Borland C++ 5.0 ObjectScripting

Ted Faison
96/4/25

The Borland C++ 5.0 IDE is a rich workspace, supporting multiple windows, toolbars, editors and installable tools. Previous versions of the IDE allowed programmers to customize the IDE in a limited way, by allowing you to select which buttons were on the toolbar and letting you install external commands that were invoked through the Tool menu. Users could also choose editors that emulated popular programs like WordStar, Brief and Epsilon. This wasn't enough, so we added the capability to fully customize the editor using key scripts. Our customers wanted more: the ability to control syntax highlighting, hooks into the debugger, control over the editor. The list just kept growing, and the only way to satisfy everyone was to create a way to customize the entire IDE in a programmatic way. ObjectScripting was born.

The first objective of ObjectScripting was to allow developers to customize the IDE without needing to recompile any code – especially the IDE code itself. The second goal was to allow users to change the IDE in arbitrary ways, with no predefined limits. We wanted all this to be possible with a minimum of complexity. It was clear from our experience with editor key customization that scripts would be a good approach. What was needed was a new scripting language that supported features such as classes, late binding, object-specific method overriding and dynamic variable typing. The result was a new language called cScript. The 'c' in the name was added because the language looks immediately familiar to C++ programmers.

The IDE uses ObjectScripting scripts internally to implement much of its functionality. This default code is the point of departure for user customizations. By adding ObjectScripting code of your own, you can install new tools, customize nearly any part of the IDE, and have your own code run automatically when a key is pressed. Through ObjectScripting, Borland C++ 5.0 now offers an unprecedented level of customization, that lets developers adapt the IDE to their individual tastes and preferences.

ObjectScripting Objects

cScript makes it possible to control the IDE because all the objects in the development environment are exposed through the global object called `IDE`, of class `IDEApplication`. This object is created by code in the `STARTUP.SPP` file, which is automatically executed when ObjectScripting starts. You gain access to the various parts of the IDE, such as the `KeyboardManager` or the `Editor` window by accessing properties of the `IDE` object. To set the text of the status bar, you simply make an assignment to the `StatusBar` property of `IDE`, like this:

```
IDE.StatusBar = "My Text";
```

cScript properties are similar to Delphi ObjectPascal properties. Properties are like C++ data members, except they are accessed internally through functions called *getters* and *setter*s. When you use a property as an Rvalue, the getter function is called to obtain the value of the property. The getter can then return the property value directly, or do something a bit more elaborate like make format conversions, notify other objects, update the screen or whatever. Using a property as an Lvalue implicitly invokes the property setter.

The bottom line is that properties make it possible for simple assignment statements to produce side effects, without you having to worry about the details. In the status bar example above, you just set the text to a new value. You don't have to worry about refreshing the screen, or whether

the status bar is visible or not. The `StatusBar` string object's setter function handles the details automatically.

There are ObjectScripting objects that encapsulate most items in the IDE. There are classes to access the Search engine, the editor, the keyboard manager, to create and extend popup menus, and more.

Executing Scripts

ObjectScripting programs are written using a plain text editor and saved as `.SPP` files. Scripts are run by the ScriptEngine, and can access any object used by the IDE, like the text editor or the debugger. cScript programs can also access C runtime functions, use Windows interface objects like dialog boxes and messages, and produce text output on the screen. To enhance performance, when ObjectScripting encounters a script for the first time, it produces a tokenized file with the extension `.spx`, which is then run. Subsequent runs of the same script use the `.spx` file and load faster.

Once you have a `.SPP` script source file, you can run it in one of three ways:

1 - By using the **Script | Run** command on the main menu.

Doing so opens a small window that looks like this:

Figure 1 - The Script | Run window.

You can type any valid ObjectScripting code in the window. When you press the `ENTER` key, the code is executed and the output (if any) is shown in the **Message** Window, on the **Script** tab.

2 - By binding your code to a key.

The easiest way is to put code in a file called `PERSONAL.SPP` script file. Using your own script file – as opposed to adding code to one of the Borland script files – protects your code from being overwritten by new releases of the IDE. Moreover, your code is available for use regardless of which keyboard emulator you use. Say you want the function `Abc()` to be called when the `HOME` key is pressed. All you have to do is write:

```
(IDE.KeyboardManager.GetKeyboard("Desktop")).Assign("<Home>",
"Abc();");
```

Another way to bind your code to a key is to add your function to the keyboard file, which will be described later. To make the system invoke the function `Abc()`when the `HOME` key is pressed, just go into the file `default.kbd` with a text editor and put the function `Abc();` in the `KeyCommand` field for the `HOME` key.

