C++ Builder For Delphi Users

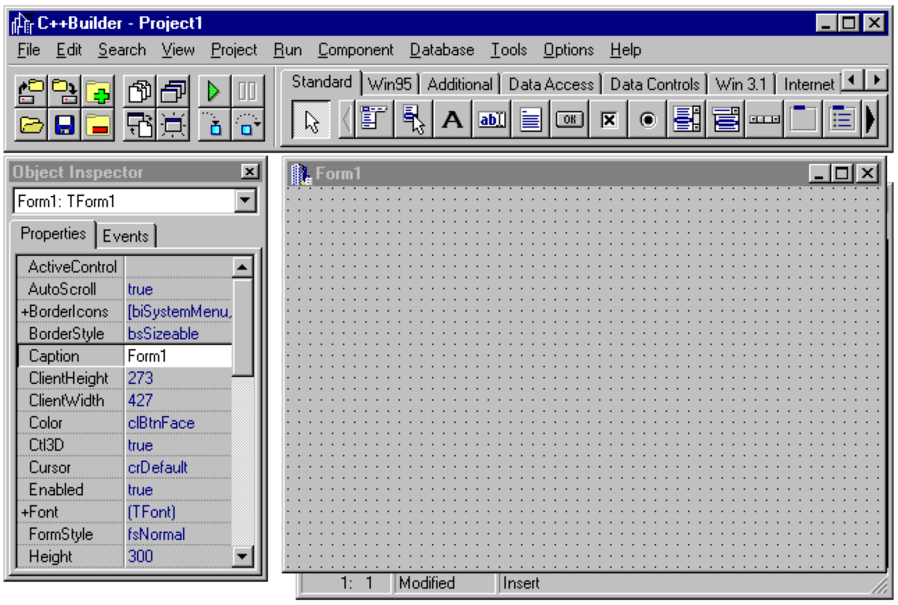
by Brian Long

C++ Builder is the third of four similarly-named RAD tools from Borland, each of which touts three major technology attributes. The products are, in release order:

- ➢ Borland AppBuilder (or Delphi as it is known, since Borland couldn't shake the code name off) which we all know and love;
- ➢ Borland IntraBuilder, a JavaScript-based RAD tool offering database-publishing facilities to Web site developers;
- ➢ Borland C++Builder, a C++-based RAD tool offering high performance with an unmatched degree of control (as C++ is renowned for);
- ➢ Borland JBuilder, a Java 1.1 based RAD tool for developing cross-platform Java applets, often for use on Web sites.

This article attempts to show Delphi users what to expect if you buy C++ Builder, and how to tackle some commonly used Delphi techniques and constructs within a C++ dialect. This can't be an all-encompassing C++ Builder tutorial (since I don't have the space), but it should give you a feel for the levels of similarity and difference.

Everything here is based on the publicly available pre-release beta version *[available on our Collection '96 CD-ROM amongst other places. Editor]*. The comments on Delphi 3 (aka Delphi97) are based on information released publicly by Borland. C++ Builder should be shipping by the time you read this.


C++Builder - Project1
File Edit Search View Project Run Component Database Tools Options Help
Standard Win95 Additional Data Access Data Controls Win 3.1 Internet
Object Inspector
Form1: TForm1
Properties Events
Form1
1: 1 Modified Insert

ActiveControl	
AutoScroll	true
+BorderIcons	[biSystemMenu,
BorderStyle	bsSizeable
Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300

➤ *Figure 1: Deja vu?*

First Impressions

When you start C++ Builder you could be forgiven for thinking you'd started Delphi by mistake (see Figure 1). This is particularly true if you set your C++ Builder shortcut up to use the `-NS` or `/NS` command-line switch, which disables the splash screen. Until you embark on a close examination of the IDE, you will be hard pressed to tell the difference between the two products.

The environment is basically a cross between Delphi 2 and 3 with a little of Delphi 1. Clearly the component palette looks just the same as it does in Delphi 2. However, most components have one or two extra properties, as sported by Delphi 3, such as `ImeMode`, for Win32 international language input support. But all of this notwithstanding, the menu structure is rather like that of Delphi 1, with an `Options` menu giving access to project and environment options. This is probably to make it more familiar to users of Borland C++, which has an `Options` menu.

The IDE is built from the same source code base as Delphi's, with a few different files pulled in here and there, so you can be confident of feeling at home. At least until you need to start writing code!

Hardware Requirements

Do not use C++ Builder on a skimpy machine! It's reasonably well known that Delphi 2 developers are advised to have a 24Mb Pentium at their disposal for sensible development. This will also be true for C++ Builder, although if you can get more memory then do so.

The most important thing to remember is to have lots of disk space available. C++ Builder is very disk-hungry. The incremental linker technology generates four files per project which will consume a lot of disk space as you move from project to project. Even on a fresh project they amount to 3.25Mb. Each time you modify the project and re-compile, they stand to increase in size. Also, there is a debugging information file which seems to have a minimum size of 768Kb.

