Scientific Programming with Borland's C++Builder

Clopper Almon
January 1998

Scientists who had for years done their own programming in Fortran, C, or C++ found themselves effectively shut out of Windows programming by the huge amount of material which had to be mastered and by the volume of detailed work that had to done to use the Windows Application Programmer's Interface (API). Visual Basic brought Windows programming in Basic within the grasp of non-specialists, but programmers accustomed to the power of C++ were not likely to be attracted to programming in Basic. Borland's Delphi did the same thing for Pascal, but programmers with thousands of lines of C++ code were hardly likely to want to move it to Pascal, especially since they would have to give up such important features of C++ as overloading of constructors and multiple inheritance. Borland's Object Windows Library (OWL) and, to a lesser extent, Microsoft's Visual C++ helped the C++ programmer somewhat, but still left an immense amount of tedious work to be done. Finally, in 1997, Borland's C++ Builder brought programming for Windows within the grasp of ordinary scientists, not specialists in programming. Unfortunately, the documentation which accompanied Builder was not written with their needs in mind. These notes are intended to help them over the initial hurdles.

The documentation that comes with Builder does cover adequately the basic techniques of creating forms and putting components, such as buttons, memos, and menus on them. We assume that the reader is familiar with these techniques from *Teach Yourself C++ Builder in 14 (or 21) Days*, which comes with the software. The useful material is in Days 1, 5, and 6. We assume that the reader has worked through these Days and is familiar with C++.

Here is a quick review of Builder vocabulary. One of the rectangles with three little squares in the upper right corner which you see on the screen of a Windows program is a *form*. On a form there may be one or more *components*. Components include a menu, labels, buttons of all sorts, check boxes, radio buttons, edit boxes, list boxes, combo boxes (combinations of an edit box with a list box), common dialog boxes (for opening files, for example), panels (used to group other components), bevels (for visual effect), and memos, which are multiline editors, to mention only the most common ones. Components can be thought of as if they were C++ classes. They have accordingly data elements, which are called *properties* to which values can be assigned. They also have functions which do things, called *methods*. Methods can be called; properties can be assigned values. Finally, there are *events*. It is these events which make programming for Windows quite different from traditional programming. The traditional program has a main program. Once it is started, it carries on according to its own internal logic until it finishes. In Windows, Windows itself is the main program. As the user moves and clicks the mouse or taps keys on the keyboard, Windows calls what I think of as the dispatcher program of the component or form affected by the event. From the parameters passed to it, this dispatcher program has to figure out what to do to respond to the event. (In Windows lingo, these parameters were called "messages, " a term which has happily disappeared in Builder.) Usually the response is the

default, which is more or less to do nothing. For example, if the mouse passed over a button, it would be most unlikely that the program should respond. But if the button were clicked, a response would be required. In programming for Windows before Builder, writing these dispatcher programs with long **case** statements was a major mechanical but error-prone task. It has virtually disappeared in Builder. Instead, we just have to double-click beside the event in the Object Inspector and are presented with the frame for the C++ code to respond to the event. Builder takes care of all the dirty details of the dispatcher program for us.

In searching Builder help for information about components, it is important to remember to put a T (for template) in front of the name of the component. Thus, if you want information about an Edit component, look for TEdit. You can then find out all about the properties, methods, and events of this component.

In summary then, Builder works with *forms* which have *properties, methods,* and *events*. On the form may be placed a number of *components,* and each of them may have properties, methods, and events. Our task is to write responses to some of these events.

From that starting point, we will first show how to recover for use in Windows the printf() function which is the cornerstone of output from C and C++ programs but does absolutely nothing in a Windows program. We will replace it so that nearly all of your printf() statements will work as they stand, with the output going to a scrolling window, very like a DOS screen, except that it has scroll bars so that the user can scroll up and find material which would have scrolled away in a DOS window. A related task is to provide for DOS-like input, so that the user can give commands to your program. We will do this with a "Combo Box". Then we introduce a List Box to let the user select an item from a list. The item will be the name of a piece of data. We will then have the program graph this data and allow the user to save the graph as a Windows Metafile. Finally, we add a simple Menu with Help as an item and show how to hook up the substantive help files to this menu item. These simple tasks will get us over the items which were the main stumbling blocks for the author. In each case, there was a simple solution, but finding it was not easy.

