Towards a Cross-Platform Version of G7

Clopper Almon

20th Inforum World Conference

Florence, Italy, September 2012

Contents

Motivation

The G7 model-building program lies at the heart of Inforum work around the world. Keeping it alive and well is therefore an important task. Currently, it is built using Borland C++ Builder 6, (BB6). Back in the early 1960's, I wrote a Fortran program called LS (for Least Squares) which did basic regression and transformation of variables. From two time series, it could create a third by addition, subtraction, multiplication, or division. The earliest versions of G go back to the early 1980s, were written in De Smet C, and introduced the f command for variable transformation, but did not use a graphical interface, though they drew beautiful, full-color, full-screen graphs. After Windows 95 appeared in 1996, my students became increasingly resistant to a nice old DOS program, and I tried various tools for writing a Graphical User Interface. In 1997, Borland C++ Builder appeared, and with it I accomplished more in a day or two than I had with previous tools in months. And the price was, as I recall, under $200. It took Microsoft several years to offer anything comparable, and Builder got better and better up to version 6 in 2002. But eventually, Microsoft offered a somewhat comparable product, and after some gyrations Borland disappeared. The product is now owned by Embarcadero Software, which will sell an upgrade for $2,299.

BB6 worked fine under Windows XP, was not easily persuaded to work under Windows Vista, and now works again under Windows 7. But who knows when Microsoft will again make some change that again disables this venerable, 10-year-old program. And if that happens, programs built with BB6 – such as G7 – may also not work on new machines.

These concerns fueled my already keen desire to get a working version of G7 built with tools that are kept up-to-date and that work on all three major platforms, Linux, Mac, and Windows.

Back in about 2008, I had a laptop that had become so slow under Windows that it was practically useless. I installed Ubuntu Linux with a dual boot; and, presto, I had a lively new computer. I soon learned to use the free software which came with Ubuntu or was easily installed from its website. It included a word processor distinctly better than Word, a spreadsheet equal to Excel for my purposes, slide show software comparable to Powerpoint, a good image processor, a music writing editor, a website builder, and a movie maker, and, of course, a C++ compiler. All free. And no viruses, because Linux was built from the bottom up to defeat would-be virus writers. In short, it has everything I want except G7.

Before Borland vanished, it made an attempt to port Builder to Linux. The product was called Kylix, and we bought a copy. It worked fine on a couple of Linux distributions now long out of date, but does not work on any current versions.

My first decision, therefore, was what system to use for writing a new, cross-platform version of G7, one which would initially be for Linux but would use tools available for Windows and Mac. The choice fairly quickly narrowed down to (1) Qt or (2) wxWidgets with either CodeLite or Code::Blocks as the Interactive Development Environment (IDE).

Qt was originally a proprietary product of a Norwegian company called TrollTech, created by the originators of Qt. TrollTech was later bought by Nokia, which intended to use Symbian, created with Qt, as an operating system on its smart phones. In February 2012, Nokia gave up Symbian in favor of the Microsoft system, and the Qt project became an independent, open-source software organization. It has announced the release of Qt 5 for August 2012. By contrast, wxWidgets, Code::Blocks and CodeLite, have always been open-source projects without user fees. Both Qt and wxWidgets have an impressive list of applications created with them.

I consulted with Frank Hohmann and Douglas Meade. Frank strongly recommended Qt but without much recent experience with it; Doug had already made considerable headway using Qt 3, which is rather different from the current version, Qt 4. So I started with Qt 4 in about February of 2012.

I soon learned that there was a new tool called Qt Creator, an interactive development environment (IDE) which for the first time made Qt comparable to Builder in ease of use. There was, however, only one tutorial on the use of Qt Creator. It did not start up in the way described, but once I got it started, I made some progress. But my project built according to the tutorial did not work. I repeated the tutorial four times. One time, the third try, the project worked as advertized; the other three times I got programs that did not fully work.