But it's not just the files in the project directory that keep chomping megabytes out of your hard drive. In my LIB subdirectory six files have been created over the last few days, related to pre-compiled headers amongst other things, which together take up just under 19Mb.

Human Requirements

Patience. One of the many reasons Pascal was chosen as the language for Delphi was because it is so quick at compiling when compared to other well-known languages, such as C++. The first time you compile a C++ Builder project you can go and make some coffee. The pre-release version of the product I am using for this overview took 40.8 seconds to make a fresh empty project the first time. However C++ Builder implements an incremental linker to help expedite all but the first build. Having made a few changes to my fresh project, the next make took only 4.3 seconds. If you concentrate on developing one application at a time in C++ Builder, the incremental linker will keep the compile times down to a more manageable level.

Above all, you need to remember to avoid rebuilding the project (`Project | Build All`) as this takes much longer than is really bearable: all the referenced C++ header files are re-compiled and the incremental linker storage buffers are ignored. Generally, you should just make the project (`Project | Make` or `Ctrl-F9`) or run it (which invokes the make process), which will take most headers from the pre-compiled header caches that are stored here and there. My simple project mentioned above took 69.9 seconds to rebuild, just after the 4.3 second make process.

Project Overview

C++ Builder is a C++ product designed to support the RAD process and let C++ developers get access to the well established Visual Component Library (VCL). In order for the C++ Builder component library to be an accurate facsimile of the Delphi VCL, the source files which implement C++ Builder's components are all written in Delphi Pascal. No translation has been made, which should make it easy for C++ Builder to keep up with Delphi VCL developments.

C++ Builder comes with two IDE compilers for automatically compiling either C++ or Pascal source modules. This means that if you have a team developing a project then some can develop code in Pascal using Delphi and some can develop in C++ using C++ Builder. It also means that C++ Builder users can take advantage of the wealth of Delphi source code available.

C++ Builder cannot manufacture Pascal files, but you can add existing Pascal files into your C++ Builder projects.

```
#include <vcl\vcl.h>
#pragma hdrstop
//---------------------------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
USERES("Project1.res");
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    Application->Initialize();
    Application->CreateForm(__classid(TForm1), &Form1);
    Application->Run();
  }
  catch (Exception &exception)
  {
    Application->ShowException(&exception);
  }
  return 0;
}
```

➤ *Listing 1*

The main project source file itself is always a C++ file (see Listing 1): there is no getting away from that. If you examine Listing 1 you can immediately see differences and similarities to what you are familiar with in Delphi.

When you open a C++ Builder project, you do not have to pick the project source file name, as you do in Delphi. All C++ source files have a .CPP extension and so it isn't always easy to spot which file represents the project. Normally, you pick the project makefile, which has a .MAK extension. A makefile is a historical text file which can be processed by a command-line tool called MAKE.EXE. It contains various options, rules and dependencies which make sure the most up to date version of the target file can be generated with least effort by the other command-line tools. The .MAK file can still be used on the command-line, but most users will be happy to let the IDE interpret it.

To give you an idea of a project's structure and layout, Tables 1 and 2, *C++ Builder Project Files* and *Delphi Project Files*, list all the files which might get manufactured for a given project in C++ Builder and Delphi (the tables are at the end of the article, pages 18 and 19). They also highlight those which should be archived and, by implication, those which can safely be deleted.

The list shows that C++ Builder stores form code in two files, a .CPP file (source file) and a .H file (header file). In fact this is the same for any unit: it is split into two files. The .H file can be considered to be much the same as the `interface` section of a Pascal unit and the .CPP file the same as the `implementation` section. For one source module to refer to items in another source module, it (or its header) must include the other module's header file. This is achieved with a `#include` directive and is analogous to adding a unit name into the `uses` clause of the `implementation` or `interface` section in Delphi. This process can be slightly automated by using `File | Include Unit Hdr...` (or `Alt-F11`).

To access various C and C++ RTL routines you will need to include various headers from the large collection supplied in the INCLUDE directory hierarchy.

User Interface Designing

The general operation of visually designing an application is almost identical between Delphi and C++ Builder. There are some different menu captions in places and one or two extra menu items, but little else that is noteworthy. One nice thing is the addition of more keystroke shortcuts for various menu items. For example:

- ➢ `Project | Add To Project...` is `Shift-F11`
- ➢ `File | Open Project...` is `Ctrl-F11`
- ➢ `View | Call Stack` is `Ctrl-F3`
- ➢ `Project | Options...` is `Shift-Ctrl-F11`.

Also, `View | CPU` (which is available in Delphi 2 if you set up an undocumented Registry entry) is `Alt-F2`. Lastly, the equivalent of Delphi's

File | Use Unit... in C++ Builder is File | Include Unit Hdr... and that has a shortcut of Alt-F11.

Object Inspector

One example of syntax difference you will soon encounter is shown in Listing 1. To access a property or method of any VCL object you must resist the temptation to use dot notation. Delphi users use a dot because Delphi hides the fact that objects are implemented via pointers. This is not hidden in C++ and so you must use the appropriate operator: the arrow or crow's foot or *points-to* operator: a hyphen followed by a greater than symbol (->). If you use File | Include Unit Hdr... (Alt-F11) to reference another form's components, the Object Inspector reminds you of this syntax if it ever shows you a component from another form, as shown in Figure 2.