Restoring printf()

Start Builder. On the opening form, in the Object Inspector, change the Name property from Form1 to MainFm. Set the KeyPreview property to True; we'll come back to explain why. Now drop onto this form a Panel component (the right-most component on the "standard" tab of the component pallet). In the Object Inspector, set the Align property of the panel to alTop and the Caption. This panel can be used later to hold speed buttons. Next, select a Bevel component from the Additional tab of the component pallet; drop it onto the form below the panel. Set its Align property also to alTop, and set its height to 1. The sole purpose of this or any bevel is just the visual effect of drawing a line across the form. Next, drop another panel onto the form below the bevel and set its Align property also to alTop and its height to 24. It should be under the bevel. It will serve to hold the combo box control which will provide for the input of commands from

the keyboard. Now drop a ComboBox component onto this panel, and set its Name property to CmdBox. You may also set its Left property to 2 and its Top property to 0. Clear its Text property. Finally, go to the Win95 tab and select and drop a RichEdit component. (It is near the middle of the Win95 display, and the icon looks like a memo.) Set the Align property to alClient and the ScrollBars property to ssBoth. You may also change the Color property if you don't like looking at a big bright white blob. The component will expand to fill the rest of the form. This space is where the results of the calculations will be written, so let us set its Name property to Results.

Now tap F12 to bring up the code for this main form. Just under the green items at the top put the line:

```
char printout[240], inbuf[240];
```

This printout variable will collect what is to be printed until a '\n' is received or until this buffer is full and must be emptied lest we exceed its 240-byte capacity. The reason for inbuf will be explained later.

Below this line, enter the following code.

```
void printf(char *fmt, ...){
        char buf[120];
        short lenb, lenp;
        /* The following four statements handling printing into buf with a variable argument list.
        See Builder help under va_list for a detailed explanation. See Builder help under vsprintf
        for the example on which this code was based.
         */
        va_list args;
        va_start(args,fmt);
        vsprintf(buf,fmt, args);
        va_end(args);

        lenb = strlen(buf);
         lenp = strlen(printout); // We must initialize printout[0] to \0.
        // Insure that the printout line does not become too long for its buffer
        if (lenb+ lenp >= 238){
                MainFm->Results->Lines->Add(printout);
                printout[0] = '\0';
                 }
        // Tack buf onto anything already in printout.
        strcat(printout,buf);
        lenp = strlen(printout);
        /* If the last character in the printout buffer is a new-line character, it is time to "print" the
        printout buffer and clear it.
        */
```

```
    if(printout[lenp-1] =='\n'){
        printout[lenp-1] = '\0'; //remove the new line to avoid double spacing of output.

        MainFm->Results->SetFocus();
        /* The Lines property of the Results rich text edit component holds the text that
        shows  in this component.  Add is a "method" of the Lines property. This next line
        it what causes the text to appear in the Results window.  It would not actually
        become visible if we had not set the focus to this window.
        */
        MainFm->Results->Lines->Add(printout);
        MainFm->CmdBox->SetFocus();
        printout[0] = '\0';
        }
    }
```

Because this code will be in one of the modules of the program, the linker will not search the libraries for the printf function and this one will be used in place of the standard one. The prototype for this one, however, is the same as for the standard one, so we need only include the standard stdio.h in routines which are to use this substitute routine.

Now that we are ready to handle printf, we need some event that uses it. There are two basic events at the start of the any Builder program for Windows. First, the main form, which we have called MainFm is created and then it is activated. The creation happens once and only once, so it is a good place to initial variables. The activation occurs whenever it becomes the active window. If you noticed carefully in printf, a comment mentioned that printout[0] must be initialized to \0. The creation of MainFm is a good time to do this. So click in the top line of the Object Inspector and select MainFm from the drop-down list. Then click in the Events tab. Then double click in the OnCreate event and put in the line

```
    printout[0] = '\0';
```

We cannot, however, put our test of printf here, because the form on which we are to print has not yet been fully created. So back in the Object Inspector, click on the OnActivate event and enter the line

```
    printf("Hello, World!\n");
```

You are now ready to compile, link and run the program. Click on the green triangle pointing to the right; after a bit of grinding, the screen should appear with the words "Hello, World!" in the upper left corner. Printf has been rescued and with it thousands of lines of precious code!