3 - By attaching code to an event.

ObjectScripting allows you to attach static or dynamic events handlers to execute your own code. Handlers are described in more detail a little later. Events may be raised either internally by the IDE or externally by a user script.

A Simple Example

The easiest way to show what ObjectScripting can do for you is through an example. Borland C++ 5.0 ships with a number of ObjectScripting sample programs that you can use immediately or as the basis for your own customizations. There are also a number of scripts used to implement much of the IDE functionality. For example, the file `EDIT.SPP` contains the script code for the Borland C++ editor. It has code that manipulates the contents of the editor window, allowing you to select blocks of text, move the cursor, scroll the window, etc. For example, the code to indent a block of text is implemented by the function `SlideBlock`, that looks like this:

```
on editor:>SlideBlock(backward) {
   declare eb = .BlockExists();
   if (eb != NULL) {
      declare nIndent = .Options.BlockIndent;
      if (backward) {
        nIndent = nIndent * -1;
      }
      eb.Indent(nIndent);
   } else
      IDE.StatusBar = "No block exists";
}
```

The function creates a handler for the `editor` object, which is an ObjectScripting object that encapsulates the Borland C++ 5.0 text editor. The call `.BlockExists` calls the member function `editor.BlockExists`. This function is not declared inside class `editor`, but added dynamically to the class at runtime. You can add your own member functions to the built-in ObjectScripting classes without needing to recompile those classes. The variable returned by `BlockExists` is an object of class `EditBlock`, through which you can manipulate selected text. If no block is selected, `BlockExists` returns `NULL`. If there is selected text, the call `Options.BlockIndent` gets the number of space characters corresponding to one indent. This value is set by the user through the **Options | Environment | Editor | Options** window. The call `eb.Indent` invokes the function `EditBlock.Indent` to actually move the selected text. All the functionality of the Borland C++ 5.0 editor is implemented by ObjectScripting code. There are no tricks, no hidden variables or undocumented functions of any kind. Anything the editor is capable of can be controlled using ObjectScripting.

Editor key mapping

The text editor windows are controlled by keyboard commands, but the command invoked by a given key sequence is not hardcoded into the system. The way key sequences are handled is determined by ObjectScripting code that executes code bound to a key sequence. Files called *keyboard files*, with the extension `.kbd`, define the key-command mapping. By default, the IDE uses the file `default.kbd`, which maps keys according to the Microsoft Common User Interface Architecture guidelines. By selecting a different keyboard file, you can emulate other popular editors or even create your own key mappings. Borland C++ 5.0 ships with files to emulate Brief, Epsilon and WordStar commands. To change the keyboard file, you use **the Options | Environment | Editor | File** command.

When the IDE starts up, it automatically executes the ObjectScripting code in the file `startup.spp`. This file contains the code

```
LoadKeyboard() {
  declare String sFileName(scriptEngine.StartupDirectory +
                           IDE.KeyboardAssignmentFile);
  LoadKeyboardDataFile(sFileName.Text,FALSE);
}
```

The string `sFileName` is initialized to contain the path of the keyboard specified by the user. The `LoadKeyboardDataFile` function resides in the file `KEYMAPS.SPP`. This function clears any previously-made assignments and then processes the data it finds in the `.KBD` file. This data associates key-names with scripts, and part of the processing performed by `LoadKeyboardDataFile` is to make the necessary assignments to the keyboard objects, such that pressing a key will trigger the associated script. For example, assume the `default.kbd` file is being used. When the user hits the `HOME` key, the associated script is invoked, which looks like this:

```
editor.TopView.Position.MoveBOL();
```

The code uses `editor.TopView` to retrieve the `EditView` object associated with the active editor window. Its `Position` property returns an `EditPosition` object, through which the cursor position is changed by calling the `MoveBOL` function. As a result, the cursor is moved to the beginning of the line containing the cursor.

Had the user selected a different keyboard file, a different command might have been executed for the `HOME` key. For example, the `Epsilon.kbd` file maps the `HOME` key to the script:

```
editor.EmacsBeginningOfWindow();
```

which moves the cursor to the beginning of the first line displayed in the editor window.