Discouraged but not defeated, I set to work to try to work through the official book on Qt. It was written prior to the release of Qt Creator, and a number of things were different with Creator. As I got deeper into Qt, however, I saw that I would have to use it every day to acquire and maintain fluency with it. One of the great features of Builder had been not only good tutorials but the fact that use was so easy and natural that I could work with it one day, put it down for several, and then pick up where I had left off. I did not have to carry a lot of argument lists for function calls around in my head, as had been the case when writing directly to Windows. Using Qt was going to be much like writing directly to Windows with the necessity to look up details frequently. Finally, I became so discouraged that I decided to put Qt on hold and try wxWidgets with Code::Blocks as an IDE. Perhaps the appearance of Qt 5 with Qt Creator playing a central role will generate some new documentation which will make it more a more friendly and dependable tool.

wxWidgets is a collection of visual objects – frames, dialogs, buttons, labels, check boxes, radio buttons, text boxes, drop-down boxes, combo boxes, grids, multi-line editors – together with sizers, a string class and routines for drawing and saving graphs. I was soon delighted to find that part of the Code::Blocks IDE was something called wxSmith that combined with wxWidgets to give a tool very like Builder. Better still, on the Code::Blocks web site there was a set of tutorials by Bartlomiej Swiecki, the creator of wxSmith.[1] They were clear and very helpful. But they stopped short of two techniques necessary for writing G: (1) drawing and saving graphs, and (2) getting commands from the keyboard (via the white command box in G) and displaying numerical results on the scrolling (blue) screen using the C function `printf()`. With the help of the wxWidgets book[2] and friendly correspondents on the Code::Blocks forums, I was able to work out these techniques.

There were a few problems in the tutorials, and Swiecki seems to have turned his attention elsewhere, so I was encouraged by the Code::Blocks authorities to use the wiki capability to edit them. I did so rather extensively, updating Tutorial 1 on getting started, adding the section in Tutorial 2 on creating the main menu, and writing Tutorials 8 (on graphs) and 9 (on restoring `printf()`) from scratch. I changed some terms to conform to the usage in the wxWidgets book, and I changed the accent in the English from youthful Polish to elderly American. I find it nice to work with the tutorial in hard copy so I have included most of them in slightly edited form in the appendix to this paper. I have omitted Tutorials 5 and 7. I do not really understand Tutorial 5, and it is not necessary for the following ones. Tutorial 7 is just a fragment barely begun by Swiecki and is not necessary for my Tutorials 8 and 9.

With these techniques in hand, I felt I would be able to handle the problems of getting a very basic version of G7 running on Linux. Namely, I want to be able to assign a data bank, type to the screen the values of a variable, create new variables in the workspace bank with the *f* command, run a regression and show the results on the screen. A second level would be to introduce data into the workspace and to draw and save graphs. A third level would be the use of *add* commands with arguments. And from there gradually nearly all capabilities could be added. The Excel commands are, of course an exception. Excel is not available for Linux, but Calc, a very capable spreadsheet is available for all three platforms. Rewriting all of the xl commands to work with Calc involves learning the Calc application programmer's interface (API), a task I am not likely to take on. It is perhaps worth noting that Build, idBuild, Interdyme, and various utilities are all command line routines in straight C++ without any GUI programming. They should all be fairly easily converted to run under Linux.

As I began to study the current code of G7, however, I realized that I had another task.

1 See http://wiki.codeblocks.org/index.php?title=WxSmith_tutorials or Google "wxSmith Tutorials"
2 *Cross-Platform GUI Programming with wxWidgets*, by Julian Smart and Kevin Hock, (Pearson Education, 2006)

Since I converted the code for the DOS-based G6 to Windows-based G7, now nearly 15 years ago, the code has been worked on by several hands. In particular Frank Hohmann, Douglas Meade and Ronald Horst have introduced techniques to make the program more reliable and versatile. I feel complimented that my code was comprehensible to them. Now, however, the code is full of older code that has been turned into comments and replaced by newer code, sometimes accompanied by explanations which are sometimes comprehensible. I feel it is very important to make the structure of the program plain and to keep the code readable. On looking into the code, I tried to put myself in the position of a programmer who knows his C++ and is asked to fix something or add something to G7. I realized that it would be a rather terrifying position to be in.

I resolved to try to create the port of G7 to Linux as a series of tutorials, explaining as well as possible the code as it moves from a very simple G to a nearly complete one. In some cases, especially near the beginning, it appears to me that the code can be significantly simplified. Certainly, some of the accretion of old code turned to comments can be eliminated and the current coding and its rationale explained. I will, however, try to use the same variable names wherever possible. For lack of a better name, I have called emerging program Gwx.