Before proceeding, you should save your code. Click on the diskette icon. I chose to call Unit1 "science" and the project "sci".

Input from the keyboard

The next task is to find some way to get input from the keyboard. The basic Windows device for this task is the Edit component or the closely related ComboBox. We will use the latter because it enables us to remember previous input and easily repeat it. In fact, we have already put this component on the MainFm and given it the name CmdBox. The way that one almost certainly wants to use the edit component is to type into it, tap 'Enter', and have the program read it. Remarkably, there is no easy way to do just that in Windows! Microsoft has failed to provide the most obviously useful basic functionality. Instead, the user-written program must watch every keystroke in the edit window and , when an 'Enter' comes along, read the text from the edit control, and act on it. To set up the program that does this, get MainFm in the Object Inspector, be sure that the KeyPreview property is True, then click on the Events tab, and double click in the space to the right of the OnKeyPress event. A frame opens into which you need to write a few lines; here is the result with the frame included for reference.

```
void __fastcall TMainFm::FormKeyPress(TObject *Sender, char &Key){
        if(Key == '\r'){
                int Size = CmdBox->GetTextLen()+1;
                CmdBox->GetTextBuf(inbuf,Size);
                Results->Lines->Add(inbuf);
                CmdBox->Items->Insert(0,inbuf);
                if(CmdBox->Items->Count == 30)
                        CmdBox->Items->Delete(29);
                CmdBox->Text = "";
                anything();
                }
        }
```

Thanks to having set the KeyPreview property of MainFm to True, every time the user types a stroke in CmdBox , this function is called. It gets the contents of the CmdBox into the global variable inbuf , writes the line to the Results window, and then inserts it into the list of items in the drop-down list of the ComboBox which we call CmdBox. To be sure that this list does not grow forever and eventually stop the program at a awkward moment, we limit this list to 30 items. Then we clear the text in the CmbBox's edit window and call a function, which we have called anything(), that goes off and does something — anything you like — on the basis of the command which is now in the global inbuf. This anything() is where the real work of the program gets done. The contents of inbuf will be a 0-terminated string.

To keep matters simple, we can write anything()t as just

```
void anything(){
        printf("%s\n”,inbuf);
        if(strcmp(inbuf,"q") == 0){
                exit(0);
                }
```

```
}
```

Remember to provide a prototype for anything(). Then compile and test the program. Every line that you type into the CmdBox edit should, when you press 'Enter', appear twice in the results screen, once from the OnKeyPress routine and once from select(). You can use the q command to quit (exit) the program.

Our interface still leaves one thing to be desired. You may discover that you can resize the form as the program runs, but that the CmdBox edit window does not adjust. Indeed, one finds in many commercial programs that the edit windows are far too small for what they must show and that they do not expand as the form around them is expanded. It is not, however, difficult to make the box expand. Get MainFm into the Object Inspector, click the Events tab, and double click the space to the right of the OnResize event. Write into the frame which appears the line

```
CmdBox->Width = ClientWidth - 2;
```

Recompile and try changing the size of the form.

With these two elements, we have essentially recovered in the Windows environment the command line interface which has served scientific programmers and users well in DOS, Unix, and other environments. We can add on menus, buttons, forms, and so on as appropriate, but we have the fundamental command-response interface that we need for much work.

Graphs and a button

Now let us draw a graph and learn to save and print it. We will activate this graph drawing by clicking a button on the panel we saved for buttons. Since the graph will draw a circle of various sizes, we shall call the button Giotto. (If you don't know the connection between Giotto and drawing circles, look up Giotto in Vasari's *Lives of the Artists*.)