Controlling the IDE

The text editor is not the only thing you can control through ObjectScripting. Any feature of the IDE is accessible. You can add your own commands to the top level menu. Say you want to add your own **Spell Check** command. The code

```
#define TEXT         "&Tool | Spe&ll Check"
#define DESCRIPTION  "Runs a custom spelling checker”
assign_to_view_menu("IDE", TEXT, "RunSpellChecker();", DESCRIPTION);
```

adds a menu item labeled **Spell Check** to the **Tool** menu. The `DESCRIPTION` string specifies what to show on the status bar when the user selects the command. When the **Spell Check** command is executed, ObjectScripting calls the user function `RunSpellChecker`, which then will take charge of the actual spell checking, possibly by invoking a user-supplied DLL.

Debugging an ObjectScripting script

Because you can develop arbitrarily complex cScript programs, there may be times when a program doesn’t immediately do what you intended. ObjectScripting provides you with debugging support, allowing you to set breakpoints, single step through code and inspect variables. To set a breakpoint, all you do is embed the statement

```
breakpoint;
```

in your program. When the system hits a breakpoint, it displays the **Breakpoint Window**. As an example, say you put a breakpoint in the `GetWord` handler for `editor`, contained in the file `EDIT.SPP`. The code looks like this:

```
on editor:>GetWord() {
```

```
   breakpoint;
   if(!initialized(.TopView)){
      // no edit window exists
      return "";
   }
 // ...
 // ...
}
```

When `GetWord` is executed, script execution stops on the breakpoint line, and the **Breakpoint Window** comes up, as shown in Figure 2.

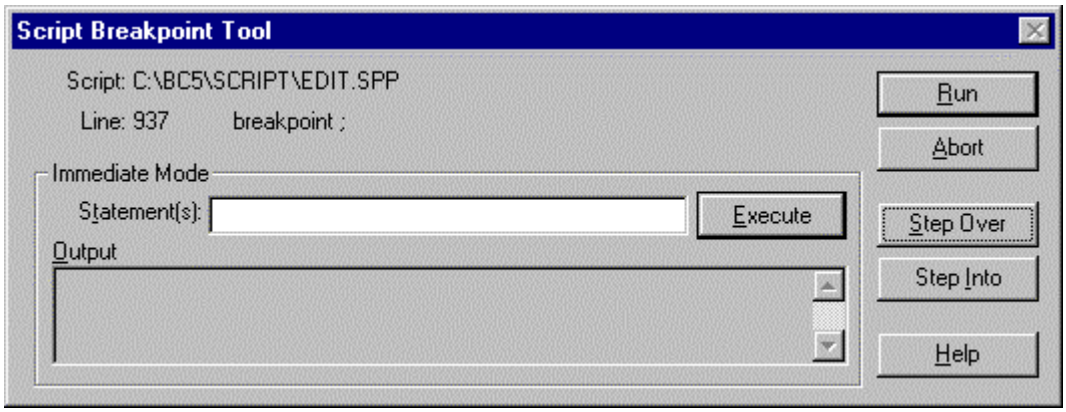

Figure 2 - The ObjectScripting Breakpoint Window.

Once you hit a breakpoint, you can single step through your script code. The **Line** field shows you the line number and cScript code that was last executed. Using the **Immediate Mode** controls, you can execute ObjectScripting commands to inspect variables or test code before adding it to the ObjectScripting program. Figure 3 shows an immediate statement that returns the number of lines in the active edit window.