Gwx Tutorial 1: The Basic Framework

Getting the Tools: C++ and Code::Blocks in Ubuntu Linux

I will assume that you already have Ubuntu or similar Linux installed on your computer. You do not need to give up Windows; you can install it so that, on start up, you are asked which operating system you want to use. If you choose Linux, you can access all your Windows files but you can't run Windows programs. The next step for anyone wishing to work through the tutorials is to get the GNU GPP C++ compiler and related material installed. This step is pretty easy. Click "Applications" in bar across the top of the screen; then pick Ubuntu Software Center and install **build-essential.** The next step is to get the Code::Blocks IDE version 10.05 or later installed. (Code::Blocks uses the Ubuntu release numbering scheme. The 10.05 means the release of May 2010. This is currently (2012 August) the standard "stable" release.) Perhaps it can now also be installed from the Ubuntu repositories just as was build-essential. Look to be sure you are getting 10. 05 or later. If not, then edit the `sources.list` file in the `/etc/apt` directory to add the line

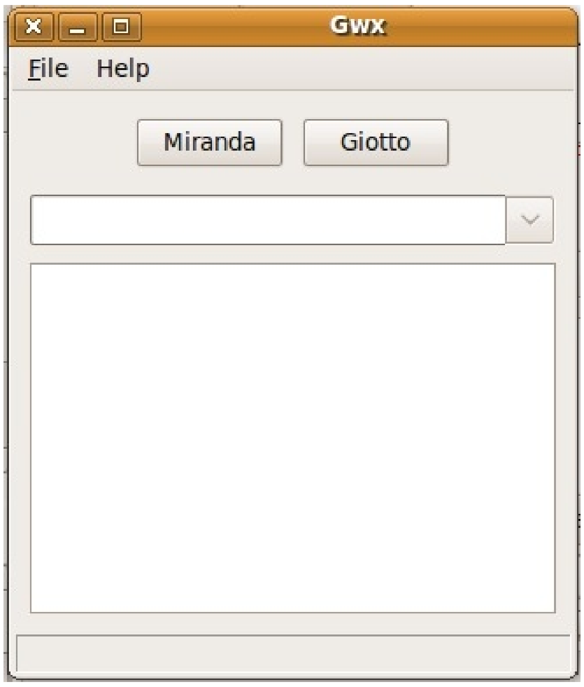
```
    deb http://apt.jenslody.de/stable stable release
```

This file was protected from writing, so I first had to use a terminal to do

```
    sudo chmod a+rw sources.list
```

and then use gedit to add the line.

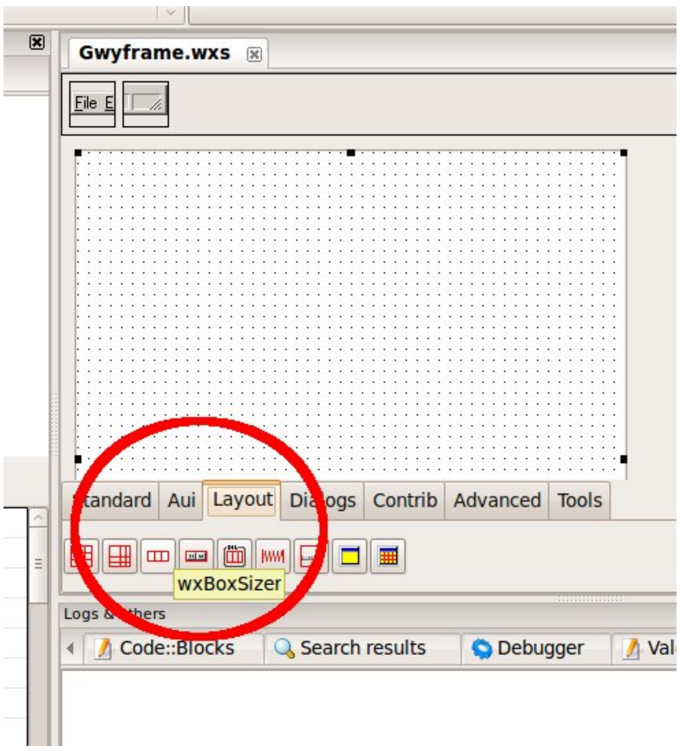
The Gwx Framework



The basic visible framework of G7 has five elements: (1) a window with a title in the title bar at the top (2) a main menu (3) a row of buttons, (4) a combo box for giving commands and (5) a scrolling window to show the results. Such a framework is shown to right. There are two buttons in a horizontal row with the rather unusual labels of "Miranda" and "Giotto". Below them is the combo box for giving commands, and below that is the large rectangle for showing the results. We will describe here very succinctly how it was created. A screen video recording of the construction is available . I am going to assume that you have either watched it or read at least the first tutorial, which provides the basic terms for referring to the Code::Blocks IDE. If the exposition proves cryptic, please work through the tutorials where everything is spelled out at greater length.