Event Handlers

These are manufactured in exactly the same way as in Delphi and a couple of examples are shown in Listing 2. There may be a good argument for not bothering with a form's OnCreate handler in C++ Builder, since you will always find the form constructor sitting in the form unit, waiting to be used. This is also shown in Listing 2, at the top.

Notice the repeated use of the __fastcall modifier. This is the same as Delphi's register keyword and implements the same calling convention as 32-bit Delphi defaults to. If you ever set up event handlers by hand, do not forget to use this modifier.

Also notice that the standard Sender parameter, as taken by practically all event handlers, is explicitly declared to be a pointer to a TObject by way of an asterisk (*). This is emphasising that VCL objects are really pointers to objects, despite what the Delphi syntax might suggest. In C++ Builder, VCL objects must be declared as pointers since they must live on the heap. You cannot declare a VCL object without the pointer syntax. This implies that stack-based VCL objects are not allowed.

Notice in the OnCloseQuery event handler in Listing 2, one of the parameters, CanClose, is declared with an ampersand (&). This is C++ syntax for a pass by reference parameter: like a Delphi var parameter. The event handler can write a value to the parameter and the code that called the event handler will see the new value.

General Language Differences

Some of the syntax problems you encounter with C++ Builder will simply be due to the C++ language. However, a lot of the code you write will basically be Delphi code with appropriate tweaks to make C++ Builder happy (in other words, C++ syntax is not a million miles away from Pascal). Some important points to remember are:

- C++ is a case-sensitive language: you must type things in the right case or you will get compiler errors!
- Strings are delimited by double quotes.
- The assignment operator is a single equals sign (=) instead of colon equals (:=) and the equality operator is a double equals (==) instead of a single equals.
- The semicolon (;) is used as a statement *terminator* in C++, whereas Pascal uses it as a statement *separator*.
- Compound statements are signified with a pair of braces (curly brackets: {}) instead of begin and end. This is a bit confusing as Delphi uses these characters for comment notation.
- When a parameter-less function is called, a pair of empty parentheses ( ) must be supplied. Delphi 1 insists on no brackets being used, Delphi 2 allows empty brackets.
- Comments can be started with /* and terminated with */, or a single-line comment can be started with // (as in Delphi 2).

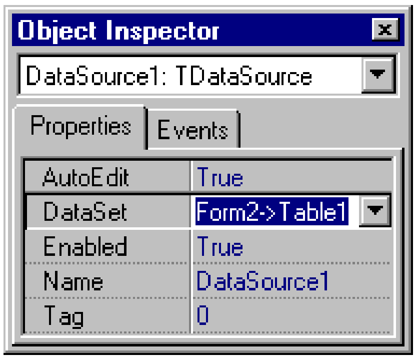
Variables And Objects

All of the pre-defined Delphi data types are defined for use in C++ Builder in the SYSDEFS.H header file, so you don't necessarily have to rush into learning all the C++ native types. The syntax of a variable declaration is (in Delphi terms) back to front, with the type specified first before the list of variable names. However, a nice feature of C++ means that you can initialise variables in declarations (both global and local) and you can also insert variable declarations almost anywhere in a subroutine.

Certain Delphi-specific types such as Pascal strings and sets are implemented as classes. I'll look at these more closely later. Listing 3 shows a few variables being declared and used.

When you need to dynamically create VCL objects in C++ Builder, you do as shown for a form object in Listing 4. This includes the declaration of a VCL object pointer (or object reference as it would be called in Delphi) and simultaneous construction using the new operator. Objects are destroyed not by calling Free(), but using the delete operator, for C++ conformance.