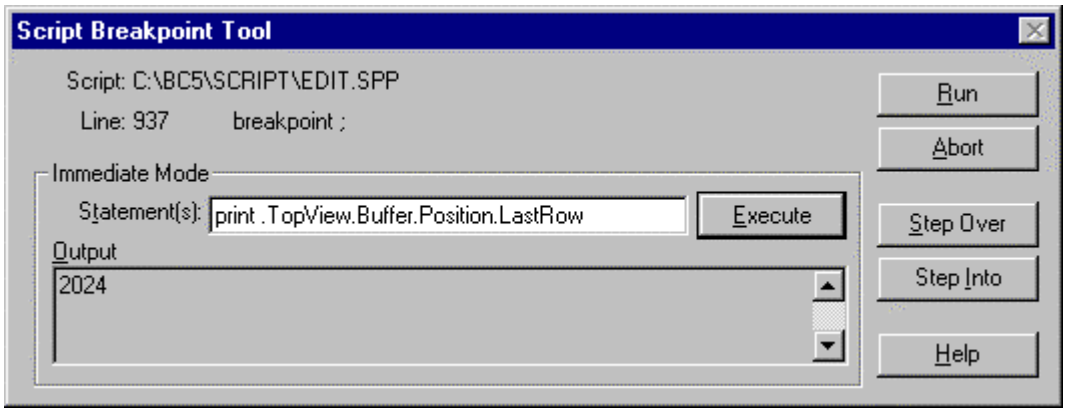

Figure 3 - Browsing the system using Immediate Statements.

The **Step Over** button lets you execute complete functions at full speed, with execution stopping after the function returns. The **Abort** button quits the script, unloads the ObjectScripting debugger and returns control to the IDE.

Capabilities exposed through the IDE object

When your start the Borland C++ 5.0 IDE, the cScript object `IDE` is automatically created as a global object. `IDE` gives you control over the system not only through its `KeyboardManager` and `Editor` members, but also through a rich set of member functions. Using function calls you can control the Debugger, invoke Edit commands, perform File operations, etc. All the items contained in menus can be accessed using ObjectScripting code. Table 1 shows the main function groups, organized according to the menu they correspond to.

Function Group	Description
Debug	Allows control over the debugging process. Using functions in this group you can load the debugger, run it, set breakpoints, add watches, inspect variables, stop the debug process and more.
Edit	All the commands in the Edit menu are available here. You can cut, copy, paste, undo, redo and select all the text in a given window.
File	This group of functions lets you open, close, save and print files. You can also open your own **FileDialog** common dialog to perform file operations, or even send a file by mail to another user.
Help	All the Help menu commands are available in this group, so you can launch the Help engine, display the Borland C++ 5.0 Help Contents or open the various specialized help files on OWL, the keyboard or the Windows API.
Options	Using Options functions, you can access the project options, the environment options, change editor preferences, add your own tools to the Tool menu, and modify your project's Style Sheets. To fill in the various Options dialog boxes, you use the function `SendKeys` to feed keystrokes into the keyboard buffer.
Project	These commands let you compile a file, build the project or rebuild the whole project.
Script	These commands allow you load, run and compile cScript files.
Search	Using these commands you can search for text, repeat a search and replace text.
View	These commands give you control over which windows in the IDE are visible. You can show or hide the Breakpoint window, the Stack window, the Classes window, the CPU window, the Global variable window and more.
Window	These commands allow you to arrange the editor windows, by cascading, tiling, closing, minimizing or restoring them.

Table 1 - The menu-related IDEApplication member functions.

Besides the functions in Table 1, there are a number of miscellaneous others that don't directly correspond to menu commands. Table 2 describes some of them.

Function Name	Description
`CloseWindow`	Lets you close the active window
`DirectionDialog`	Displays a dialog letting the user specify a direction: Up, Down, Left, Right.
`DirectoryDialog`	Displays a directory-browsing dialog allowing the user to specify a directory.

`DoFileOpen`	Lets you hook what happens after a common File Open dialog box closes.
`ExpandWindow`	Expands the size of a window.
`GetWindow`	Returns a window
`Message`	Displays text in an IDE-style message box.
`Quit`	Terminates the IDE and shuts down the entire Borland C++ system, without saving or prompting. To exit and prompt the user to save changes, you can use `FileExit`.
`StartWaitCursor`	Allows you to set the cursor to an hourglass. The function `EndWaitCursor` restores the normal cursor.
`Tool`	Invokes a given tool, passing optional parameters to it.
`YesNoDialog`	Displays a dialog that prompts the user for a Yes/No answer to a question.

Table 2 - The miscellaneous member functions of the global object IDE.

There are also numerous properties, like `StatusBar`, that allow direct and simple access to different kinds of information. Table 3 gives a list of the main properties of the `IDE` object.