Drawing in C++ Builder is intimately connected with the concepts of a Metafile and a Canvas. Briefly put, a Metafile is a structure which holds commands for making a drawing. It is generally much smaller in size than a bitmap of the same drawing. A Metafile can be saved to the hard disk. A Canvas, on the other hand, is a property of a Metafile, and also of a Form, and some other things. More information can be found in the Builder help file by looking for TMetafile and TMetafileCanvas. The basic steps in drawing are:

1. Create a Metafile with something like
   ```
   TMetafile *mmf;  //mmf is just short for My Metafile
   mmf = new TMetafile;
   ```
   (For the moment, do not worry about exactly *where* these statements are put; that will become clear later. What matters here is the order and the action performed by each.)

2. Create a Canvas for this Metafile; note that the Canvas "knows" what Metafile it belongs to.
```
TMetafileCanvas *mc;
mc = new TMetafileCanvas(mmf,0);
```

3. Draw on the canvas with commands like;
```
mc->MoveTo(10,20);   /*Position the pen 10 pixels to the right and 20 pixels
                              down from the upper left corner of the canvas. */
mc->LineTo(100,200); // Draw from where the pen is to the point (100,200).
```

4. *Delete the canvas!* This sounds crazy, but the destructor must in some way attach the data in the canvas to the Metafile. It is useless try to display the graph in the metafile until the canvas has been deleted.
```
delete mc;
```

5. Close and then Show again the form on which the graph is to be displayed. This process erases whatever was previously on the form. If it is going to be displayed on a form called GiottoFm, as in the example below, then the commands are:
```
GiottoFm->Close();
GiottoFm->Show();
```

6. "Stretch" the Metafile onto the Canvas of the form.
```
GiottoFm->Canvas->StretchDraw(ClientRect,mmf);
```
This last step is best put in the response to the FormPaint event in the form where the graph is to be drawn. Then, whenever Windows detects that the form must be repainted, the graph is redrawn. One should also put in the response to the Resize event of that form the command
```
GiottoFm->Invalidate();
```

Now let us put all this to work in a simple example. Along with the declaration of printout and inbuf near the top of science.cpp, put the line
```
TMetafile *mmf;
```
and in response to the OnCreate event for MainFm, put
```
mmf = 0;
```

Next, drop a button onto the panel above the command box. Then, in the Object Inspector, set its Caption property to Giotto, its name to GiottoBtn, and give it an appropriate size and position on the panel so that it looks nice. Then click the Events tab and double click in the space to the right of the OnClick event. Fill the framework provided so that the whole looks like the following.

```
void __fastcall TMain::GiottoBtnClick(TObject *Sender)
{
        TMetafileCanvas* mc;
```

```
if(mmf != 0') delete mmf;
mmf = new TMetafile;
mc = new TMetafileCanvas(mmf,0);
// Draftsman does the actual drawing on the Canvas
draftsman(mc);
delete mc;
GiottoFm->Close();
GiottoFm->Show();
}
```

Follow these lines with the draftsman() routine below. To understand it, you need to know that a point on the "Canvas" is described by two coordinates, (x,y) with x measured from the left as usual and y measured *down from the top* of the canvas. These coordinates are measured in pixels (picture elements), the little dots of which the computer screen is made. The program below assumes that the screen size is 800 by 600, a common sized screen at present. A Canvas has a Pen with which it draws lines or outlines of shapes and a Brush with which it paints the interior of closed shapes, like the ellipse in draftsman(). Pens have a Style (like psDot, psDash, psSolid) , Width (in pixels), and Color; Brushes have Color and Style (which has to do with hatching). See Tbrush and Tpen in the Builder help for full details.

```
void draftsman(TMetafileCanvas* mc){
    static int i = 10;
    int height, width;
    i+=5;
    height = 600;
    width =  800;
    mc->Pen->Style = psDot;
    mc->Pen->Width = 1;
    // Draw a box
     mc->Rectangle(5,5,width-5,height-5);
    // Draw its diagonals
    mc->MoveTo(5,5);
    mc->LineTo(width-5,height-5);
    mc->MoveTo(5,height-5);
    mc->LineTo(width-5,5);

    mc->Brush->Color = clRed;
    // Draw an ellipse which fits in the rectangle with top left corner at (i,i) and
    // bottom right at (i+100, i+100)/
    mc->Ellipse(i,i,i+100,i+100);
    }
```

Be sure to provide a prototype for draftsman near the top of the file.