➤ *Figure 2*


Object Inspector

DataSource1: TDataSource

Properties | Events

AutoEdit	True
DataSet	Form2->Table1
Enabled	True
Name	DataSource1
Tag	0

➤ *Listing 2*

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  // This is the form constructor
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
}
void __fastcall TForm1::FormCloseQuery(TObject *Sender, Boolean &CanClose)
{
}
```

If Statements

The condition used in an `if` statement must be enclosed in parentheses. Listing 5 shows a couple of conditional expressions which put a message on the caption bar indicating which quadrant of the form the mouse is over.

Note the use of the += operator. There are many such operators available in C++. `X += 5` is the same as `X = X + 5` and `X++` is the same as `X = X + 1`. That particular operator (++) is where the name C++ came from: it was supposed to be one step ahead of C.

You have to be a bit careful with the expressions you write within these brackets since C++ operator precedence is a little different to that in Delphi (search for `precedence` in the two environments' Help systems for a comparison). You will find that the equality operator (==) takes precedence over the Boolean And (&&) and Or (||) operators, the opposite to how it is in Delphi.

Case Statements

Pascal `case` statements must be replaced by `switch` statements (see Listing 6). The expression you are checking the value of must again be enclosed in brackets. Each individual value is listed after the reserved word `case` and followed by a colon and the statement to execute. If you want multiple statements to execute for any case, you must enclose them within braces (just like a Pascal compound statement needs a `begin` and `end`).

If you need the same actions performed for many values, you can list one case after another, as is done for `mrNo` and `mrAbort` in the listing. Notice that you need to use `break` to stop execution going onto the next case statement. In C++, the case values are used to indicate where to start execution, not to indicate which section to solely execute.

As in Pascal, the `default` section, which executes if no other values match, is optional.

String Operations

In Delphi 2, the native string type is called `AnsiString`, but you can (and probably do) also use the `String` identifier. Strings are automatically dynamically memory-managed by RTL code, contrary to how strings work in C++. The native C/C++ string type is implemented as a `char *`, a pointer type which Delphi defines as a `PChar`. C++ also offers a string class, but it is not compatible with the Pascal `string` type.

So, C++ Builder implements a new string class called `AnsiString`, defined in the DSTRING.H header file. There is also a type called `String` in SYSDEFS.H, defined to be the same as `AnsiString`. Also, to support old-style Delphi 1 small strings, there is a template class called `SmallString` and a `ShortString` type (the same identifier as in Delphi), defined via the `SmallString` type.

Incidentally, if you do not know what a template class is, don't worry: the details are not important and I won't cover them here.

One useful side effect of strings being implemented as classes is that all strings which are declared without initial values will start off as empty strings. In Delphi, the initial value is undefined.

Another benefit is that we have many constructors supplied. When a `String` parameter is required we are able to pass an `Integer`, a `Double`, a null-terminated C string, another `String` or a null terminated wide string (designed to support Unicode). This means that we can

➤ *Listing 3*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   ShowMessage("In Button1's OnClick handler");
   Integer Count, Result = 10;
   String S = "Variables declared after a code statement";
   ShowMessage(S);
}
```

➤ *Listing 4*

```
void __fastcall TForm1::About1Click(TObject *Sender)
{
  TAboutBox *AboutBox = new TAboutBox(Application);
  AboutBox->ShowModal();
  delete AboutBox;
}
```

➤ *Listing 5*

```
void __fastcall TForm1::FormMouseMove(TObject *Sender,
  TShiftState Shift, int X, int Y)
{
  String S;
  if (Y > ClientHeight / 2)
    S = "Bottom ";
  else
    S = "Top ";
  if (X > ClientWidth / 2)
    S += "right";
  else
    S += "left";
  Caption = S + " quadrant";
}
```

➤ *Listing 6*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TForm2 *Form2 = new TForm2(Application);
  switch (Form2->ShowModal())
  {
    case mrOk: {
      Caption = "Okay";
      Color = clBlue;
    } break;
    case mrNo:
    case mrAbort: Caption = "No or Abort"; break;
    default: Caption = "Some other caption";
  }
  delete Form2;
}
```

call ShowMessage with a parameter of 12. 12 will be passed to the String constructor and the resultant string is passed to ShowMessage.

Because strings are implemented as classes, normal string routines like Pos and Insert are implemented as member functions (or methods) of the relevant classes. To get a native C string from a String object, use the c_str() member function. Listing 7 shows some examples of simple string manipulation.

Set Operations

C++ has no concept of sets and so the basic Set type is again implemented as a template class. All the various other set types are defined in terms of the Set class and the enumerated type which can go in the set. In Delphi you can create a set on the fly using a pair of square brackets and the appropriate values from the enumerated type.

Unfortunately it is rather more involved in C++ due to the class implementation of sets: you need to know the set type name. Listing 8 shows a Pascal call to MessageDlg and the equivalent C++ call. Notice that the TMessageDlgButtons class name is used to create a local set object. The << operator is used to include as many elements in the set as are required.

The Set class also defines all the other set operators, eg union (+), intersection (*) and difference (-).

Listing 9 shows an event handler which takes a set as a parameter (Shift). It allows you to Ctrl-drag from an edit control by checking if ssCtrl is in the set. To manipulate the set there are a variety of methods and operators defined, such as Contains (used in the listing).

Creating A New Set Type

Listing 10 shows how to use the Set template class to make a new set type. Notice that you specify the enumerated set type along with the lowest and highest enum value that can go in the set between angled brackets.

Open Arrays

Delphi doesn't support user defined routines which take an arbitrary number of parameters, but C and C++ do. To get over this, Delphi implements open arrays and since the C++ Builder VCL is written in Delphi, C++ must be made to support Delphi open arrays.

There are two types of open arrays. Firstly, there are those declared as, for example, array of Integer, which take an arbitrary number of values of a fixed type. The Polygon method of TCanvas takes an array of TPoints. The other type is declared as array of const and can take an arbitrary number of values of almost any type. The Format function and the FindNearest method of TTable both take these. Internally, array of const gets converted into array of TVarRec.

Listing 11 shows a Delphi event handler and the C++ Builder equivalent is in Listing 12. Three macros have been implemented to support open arrays: it is important to use EXISTINGARRAY if you are using an array which already exists in a variable. The macros are shorthand ways of manipulating instances of another couple of new classes: OpenArray and OpenArrayCount. Because of the implementation of class OpenArray, the word *arbitrary* as used in the previous paragraph should be changed to *up to 19*. The ARRAYOFCONST() macro is a shorthand way of referring to OPENARRAY(TVarRec, ()) although you do need to specify an extra pair of brackets before it'll compile.

Exception Handling

Delphi's try..except..end blocks change to C++'s native try..catch blocks. Apart from that exception handling is much the same, except that C++ can raise or throw things other than exception objects. For example, you can throw an Integer, which of course is less meaningful. See Listing 13 for an example exception handler which traps

➤ *Listing 7*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  String S = "Hello world";
  S.Delete(1, 5);
  S.Insert("Goodbye cruel", 1);
  if (S.Pos("cruel") != 0) // != matches the Delphi <> operator
    Caption = S;  // Displays "Goodbye cruel world"
}
```