Property Name	Description
`Application`	This is a property required by the Microsoft guidelines for automation servers. Its purpose is to provide something that an automation controller, like Word or Excel, can use as a starting place to access the functionality of the server.
`Caption`	Returns the caption of the main IDE window, in the form “Borland C++ - %s”, where the %s is the name of your project. You can also change the caption by assigning a value to `Caption`.
`CurrentDirectory`	Returns the current directory used to open files.
`Editor`	Provides access to the properties of the editor window.
`KeyboardManager`	Allows access to the IDE keyboard subsystem.
`StatusBar`	Allows you to get/set the text displayed on the status bar.
`Version`	Returns the version of the Borland C++ environment.

Table 3 - The main properties exposed by the global object `IDE`.

There are also a number of events that you can handle, to customize what Borland C++ 5.0 does at various times, such as when starting a build or exiting back to Windows. The following table lists the events.

Event	Description
`BuildComplete`	Raised at the end of a build. By default the system plays a `.wav` file.
`BuildStarted`	Raised when the user starts a build.
`DialogCreated`	Raised every time Borland C++ 5.0 shows a dialog box to the user.
`Exiting`	Raised when the IDE is exiting.
`HelpRequested`	Raised when one of the Help methods of the `IDE` object is called.

`Idle`	Raised if the IDE is left idle for more than a certain user-definable interval.
`KeyboardAssignmentsChanging`	Raised after the user selects a new keyboard assignment file, but before the file is actually changed.
`KeyboardAssignmentsChanged`	Raised after the keyboard file name has been changed.
`MakeComplete`	Raised at the end of a MAKE.
`MakeStarted`	Raised when a MAKE is started.
`ProjectClosed`	Raised when a project file has been successfully closed.
`ProjectOpened`	Raised when a project file has been successfully opened.
`SecondElapsed`	Raised once every second.
`Started`	Raised after the IDE has been loaded and initialized and all startup scripts have been processed.
`SubsytemActivated`	Raised when the active subsystem is changed.
`TransferOutputExists`	Raised when a transfer tool has created output that needs processing.
`TranslateComplete`	Raised at the end of a translation.

Table 4 - The events raised by ObjectScripting.

The `IDE` object can also be accessed by non-ObjectScripting programs. The object is registered as a Windows automation server, so any automation controller can drive the full Borland C++ 5.0 IDE programmatically.

A very significant part of the overall functionality of the Borland C++ 5.0 IDE is implemented through ObjectScripting code, which leaves the door wide open for developers to go in and make customizations without any predefined limits or restrictions.

The cScript Language

The key to ObjectScripting is cScript, which is a powerful, object-oriented language. You declare classes and provide them with properties and member functions. As explained earlier, properties are equivalent to C++ data members that use get and set functions as implicit accessors. There are no access specifiers for members, so everything is always public. A common practice in cScript is to derive classes from the built-in classes to override one or more functions. You declare classes using this notation:

```
class EditorKeyboard(name) : Keyboard(name)
{
  //…
};
```

This looks a lot like a C++ class, but with important differences. There are no constructors as in C++. Any code appearing inside the class declaration that is outside any function definition is executed as the class initialization code. The class name can have an optional parameter list, which declares the arguments used in the initialization code.

Variables are not typed at compile time. When you declare a variable, you don’t indicate a type. The ObjectScripting runtime engine determines types at runtime. Note that the parameter `name` declared with class `EditorKeyboard` is untyped. The same variable may also be used over again with a different type. The code

```
declare myVariable = 100;
myVariable = “Now it’s a string”;
```

declares a generic variable that has an integer type. The variable is then assigned a string, changing its type. The cScript runtime engine knows how to perform certain type conversions at runtime. You can assign a string number to a integer, or an integer to a string. For example the code

```
declare myString = “String”;
myString += 100;
```

will make `myString` assume the value “String100”. cScript supports C-style arrays, but the array contents can also be heterogeneous, meaning each slot can contain data of a different type. The code

```
declare myArray;
myArray = new Array [10];
myArray [0] = “Hello, “;
myArray [1] = 5;
print myArray [0] + myArray [1];
```

produces the output:

```
Hello, 5
```

The `new` operator is used to allocate storage for an object and initialize it. The ObjectScripting engine performs automatic garbage collection, so you don’t have to worry about memory leaks.

