The program is several years old now and doesn't know anything about recent language extensions (either in C or Pascal), but it does provide a useful head start in converting a large chunk of existing C code. The shareware version is included on the disk with this issue.

Dave Jewell is a freelance consultant and programmer, specialising in systems-level work under Windows and DOS. You can contact Dave on the internet as djewell@cix.compulink.co.uk