Now we have to provide a form on which our draftsman can display his skill. Click File in the C++ Builder main menu, then click New Form, and a new form appears. Set its name to GiottoFm and its caption to something like "To Benedict from Giotto". Make its OnResize response just

```
GiottoFm->Invalidate();
```

and its OnPaint response

```
GiottoFm->Canvas->StretchDraw(ClientRect,mmf);
```

Now when you got the new form, you also got a new .cpp file to go with it. These last two additions have been in this .cpp file. If you now try to compile, you will find that you cannot because mmf has not been declared in this file. It was declared in the other one, so what we need to do is to inform the compiler that mmf is a pointer to a TMetafile structure defined in some other unit. So at the of the file, just below the green lines, put in the line

```
extern TMetafile *mmf;
```

You will find that every time you make a new form you get a new .cpp file associated with it. For some reason, builder won't let you give them exactly the same name except for the extension, so I use the convention that if I call the .cpp file X.cpp, then I call the form Xfm . If you now click the save button, you can give the new unit the name Giotto.

Try again to compile. You will hit a snag in compiling science.cpp because it refers to GiottoFm but GiottoFm has not been declared. Click File and far down the list you will find "Include unit header". Click that and select Giotto as the unit whose header you want to include. You should now be able to compile and run the program. When you click the Giotto button, you should get a red circle on a white background. If the circle looks a bit elliptical, adjust the right side of the form until its proportions are 4 across to 3 down. Then Giotto will draw perfect circles which move down with successive clicks of the button.

ANSI Strings

Wherever Builder needs a string of characters, it uses a class called ANSIString. The Text of an EditBox is an ANSIString; every line in ComboBox is an ANSIString; every line of a Memo or Rich text edit component is an ANSIString. This class does more than the traditional zero-terminated C string. In particular, if b and c are both ANSI strings, then `b = = c` is true if b and c are the same and `b + c` is the concatenation of b and c. It is not difficult to make an ANSIString. For example, if b has been declared to be an ANSIString, then any one of these statements will provide it with content:

```
b = 2.5;
b = 7;
b = "This right side is just a zero-terminated C string."
```

ANSIStrings cannot, however, be used in place of the traditional C string in functions such as fopen() or strcmp(). Fortunately, there is an easy way to extract the C string, namely with the function c_str() which is a member of the ANSIString class. For example, to get the "contents"

of the ANSIString b into the character array “filename”, we just do

```
strcpy(filename, b.c_str());
```

Help Files

One of the nicest features of a good Windows program is the help facility. It is, however, extremely hard to find help about how to write help files and how to connect them to your program. Here are the steps, as ascertained from scraps of information here and there and considerable trial and error. Basically, you write the contents of your help files as you would any other document. Put in subtitles for each topic as you go along. Then go back and put some very special footnotes in front of each subtitle and put a page break at the end of each topic. Save this file in the rich text format. Then use the help compiler to create a help project, a help contents file, and ultimately, a compiled help file which will have the extension .hlp. Finally, connect this file with your program. Here are the steps in more detail.

1. Write the text of your help file in a word processor such as WordPerfect. You can use italics and bold type, and various fonts and sizes. I would not recommend using the equation editor. Divide the text into topics and provide a title line for each topic. Put a hard page (sometimes called a page break) at the end of each topic. (In WordPerfect, Ctrl-Enter puts in a hard page.) It is perfectly all right to expect the user to read the topics in order, for the browse buttons (marked with << and >>) allow one to do so. It is not necessary to assume, as is often done, that the user suffers from attention deficit disorder and can follow nothing longer than half a page. Numbering the topics can be useful in deciphering some of the error messages you may get from the help compiler.