You can create special arrays known as *associative arrays*, in which array slots are accessed using strings instead of integers. Associative arrays make it easy to create dictionaries or symbol tables. The following code creates and initializes an associative array:

```
declare TelephoneNumbers;
TelephoneNumbers [“Ted”] = “555-1111”;
TelephoneNumbers [“Sue”] = “(213) 555-2222”;
```

To lookup a telephone number you can do something like this:

```
declare number;
declare name = “Sue”;
number = TelephoneNumbers [“Ted”];
number = TelephoneNumbers [name];
```

cScript has control statements that are similar to C++, including `switch, if, else`, `do`, `while` and `for`. The following is valid cScript code:

```
if (a > 100)
{
  // … do something…
}
else
{
  // … do something else…
}

for (declare i = 0; i < 10 + 1; i++)
{
  // .. do something…
}

declare j = 10;
while (j > 0)
{
  // … do something…
```

```
  j--;
}
```

The notation was purposely kept as close as possible to C.

Function overriding

In C++ you override member functions by class derivation. If class `A` has the virtual member function `ABC(),` you can override it by deriving a class `B` from `A` and adding a member function called `ABC()` to it. You can do this in cScript , but you also have two additional options: using closures and dynamic overriding. The best way to describe a closure is through an example. Say you want to override the `Exiting` event of the `IDE` object, to play a `.WAV` sound file when the user exits Borland C++ 5.0. Using a closure operator, you override the `IDE::Exiting` function like this:

```
on IDE:>Exiting()
{
  if (ExitingWAV != "None") {
    declare wav = new TWAVFile(ExitingWAV);
    wav.Play();
  }
  return pass();
}
```

The special cScript `:>` operator is called the *closure* operator and is used to create an `on` handler. On handlers and closure operators define a member function, not for a class, but for a specific instance of a class – in this case for the `IDE` instance of class `IDEApplication`. When the `Exiting` event is raised on object `IDE`, the custom `Exiting` function is invoked. To invoke the regular function `IDEApplication::Exiting`, you use the `pass()` call, passing all the parameters received by your overriding function to the base function. A significant advantage of `on` handlers is that they let you override a function without the hassle of creating a completely new class.

When you create an `on` handler using the closure operator, your handler permanently overrides the corresponding function in the base object. Another way to override a function is to use `attach` and `detach`. Using `attach` you can essentially hook a function in another object and replace it with your own function. The advantage of `attach/detach` over on handlers is that you have the opportunity to change overriding handlers at runtime. The hooking operation is completely dynamic, and the other object doesn't even have to be recompiled. For example, to override the `Exiting` event of the IDE object, you could do this:

```
class MyIDE : IDEApplication
{
  //…

  // declare a member function
  Exiting()
  {
    if (ExitingWAV != "None") {
      declare wav = new TWAVFile(ExitingWAV);
      wav.Play();
    }
    return pass();
  }
};

MyIDE x;
if (debugger.HasProcess)
  attach x:>Exiting to IDE:>Exiting;
else
```

```
  detach x:>Exiting from IDE:>Exiting;
```

The code that does the attaching and detaching would be written inside some user object. As script modules are loaded and unloaded, their handlers (both static and dynamic) are propertly fixed-up and maintained. The ObjectScripting system allows you to write functionality that resides completely in one module, and have other modules load/unload that module on an as-needed basis.

DLLs are on tap

cScript supports calling functions contained in DLLs. If you can't accomplish a certain task using cScript code alone, or if you already have some code in a DLL, then it may make sense to invoke DLL code. You can even call the C runtime library, contained in the file `cw3220mt.dll`. To invoke a function in a DLL, you use the `import` keyword. You then define function prototypes for the functions you want to use. For example, if you want to use the `fopen()` function, you declare it like this:

```
import "cw3220mt.dll" {
  long fopen  (const char * __path, const char * __mode);
}
```

In your cScript file, you can then access `fopen` like this:

```
declare myFile;
myFile = fopen("notes.txt", "w");
```

Can you call **pascal** functions from cScript ? Absolutely. When you declare a function imported from a DLL, ObjectScripting determines at runtime the appropriate calling convention – `pascal`, `cdecl` or `stdcall` – that applies to that function. To call a Delphi or Windows **pascal** function, you would declare it like this:

```
import "somefile.dll" {
  int SomeFunction(const char*, int);
}
```

You would then call the function normally:

```
declare result;
result = SomeFunction(“Hello”, 5);
```