➤ *Listing 8*

```
MessageDlg('Hello from Delphi', mtInformation, [mbOK, mbCancel], 0);

MessageDlg("Hello from C++ Builder", mtInformation,
  TMsgDlgButtons() << mbOK << mbCancel, 0);
```

➤ *Listing 9*

```
void __fastcall TForm1::Edit1MouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Shift.Contains(ssCtrl))
    Edit1->BeginDrag(False);
}
```

➤ *Listing 10*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  enum TDayOfWeek {dwMon, dwTue, dwWed, dwThu, dwFri, dwSat, dwSun};
  typedef Set<TDayOfWeek, dwMon, dwSun> TDaysOfWeek;
  TDaysOfWeek Weekend;
  Weekend << dwSat <<dwSun;
  if (Weekend.Contains(dwMon))
    ShowMessage("This is a long weekend");
  else
    ShowMessage("Normal weekend");
}
```

EDBEngineErrors, EDatabaseErrors and any other exception. The last catch statement also shows how to trap absolutely anything that may have been thrown.

Notice that I have chosen to use the C RTL sprintf() function instead of Format() in one handler. Also, three C++ RTL functions are being used to extract information about the source location of the exception and two C macros are used to identify the current line number and source file. This is something many Delphi users have wanted to be able to do, but have been unable to due to the lack of such information being generated by Delphi. Since the three functions are defined in a header not automatically included, we need the #include directive near the top of the file.

The help for two of these symbols states that the -xp+ compiler option must be used for them to work. This causes appropriate information to be generated by the compiler for the source code lines.

This means we must either add a #pragma option -xp+ to each of our source files, or modify the makefile. This can be done by choosing View | Project Makefile and locating the compiler flags (any line beginning with CFLAG) and adding the new option into the list. Unfortunately, these functions only appear to offer useful information if the exception was not caused by a VCL exception object. This is probably due to the VCL being written in Pascal and not having the relevant information to offer.

Resource Protection

The joyful try..finally construct is unfortunately not present in C++ Builder. Yes, you read correctly: *it's not there*. So that begs the question: how do you ensure certain bits of code get executed whether exceptions happen or not? Well, the answer is to take up the C++ approach to the problem. In C++, any local stack-based objects are guaranteed to be destroyed when the routine is exited. This is also true if an exception happens, causing the object to be tidied up and removed from the stack.

So for all those mouse cursor changes, calls to DisableControls() and EnableControls(), memory allocations and any other “resource allocation” statement pairs, we need to write a new class. The class constructor can do the allocation and the destructor can de-allocate. I can foresee many people implementing much the same resource protection, or caretaker, classes in much the same way. An example of such a class is shown in Listing 14, written with inline functions (in other words the implementation of the methods can be found in the declaration of the class).

Template classes will almost certainly aid code re-usability here, but that's another story.

Run-Time Type Checking: Is And As

In Delphi, you often typecast objects with the is and as keywords. These take advantage of RTTI and

➤ *Listing 11*

```
procedure TForm1.Button1Click(Sender: TObject);
const
  Points: array[1..3] of TPoint =
    ((X: 1; Y: 1), (X: 100; Y: 1), (X: 50; Y: 100));
  BigPoints: array[1..6] of TPoint =
    ((X: 1; Y: 200), (X: 100; Y: 200), (X: 50; Y: 300),
     (X: 151; Y: 200), (X: 250; Y: 200), (X: 200; Y: 300));
begin
  Canvas.Polygon(Points);
  Canvas.Polygon(Slice(BigPoints, 3));
  Database1.ApplyUpdates([Table1, Table2]);
  Table1.FindNearest([Edit1.Text]);
  Caption := Format(’%s (%d)’, [’Error’, 10]);
  Button1.Caption := Format(’%s (%d)’, [’Error’, 20]);
end;
```

➤ *Listing 12*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  TPoint Points[3] = {{1, 1}, {100, 1}, {50, 100}};
  TPoint BigPoints[6] = {{1, 200}, {100, 200},
    {50, 300}, {151, 200}, {250, 200}, {200, 300}};
  Canvas->Polygon(EXISTINGARRAY(Points));
  Canvas->Polygon(SLICE(BigPoints, 3));
  Database1->ApplyUpdates(OPENARRAY(TDBDataSet *, (Table1, Table2)));
  Table1->FindNearest(OPENARRAY(TVarRec, (Edit1->Text)));
  Caption = Format(“%s (%d)”, OPENARRAY(TVarRec, (“Error”, 10)));
  Button1->Caption = Format(“%s (%d)”, ARRAYOFCONST((“Error”, 20)));
}
```