2. Go to the top of the document and set the footnote options to use characters to represent the footnotes, and namely in this order: # $ K + . Set it to repeat on each page. (In WordPerfect, you do this by select Insert | Footnote | Options. In the panel called Numbering method pick Characters, give these four characters without space between them and check the box for “Repeat on each page.”)

3. In front of the text of each topic insert four footnotes as follows:

    # The identifier you will use for the topic. It should begin with IDH_ and should have no blanks in it. Examples:
        IDH_WhatIsNew
        IDH_GettingStarted

    $ The title of the topic. This is what shows up in the lower window when searching for a keyword. It should be simple, descriptive English. Examples:
        What is new in this release

K Key words or phrases which will appear in the index generated by the help compiler. The individual items should be separated by semicolons. Examples:

Distributed lags; Soft constraints; con command; polynomials

\+ Name of the section of the help file. You might have, for example one section called Tutorial and one called Reference. If the user is reading a topic in the Tutorial section, the browse buttons will display earlier or later topics in this section but will not move over into the Reference section. Leave no space at the beginning of this footnote.

You may also add *hot spots*, text which will appear underlined in green where a click will cause a jump to another help topic. To add a hot spot, type the topic identifier for the jump immediately (without a space) after the hot-spot text. Then double underline the hot-spot text. In WordPerfecrt, the double underlining is done by selecting the text to be double underlined, choosing Format | Font and checking the Double underline box in the Appearance panel. Example:

You may also use soft constraintsIDH_SoftCon to solve this problem.

The user will not see the IDH_SoftCon, but will jump to that topic if hey clicks the underlined text.

4. Save this file as a *rich text* file, which will automatically have the extension .rtf . I recommend strongly that you also save it in whatever format is native to your word processor, .wpd for WordPerfect, .doc for Word. In principle, you can read in the .rtf file when you need to make corrections to the text, but I found that in practice additional control characters then appeared which caused trouble. So save it in both its native format and rtf format. Doing so requires that you be alert in the use of Save as ... .

5. Start the Microsoft Help Compiler Workshop. This product is included with C++ Builder. Its default installation is as

C:\Program Files\Borland\CBuilder\Help\Tools\HCW.EXE

It is probably worth your while to create a shortcut for it on the desktop. Select Help from the main menu and Training cards. You will be taken through the steps for creating a help project file. Remember that all references to a Help file refer to the final .hlp file which is the end product of your efforts. If your help topics make better sense in the original order than in just random order, then you will want to add “browse buttons” to your help file, so that the reader can easily read them in the order in which you wrote them. The browse buttons are buttons on the tool bar that have << and >> on them. To get browse buttons on your help windows, you need to have the project (.hpj) file open. Then click on the “Windows” button in the stack of buttons on the right. Then click on the “Buttons” tab. In the “Window Type” drop-down list, select the window type.

(Probably you just want the “main” type.) Then in the panel labeled “Buttons” you will see a number of check boxes. Make sure the “Browse” box is checked.

6. Make the Contents file. Still in the Help Workshop, click on Help | Help topics | Step-by-step procedures | Create contents file. Follow the directions for creating the help contents file. (The help contents file is new in Windows 95; anything you find about contents topics for earlier versions of Windows should be disregarded. Unfortunately, much of the on-line documentation that comes with Windows 95 and Builder still refers to the old methods. To see how the contents file is supposed to work, play with the Help topics item of the Help menu of the Help Workshop.) The result should be a file with the same root name as the .hlp file but with the extension .cnt. Both the .hlp and the .cnt file go with the .exe file to any user of your program. Making the contents file is laborious. You have to type in all those topic ID’s and subtitles from your file with the substance of the help. I kept thinking that there must be an easier, automatic way; but as far as I can see, there is not. You can make some use of the clipboard to reduce the retyping.

7. Compiling the help file. When the contents file is all ready, select File | Compile ... . Check the box for automatic running of the help system after compilation. If all goes well (fat chance!), you can test out the working of your help system. You do not need to be in your application to run your help system. If you have called your help file science.hlp and it is in the directory \sci, the from the desktop, you can click the Start button in the bottom left corner, click Run, and give the line

```
\windows\winhelp    \sci\science.hlp
```

Your help system should function here exactly as it ultimately will from your application.