You can also call Windows API function just as easily. For example to call the function `GetWindowsDirectoryA`, you would add the import declaration

```
import "kernel32.dll" {
  int GetWindowsDirectoryA(char *, int);
}
```

to your code, then invoke the function with standard notation like this:

```
declare winDir;
declare ret = GetWindowsDirectoryA(winDir, pathLength);
if (!ret)
{
    IDE.Message("Could not get the windows directory.");
}
```

Notice how the variable `winDir` is declared and used. It is declared as an untyped variable, but then used as a `char*` in the function call. ObjectScripting doesn't require you to allocate a buffer

to pass to `GetWindowDirectoryA`, because the system automatically allocates a temporary internal 4KB buffer for you to use in the function call. The resulting code is straightforward. With the ability to call DLL functions, cScript takes all stops out of script programming and IDE customizing.

Another Example

Let's look at an entirely different type of script. One that feeds keystrokes to the IDE, simulating user typing. Keystrokes have the potential to call ObjectScripting functions, as defined by the keyboard file, so you can always call those functions directly, instead of simulating keystrokes. Nevertheless, keystrokes may produce dialog boxes that are easy to fill out with ObjectScripting code, and may sometimes allow you to save some coding.

Say you want to get a list of files that contain some string pattern. You could use the integrated Grep tool, but you can also write a simple cScript program to get the same results. `DIRTOOL.SPP` is a simple script program that ships with Borland C++ 5.0. It demonstrates the basic concepts of searching directories and printing results in the Message window. When you run `DIRTOOL`, the script installs a new tool called **Directory Listing** on the **Tool** menu. By selecting this new **Tool** command, the dialog box shown in Figure 4 will be displayed.

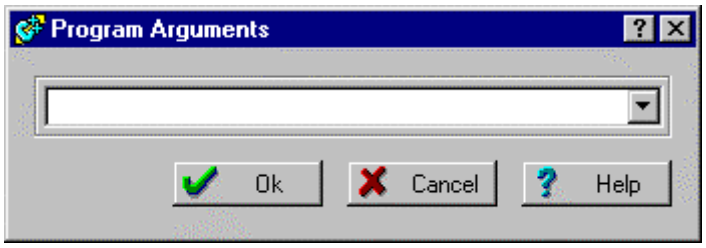

Figure 4 - The dialog box displayed by the DIRTOOL script.

A brief explanation of the `DIRTOOL` script program will help. The code in `DIRTOOL` itself doesn't actually implement the functionality of the tool. All `DIRTOOL` does is automate the process of adding a tool to the IDE's list of internal tools. It does this by driving the user interface of the **New Tools** dialog box using the function `SendKeys`. The code to add this tool looks like this:

```
IDE.KeyboardManager.SendKeys("%n", TRUE);
IDE.KeyboardManager.SendKeys("Directory Listing", TRUE);
Tab();
IDE.KeyboardManager.SendKeys("command.com", TRUE);
Tab();
declare keys = "$NOSWAP $CAP MSG(FILENAME) /c dir /b $PROMPT $SELNODES";
IDE.KeyboardManager.SendKeys(keys, TRUE);
Tab();
IDE.KeyboardManager.SendKeys("Directory Listing", TRUE);
Tab();
keys = "Creates a directory listing in the message window.";
IDE.KeyboardManager.SendKeys(keys, TRUE);
IDE.KeyboardManager.SendKeys("{VK_RETURN}", TRUE);
IDE.KeyboardManager.SendKeys("%c", TRUE);

IDE.OptionsTools();
```

Listing 1 - The cScript code in DIRTOOL that create a new tool in the Tool menu.

The code uses the `IDE` object to access objects contained inside the Borland C++ 5.0 Integrated Development Environment. `IDE` is a global object, allocated automatically when ObjectScripting starts up. The `KeyboardManager` is an object that manages keys typed by the user. The `SendKeys` method is used to simulate keys typed by the user. Using `SendKeys`, a number of keystrokes are put into the keyboard buffer, and used later to fill out the fields in two dialog boxes. The first dialog box is displayed by the call `IDE.OptionsTools()`.