➤ *Listing 13*

```
#include <except.h>
...
try
{
  //Stuff that might generate an exception
  Table2->FindNearest(ARRAYOFCONST((Edit1->Text)));
  StrToInt(“Hello”);
  throw 1;
}
catch (EDBEngineError &E)
{
  ShowMessage(“Caught an EDBEngineError”);
  throw Exception(
    “On receipt of one exception, this raises a different one”);
}
catch (EDatabaseError &E)
{
  ShowMessage(Format(“%s: %s”, ARRAYOFCONST((E.ClassName(), E.Message))));
}
catch (Exception &E)
{
  ShowMessage(“I can catch all other VCL exceptions”);
  throw; //re-raise same exception, like raise in Delphi except block
}
catch (...)
{
  //This catches all other things that get thrown
  char msg[300], format[100];
  strcpy(format, “Exception %s thrown in %s at line”);
  strcat(format, “ %d and caught in %s at line %d”);
  sprintf(msg, format,
    __ThrowExceptionName(), __ThrowFileName(),
    __ThrowLineNumber(), __FILE__, __LINE__);
  ShowMessage(msg);
}
```

➤ *Listing 14*

```
class TScreenCursorChanger
{
private:
  TCursor FCursor;
public:
  TScreenCursorChanger(TCursor Cursor) // constructor
  {
    FCursor = Screen->Cursor;
    Screen->Cursor = Cursor;
  }
  ~TScreenCursorChanger() // destructor
  {
    Screen->Cursor = FCursor;
  }
};
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  TScreenCursorChanger Obj(crHourGlass); //Local stack-based object
  //Do stuff that might cause an exception
  //Who cares if it does? The object above will still get destroyed
  //So the cursor will be changed back
  Sleep(1000);
  StrToInt("Hello"); //This will generate an exception
}
```

➤ *Listing 15*

```
{ first }
if Sender is TMenuItem then
  with TMenuItem(Sender) do
    Checked = not Checked;

{ second }
try
  with (Sender as TMenuItem) do
    Checked := not Checked
except
  on EInvalidCast do { nothing }
end;
```

➤ *Listing 16*

```
\\ first option
if (dynamic_cast<TMenuItem *>(Sender))
  ((TMenuItem *)Sender)-Checked = !((TMenuItem *)Sender)->Checked;

\\ second option
if (dynamic_cast<TMenuItem *>(Sender))
{
  TMenuItem *Tmp = ((TMenuItem *)Sender);
  Tmp->Checked = !Tmp->Checked;
}

\\ third option
if (dynamic_cast<TMenuItem *>(Sender))
{
  TMenuItem *Tmp = static_cast<TMenuItem *>(Sender);
  Tmp->Checked = !Tmp->Checked;
}

\\ fourth option
if (InheritsFrom(Sender->ClassType(), __classid(TMenuItem)))
{
  TMenuItem *Tmp = ((TMenuItem *)Sender);
  Tmp->Checked = !Tmp->Checked;
}
```

➤ *Listing 17*

```
Variant MSWord;
MSWord = CreateOleObject("Word.Basic");
MSWord.OleProcedure("AppShow");
MSWord.OleProcedure("FileNew");
MSWord.OleProcedure("Insert", Edit1->Text + "\n");
MSWord.OleProcedure("EditSelectAll");
Variant CurValues = MSWord.OlePropertyGet("CurValues");
Variant SumInfo = CurValues.OlePropertyGet("FileSummaryInfo");
//Work out how many characters we just typed in
MSWord.OleProcedure("Insert", " I Just inserted ");
MSWord.OleProcedure("Insert", SumInfo.OlePropertyGet("NumChars"));
MSWord.OleProcedure("Insert", " characters");
ShowMessage("Press Enter to terminate link to MS Word");
MSWord = Unassigned;
```

so will ensure correct results. This is the advantage they have over standard compile time type checking, although because they execute code at run-time, they are slower than compile-time type checking. Listing 15 shows two possibilities for toggling the `Checked` property to a menu item in its event handler.

There are several problems with translating these into C++. Firstly, C++ does not have direct equivalents of `is` and `as` so we need to use a different approach. The second problem is that C++ has no equivalent of the `with` clause. Lastly, the available dynamic typecast operator does not raise an `EInvalidCast` exception if the intended typecast is bad, instead it returns `0` (False in C++). The C++ construct `dynamic_cast` acts as a combination of both Delphi operators.