Start up Code::Blocks, and tell it to create a new project. When asked for the project type, choose the wxWidgets project icon. When asked for the project name, say Gwx. Otherwise, choose the defaults.

In the form's property browser in the lower left quadrant of the window, give it the Title property “Gwx” and put a check in the “Centered” box. The next thing we need to do is to put a box sizer on the form. Click the Layout tab – as shown in the circled area in the picture to the right – and then run the cursor over the various icons to find the box sizer – it is the third from the left – and click on it. Then click anywhere in the field of dots. A little square appears in the upper left corner of that field. Into it, put in a similar way a panel, which is on the Standard tab. Check the panel's Expand property. Onto the panel, put a box sizer and in its properties make it vertical. Into it drop another box sizer and leave it horizontal. It will handle the buttons. Change its Proportion property to 0; the buttons do not need more room when the user changes the size of the window.



Now we add two buttons into this sizer. On the Standard tab, click the button icon (the first on the left) and then click in the box sizer. Change the Proportion property of each button to 0. Change the Label property of the left button to “Miranda” and its Var name property to MirandaBtn. Change the Label property of the right button to “Giotto” and its Var name property to GiottoBtn.

Now we want to add a ComboBox for the commands to G. On the Standard tab, use the tool tips to find the ComboBox icon. Click it and then click in the form but a tiny bit below the the red line surrounding the two buttons. Mark the Expand property and the Focused property of the ComboBox and make its Proportion property 0. Change its name – the Var name property – to CmdBox (for Command Box).

Finally, add a TextCtrl to show results of calculations. On the Standard tab, click on the TextCtrl and then in the right half of the CmdBox. A text control appears below the CmdBox. In its property browser, remove the word “text” from its Text property; give it the “Var name” property of Results, uncheck Default size, and make it 300 wide and 200 high. Mark its Expand property, and drop down the Style properties and check AutoScroll, Multiline, and Full_Repaint_On_Resize.

Now find the Code::Blocks “Build and Run” icon – it looks like a right-pointing green triangle in front of a cog wheel – and click it. Your program should compile and run, showing something like the picture above. Try stretching it diagonally. The CmdBox should expand horizontally; the Results area should expand both vertically and horizontally. Use the x in the title bar to close it. If it refuses to close or doesn't have a main menu, consult Tutorial 1 for remedies.

If all has worked well, this is a good time to save your work. Click File | Save everything. That will save everything to your Gwx directory. I warmly suggest that, every time you get a working version of the project, you copy all of its files to a backup directory.

I call the backup directory GwxBack and use a Linux terminal to do the copy with the command

```
      cp -r  * ../GwxBack
```

Veteran DOS users should note that in Linux the * means “everything”. Had we put *.*, only files with names containing a dot would be copied, and that would be most but not quite all files. I learned this the hard way. When I needed my backup, it didn't work, and I had to start all over.

Generally, Code::Blocks is pretty stable, but I did have one total crash which messed up the files in in Gwx. I had to start over, and ever since I have been good about backing up to GwxBack. I also suggest that, at the end of Tutorial 1, you back up Gwx to a directory named Gwx1; and at the end of Tutorial 2, to Gwx2; and so on. Thus, if you really mess up – as I have a number of times – and need to start over, you won't have to start all over at the beginning.

Restoring printf()

G has myriad calls to the function printf() for writing to the Results screen. Now printf() is the time-honored way to write to the screen in console applications, but in a GUI program, output of printf() simply gets lost. In G7 for Windows, we wrote a version of printf() which wrote to the Results window. It somehow got in front of the standard printf(). In Linux, I seemed to encounter some confusion between the two. I did not really get to the bottom of this problem but instead called the new routine for writing to the Results window printg(). The format for using it is exactly the same as for printf(), so to change a long printf() call to a printg() call requires only changing one letter, namely the f to a g. So how do we write printg()?

Open the file GwxMain.h (Use File | Open … to do so.) and add printg to its public members as shown here.

```
    public:
         GwxFrame(wxWindow*parent,wxWindowID id = -1);
        virtual ~GwxFrame();
        void printg(char *fmt, ...);

    private:
```

(In Tutorial 5, we will make printg() a friend rather than a member of this class, but for the moment work will be simpler with it as a member.) Save and close that file. Double click on the “Miranda” button. The file GwxMain.cpp opens and you find the frame for OnMirandaBtnClick and fill it in as follows:

```
void GwxFrame::OnMirandaBtn(wxCommandEvent& event){
      printg("%s", "O brave new world!\n");
      }
```

You recognize the call to printg() as just like a printf() call. You should also recognize Miranda's famous ironic line from *The Tempest*, Act 5, Scene 1.