The dialog shown is the same one created by the **Options | Tools** menu command, and looks like this:

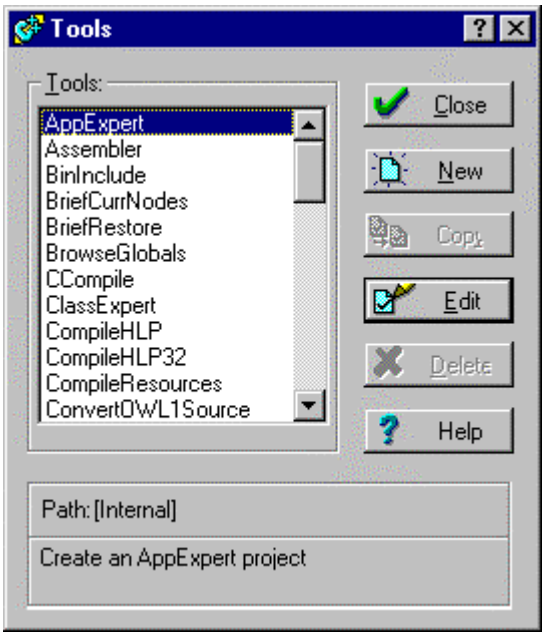

Figure 5 - The Options | Tools dialog box.

Figure 5 is the dialog that allows you to configure your own tools on the **Tools** menu. The first key placed in the keyboard buffer by `DIRTOOL` is “%n”, which equates to the **Alt-N** accelerator key. This key causes the **New** button to be pressed on the **Tools** dialog. This command causes the **Tool Options** dialog to open, looking like this:

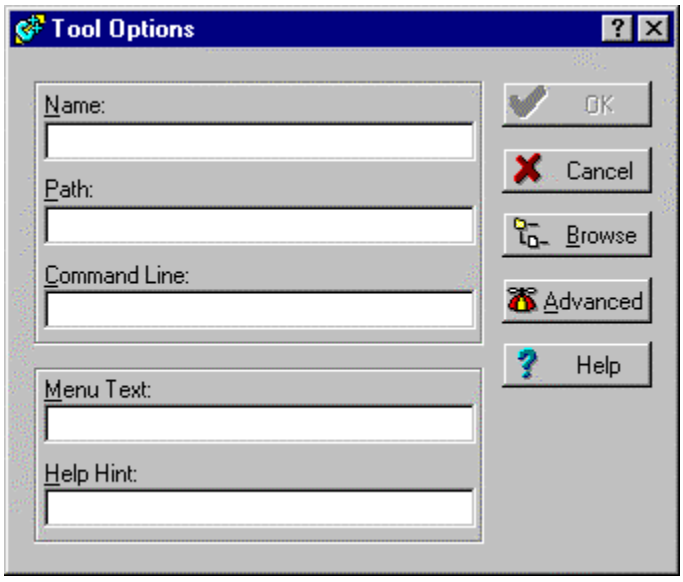

Figure 6 - The Tool options dialog box.

The remainder of the commands sent by `DIRTOOL` are used to fill out the fields in this dialog box. For example the code

```
IDE.KeyboardManager.SendKeys("Directory Listing", TRUE);
```

puts the text “Directory Listing” in the **Name** edit box. The code

```
Tab();
```

moves the focus to the next field. After filling out all the fields, the code

```
IDE.KeyboardManager.SendKeys("{VK_RETURN}", TRUE);
```

simulates the **Enter** key, which closes the **Tool Options** box. Then the line

```
IDE.KeyboardManager.SendKeys("%c", TRUE);
```

closes the parent **Tools** dialog box, by sending the **Alt-C** accelerator to press the **Close** button.

The result of running the **Directory Listing** tool is entirely equivalent to creating a tool manually with the fields shown, and running it from the **Tools** menu. You can develop script programs that automate any aspect of your work. For example, the IDE editors use scripts to process key strokes. By switching scripts, the IDE can easily change from Brief emulation to Epsilon emulation or other. The ObjectScripting environment is designed to encourage you to create your own keystroke handling scripts.

Conclusion

Borland has provided many ways to customize parts of the IDE in previous versions of Borland C++, but developers still needed more flexibility. Now, with ObjectScripting, we have opened up the entire IDE to customization and control. Using simple cScript programs that are easy to write and debug, you can now take control over all parts of the integrated development environment. With the ability to call Tools, use high level Editor functions to search and modify text, the capability to install static or dynamic handlers, the sky is the limit. The numerous sample programs shipped with Borland C++ 5.0 provide good examples cScript programming, but they represent only the tip of the iceberg of what you can accomplish with ObjectScripting.