8. Connect the help system to your application. Let us say that you have given your application a modest name like “science” and have called its help file, created by the Compile operation in step 6, “science.hlp”. Further let us assume that the user of your program is to but this file and the “science.cnt” in the same directory with “science.exe”. Then in the response to the OnCreate event of MainFm put this line:

```
Application->HelpFile = ExtractFilePath(ParamStr(0))+"science.hlp";
```

The first thing that this command does is to figure out where “science.hlp” is on the basis of the directory in which science.exe was found. Recall that an old-fashioned C program, a “console application” if you will, starts with the line

```
int main(int argc, char **argv){
```

where argc is the argument count — the number of arguments on the command line — and argv is the list of those arguments. By convention, the first of these is the full path name of the .exe which is being executed. In Windows, programs are not normally started by command lines, but they can be and the facility for using the command line is preserved by a global variable called ParamStr, which is exactly a list of the arguments (as ANSI Strings), including the first, the full pathname of the program being run. The Builder function ExtractFilePath() chops off the file’s individual name (“science.exe” in our case) and leaves just the path, as an ANSI String. We just add to the end of it “science.hlp” and

have exactly what we need to assign to the application's HelpFile property.

All of the above gets the program ready to use the help file but does not actually show it. For that you need a menu with a Help item on it. If you have not already done so, drop a Main Menu component onto your main form. It does not matter much where you put it. You can drop in the middle of the Results memo component. You will at once see the menu up above where it belongs and an icon for the menu on the Results memo. When you run the program, you will see on the menu above and not the icon. The icon, however, is the programmer's avenue of access to the menu. Right click on it to bring up a box from which you can choose Menu Designer. (*Teach Yourself Builder* is good on the Menu Designer and I won't repeat what is there.) Insure that you have a main menu item Help and that under it there is an item for "Help topics" and perhaps one for "Tutorial" and one for "Reference". When the menu is ready, get out of the menu designer and click Help on the menu and then on "Help topics". You will get the frame for the program for responding to the user's click on this item. Into this frame, put

```
Application->HelpCommand(HELP_FINDER, (DWORD)0);
```

That will bring up the top level of your help system with one or more closed books. Clicking on them causes them to open and reveal their table of contents. Clicking on a topic then takes you to that topic in the help file.

You may want more specific access from a Help menu item. For example, if you have a topic with the identifier IDH_StartTutorial and a "Tutorial" item on the Help drop-down menu, then you would click this item on the menu and a program frame would open into which you would write

```
Application->HelpJump("IDH_StartTutorial");
```

For your trouble in doing all this, your users will find that Help works the way they expect it to work, that they can look up things in the index (made from your K footnotes) or by using Find, they can find all occurrences of any word you have used.

Common Dialog Boxes and their Options Property

Many programs need to allow the user to open or save files, to print or setup the printer, to specify text to be found and possibly replaced, or to select fonts and colors. Microsoft has supplied dialog boxes for these purposes -- called "common dialogs" -- and Borland has put a C++ "wrapper" around them to make them easy to use. Using them for their intended purposes both saves you time and makes your program work like others with which the user is familiar. In Builder, you click on the "Dialogs" tag in the tool bar and then "drop" them onto the form where you want to use them. While programming, you will see their icons on your form, but when the program is running, you will see a dialog box only when its Execute() method is called. Mostly,

you just call their Execute() method and then use "properties" of the dialog when it returns. The name of the file the user wants to open, for example, will be in the FileName property of the Open dialog when it returns. The Find dialog and the Find and Replace dialog, however, have "events" to which you must respond if the dialog is to have any meaning. In the Find dialog, there is a button labeled "Find Next." When that button is clicked, the OnFind event occurs, and your program should respond to it. The dialog will stay open until you close it with its Close() method or the user closes it.