Some possible ways of expressing Listing 15 in C++ appear in Listing 16. You can see that compile time typecasting can be done with brackets, much like in Delphi, or with `static_cast`.

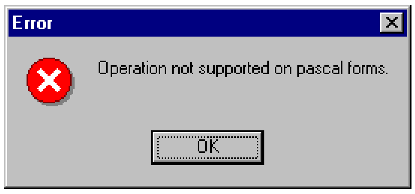
Variant Variables

C++ has no direct support for `Variant` variables and so yet another class is used to implement the required functionality. Since OLE Automation is a very common use of variants, Listing 17 has some code to automate Microsoft Word. Notice that different methods are used to get and set properties (`OlePropertyGet` and `OlePropertySet`) as well as execute a method of the OLE server you are linked to (`OleProcedure`). Indeed `Variant`s also have a method called `CreateObject`, possibly negating the need for the Delphi `CreateOleObject` equivalent.

So What About Delphi?

Having read all about the wonders of C++ Builder: being able to compile C++ and Delphi Pascal files and looking the spitting image of Delphi, you may wonder what future Delphi may have. If C++ Builder does everything that Delphi does, why would people buy Delphi any more?

Fortunately, there are a number of factors which assure Delphi's



Operation not supported on pascal forms.

➤ *Figure 3*

future, at least in the face of C++ Builder.

The first is that Delphi will always be a few steps ahead. We expect Delphi 3 to be released shortly after C++ Builder 1. Delphi 3 brings along packages and ActiveForms, to name but two of the host of new features. C++ Builder is based around the Delphi 2 VCL, with one or two items from Delphi 3 (for example the Input Method Editor support for international applications).

Secondly, despite C++ Builder being able to compile Pascal units, it is inherently a C++ product. As a result, C++ Builder project files are always C++, and C++ Builder refuses to manufacture Pascal event handlers (see Figure 3).

Also, being a C++ product it is rather slower at compiling a project than Delphi. It is also rather more disk-hungry whilst doing so.

[And the Editor couldn't resist adding this comment:
Finally, there's the language itself. C++ strikes fear into the hearts of all but the most masochistic novice programmers, whereas Pascal is more approachable and considerably easier to learn – an attribute it shares with Basic, but with arguably a better inherent structure and in the (Delphi Object) Pascal dialect there's true object orientation which is a substantial benefit over (Visual) Basic.]

Conclusions

This is a fine product which attempts to lift C++ development from the realms of the propeller head more into the mass market. Since it is based around the PME (Propery, Method and Event) component model of its multi-award-winning stable-mate Delphi, it should be a resounding success in its market sector.

It is most cunning how Borland have been able to migrate almost

➤ *Table 1: C++ Builder Project Files*

Given a project called PROJECT1.MAK where its only form unit is called UNIT1.CPP, the following files will be generated. The first six will appear when the project is first saved, the others each time the project is compiled.	
PROJECT1.MAK	Project options file, as seen by selecting `View` | `Project Makefile` containing all the options from the project options dialog's `Compiler`, `Linker` and `Directories/Conditionals` pages. This stands for MAKefile.
PROJECT1.CPP	Project source file, as seen by selecting `View` | `Project Source`.
UNIT1.CPP	Source module which implements a form's functionality.
UNIT1.H	Header file which defines a form class.
UNIT1.DFM	Binary file describing the form, all its components and their properties.
PROJECT1.RES	A resource file containing the project's icon. If no icon is specified in the project option dialog's `Application` page, a default one is supplied. This file needs to be archived if you set up a specific icon.
PROJECT1.EXE or PROJECT1.DLL	The generated executable.
UNIT1.OBJ	The compiled form of UNIT1.CPP. OBJ stands for OBJect file.
If `Options` | `Project` | `Linker` | `Use incremental linker` is enabled when the project is compiled (as it is by default), the following files are manufactured to support incremental linking:	
PROJECT1.ILC	Incremental linker storage buffer.
PROJECT1.ILD	Incremental linker storage buffer.
PROJECT1.ILF	Incremental linker storage buffer for functions.
PROJECT1.ILS	Incremental linker storage buffer for symbols.
If the `Options` | `Environment` | `Editor display` | `Create backup file` option has been selected, the following additional files will be generated each subsequent time the project is saved:	
PROJECT1.~MA	Backup of project makefile.
PROJECT1.~CP	Backup of project source file.
UNIT1.~DF	Backup of binary form file.
UNIT1.~CP	Backup of module source file.
UNIT1.~H	Backup of module source header file.
If `Options` | `Environment` | `Preferences` | `Desktop` has been selected, the following file will be generated when the project is closed:	
PROJECT1.DSK	An INI file with a different extension containing all the information required when the project is re-opened for Delphi to restore the desktop just as it was when closed.
If `Options` | `Project` | `Compiler` | `Debug information` is enabled the following file will contain all the debugger symbol information, otherwise this file will contain very little. It is principally for use by the environment but can also be used by Turbo Debugger (TD32.EXE).	
PROJECT1.TDS	Turbo Debugger symbol file.
If the `Map file` option from the `Linker` page of the `Project options` dialog is set to anything other than `Off`, the following file is generated.	
PROJECT1.MAP	Text file containing varying details of information of use when performing low-level debugging tasks.
If the command-line tool CONVERT.EXE (found in the BIN subdirectory) has been used (it converts binary form files to text files and text files back to binary forms), the following file will be present:	
UNIT1.TXT	Text file containing description of form and all components on form, including values of all non-default properties.
If there was more than one form in the project, there would be files similar to the UNIT1.* files generated for each additional form.	
The primary files to be concerned with are the .MAK, .CPP, .H and .DFM files, although the .RES file will probably also need to be saved.	