The only real complexity in using these common dialogs that I have encountered is their Options property. If you read all the help files about all the dialogs with Options properties, you will learn that you should set the options to "customize the appearance or functionality of the dialog." You may also need to "read" the Options property to find which check boxes or radio buttons the user selected, if appropriate. How you "set" or "read" the Options property, however, is nowhere revealed. We can learn, however, that the Options property is a Set. But what is a Set?

A Set turns out to be a special Borland construct for Builder. As its name implies, it is a set of something. In the case of our Options property, it is just a set of integers. Each option is simply a particular integer. If the integer is in the set, then the dialog has the property for which that integer stands. For example, in the Find Dialog, if 0 is in the Options set, the Down radio button is selected and the user wants to search down. If 0 is not in the set, the user presumably wants to search up. If 5 is in the set, the Match Case box is checked; if it is not, there is no check in this box. In order to make it easier to remember the meanings of the different integers in the Options property of different dialogs, Borland has supplied mnemonic names for them through enumerations. Thus instead of asking whether the Options set for a Find dialog contains 0, we can ask whether it contains frDown; instead of asking if it contains 5, we can ask if it contains frMatchCase. The meanings of these mnemonic names is relatively well documented in the help file for each dialog type. (Remember, if you want to find out about, say, the Find dialog, you look for TFindDialog in the help index.)

So far, so good; but how do you find out whether a particular integer is in the Options set or how do you put one into it? A Set turns out to have a method called Contains. To find out if the Options set of the xxxFindDlg contains the integer frDown, you could just do

```
bool down;
down =  xxxFindDlg->Options.Contains(frDown);
```

If down emerges with the value TRUE, then the frDown option was in the set, and you had better search down the list. If down emerged FALSE, then you should search up.

How to you put integers into the Options set or get them out? For example, the FindDialog has a check box to let the user specify whole word searching only. If you have no intention of offering that option to your users, you can spare them the frustration of checking the box and finding that the check has no effect by removing the box. To do so in the above example, you would do

```
xxxFindDlg->Options << frHideWholeWord;
xxxFindDlg->Options->Execute();
```

The dialog will appear with no Whole word check box. You can also remove all elements of the Set with the Clear() method or remove individual elements with the >> operator. Sets also support union, intersection, and difference operations. The on-line help for Set has an example that is helpful if you need to make further use of Set.

Grids for Displaying Matrices

Scientific computing is quite likely to involve displaying data in rectangular arrays resembling matrices or spreadsheets. Data can be shown in this way rather easily in Builder by using a Draw Grid, but the documentation in the help menu is hardly adequate. Here are the essential steps.

First, you use File | New Form to get a form on which to show the grid. Then from the "Additional" tab of Builder components select a DrawGrid and drop it onto the form. (It might seem that the StringGrid would be simpler; but my understanding is that it is more complicated and slower than the DrawGrid.) From the help file you can learn how to set its size and various other properties, including the number of "fixed" rows and columns, that is, rows and columns at the top and left which do not scroll out of sight. The crucial question, however, is how to get the content into the grid. The answer is that you must respond to the OnDrawCell event; so, with the Object Inspector pointing to the DrawGrid, click the "Events" tab and then double click in the space to the right of OnDrawCell.

The automatically generated code will look something like this:

```
void __fastcall TForm1::DrawGrid1DrawCell(TObject *Sender, long Col, long Row,
        TRect &Rect, TGridDrawState State){

        }
```

Our job is write the code that goes between the braces. Note that we have the column and row numbers from which to figure out what text is to be written or drawn into the cell. Let us suppose that we have written a routine, which we may call GetCellText(long Col, long Row), which writes into an ordinary zero-terminated C string the text which we want to display in the cell specified by Col and Row. Then we can fill in the DrawGrid1DrawCell program somewhat as follows.

```
AnsiString cell = AnsiString(GetCellText(Col, Row)).Trim();
int y = Rect.Top ;
int x = Rect.Left;
ShowGrd->Canvas->TextRect(Rect, x, y, cell);
```

This code will pass the column and row numbers to the GetCellText routine which we must write. It will return a pointer to a string containing the text we want to go into the cell. The code shown puts that text in the upper left corner of the cell. If that is not satisfactory, x and y can be modified to center the text or right justify it.