everything from the Delphi world into direct C++ equivalents, by using combinations of classes, templates, macros and other pre-processor directives.

In fact, I heard someone mention that during the latter stages of Delphi's original development, C++ Builder was already being planned. The code name for Delphi was Delphi. The code name for this new C++ product was originally Sci-Fi, suggesting its ahead of its time and out of this world nature.

So now we can see Delphi and Sci-Fi side by side (note for the English: remember that the people who wrote these products pronounce the second syllable of Delphi the same as the second syllable of Sci-Fi).

I believe people will only move to C++ Builder *from Delphi* if they have a background of C++, or because they think it would benefit them to get in with the C++ crowd. If so, then fine.

The major market for C++ Builder is likely to be those developers who would not consider a language change from C++ to Pascal, or who have a substantial base of legacy C++ code.

I wouldn't recommend you move *exclusively* to C++ Builder if you are a Delphi developer with no C++ experience, hoping that C++ Builder will give you everything that Delphi does, plus more. Although this is very nearly true in strict terms, since C++ Builder compiles C++ and Delphi Pascal source files, the length of C++ Builder's initial compile times might make you regret the decision. Developers who have used C++ compilers certainly appreciate Delphi's blinding compilation cycles. For existing Delphi developers, however, C++ Builder may well prove to be a useful adjunct, with the side benefit of allowing you to gain C++ experience in what is currently the least painful way.

➤ *Table 2: Delphi Project Files*

Given a Delphi 2 project called PROJECT1.DPR where its only unit is called UNIT1.PAS, the following files will be generated (Delphi 3 throws a couple more in for good measure). The first four will appear when the project is first saved, the fifth one should only be seen when the project is closed, the others each time the project is compiled.	
PROJECT1.DPR	Project source file, as seen by selecting `View | Project Source`.
UNIT1.PAS	Source module which represents a form.
UNIT1.DFM	Binary file describing the form and all its components. Stands for Delphi ForM.
PROJECT1.RES	A resource file containing the project's icon. If no icon is specified in the project option dialog's `Application` page, a default one is supplied. This file needs to be archived if you set up a specific icon.
PROJECT1.DOF	(Delphi 1 uses the .OPT extension) .INI file with different extension containing all the options from the project options dialog's `Compiler`, `Linker` and `Directories/Conditionals` pages as well as anything specified in `Run | Parameters`. If you change any of these settings, this file should be kept. Stands for Delphi Options File.
PROJECT1.EXE or PROJECT1.DLL	The generated executable.
UNIT1.DCU	The compiled form of UNIT1.PAS. DCU stands for Delphi Compiled Unit.
If the Delphi 2 option `Tools | Options | Display | Create backup file` or the Delphi 1 option `Options | Environment | Editor display | Create backup file` has been selected, the following additional files will be generated each time the project is saved:	
PROJECT1.~DP	Backup of project source file.
UNIT1.~DF	Backup of binary form file.
UNIT1.~PA	Backup of module source unit.
If the `Desktop` option has been selected from the `Preferences` page of Delphi 1's `Options | Environment` or Delphi 2's `Tools | Options` dialog, the following additional files will be generated when the project is closed (the latter only if `Desktop and symbols` is selected in the same options page):	
PROJECT1.DSK	An .INI file with a different extension containing all the information required so that when the project is re-opened Delphi can restore the desktop just as it was when closed.
PROJECT1.DSM	Symbol table for your application. If you load your project, Delphi will not need to compile your project to enable the Object Browser etc.
If the `Map file` option from the `Linker` page of the project options dialog is set to anything other than `Off`, the following file is generated.	
PROJECT1.MAP	Text file containing varying details of information of use when performing low-level debugging tasks.
If the command-line tool CONVERT.EXE (found in the DELPHI\BIN directory) has been used (it converts binary form files to text files and text files back to binary forms), the following file will be present.	
UNIT1.TXT	Text file containing description of form and all components on form, including values of all non-default properties.
If there was more than one form in the project, there would be files similar to the UNIT1.* files generated for each additional form.	
The primary files to be concerned with are the .DPR, .PAS and .DFM files, although others that may need to be saved are the .RES and .DOF/.OPT files.	

Brian Long is a UK-based freelance Delphi consultant and trainer, although he now also has Borland C++ Builder on his CV. He is available for bookings and can be contacted by email at blong@compuserve.com