Just below it, you can now put this definition of some constants and the character arrays printout, inbufAsRead, and inbuf and the function printg(), followed by the code for printg():

```
#define MAXPRINTOUT 500
#define MAXLINE 500
#define OK 1
#define ERR -1

char printout[MAXPRINTOUT], inbufAsRead[MAXLINE], inbuf[MAXLINE], *pinbuf;

void GwxFrame::printg(char *fmt, ...){
    char buf[MAXLINE];
    va_list args;
    va_start(args, fmt);
    vsprintf(buf, fmt, args);
    va_end(args);

    short lenb, lenp;
    lenb = strlen(buf);
    lenp = strlen(printout);
    if(lenb+lenp >= MAXLINE-2){
        wxString wxPO(printout, wxConvUTF8);
        Results->AppendText(wxPO);
        printout[0] = '\0';
        }
    strcat(printout, buf);
    lenp = strlen(printout);
    // If the last character in printout is \n, then it is time to
    // print
    if(printout[lenp-1] == '\n'){
        // convert printout to a wxString, which we will call wxPO
        wxString wxPO(printout, wxConvUTF8);
        Results->AppendText(wxPO);
        printout[0] = '\0';
        }
     }
```

The key to this rather unusual piece of code is the "variable argument list", abbreviated to "va" in the code. It is the use of this technique which enables printf() and our printg() to take a number of arguments unknown to the writer of the code. The unknown number of arguments is represented by the three dots . . . ; there are no spaces between the dots, and they must be preceded by at least one normal, non-optional argument. The "vsprintf(buf,fmt,args)" function is the variable-argument version of "sprintf()"; it writes the arguments passed by the calling program into "buf" using the format supplied by the calling program. The net result of all this is that the writer of the calling program writes a call to "printg()" exactly as he would write a call to C's standard "printf()"; and at the end of the four lines beginning with the letter "v", there will be in "buf" exactly the normal zero-terminated C string which "printf()" would have produced.

Now the task is to get that string displayed in the wxTextCtrl. There may be several calls to "printg()" before a new-line (\n) character is encountered. With each call, we add the contents of "buf" to "printout" until either (a) an end of line (\n) is encountered or (b) "printout" won't hold "buf" plus what is already in printout. In either case, it is time to display "printout" in the wxTextCtrl and clear "printout". (We have made both "buf" and "printout" pretty big and are not going to worry about their not being big enough to handle a call to printg().)

Now we encounter a new hurdle. Our "printout" contains a normal C-string, but to display it in the wxTextCtrl, we have to convert it to a wxString. Fortunately, wxWidgets gives us a one-line solution to that problem:

```
  wxString POwx(printout, wxConvUTF8);
```

This line declares POwx as a variable of type wxString and initializes it by converting our C-string, printout, to the UTF-8 representation of the string and stores that string in a variable of the type wxString. ("POwx" is just a name I made up to suggest printout transformed to a wxString. You could replace it with "abcd" or whatever. UTF-8 stands for Universal character set Transformation Format, 8 bit. It is a variable-width encoding that can represent every Unicode character.)

Now that we have our wxString, we can append it to the Results member of the TextCtrl by the line

```
      Results->AppendText(POwx);
```

Finally, we set "printout"'s initial length to 0 by the line

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

You can now compile and try to run our program. Click on the button and Miranda's ironic exclamation will most probably ring out on the text control to express amazement at finally doing what was once so easy.

But why just "most probably"? Why not certainly? Because our code assumes that printout[0] was \0 when the program began. That is probably a good assumption, but it would be better to be sure. Besides, being sure will require that we learn how to initialize variables before the program starts asking for input from the user.

Initializing Variables and the Event Table

When a form is first created, it is not uncommon for it to need a chance to do some computing – such as setting printout[0] to '\0' – before it is ready to accept input from the user. As you might expect, an event, namely EVT_WINDOW_CREATE, is generated. Our problem is how to catch that event and act on it.

In the Code::Blocks editor, open the file GwxMain.cpp if it is not already open. At the bottom, put the lines

```
void GwxFrame::OnCreate(wxWindowCreateEvent& event){
     printout[0] = '\0';
     }
```

The window we are concerned with is precisely GwxFame. These lines say that, as soon as this window is created, printout[0] should be set to \0. The words

```
      wxWindowCreateEvent& event
```

must be exactly those. "OnCreate" is my choice; it could be "OnWindowCreate" or "FirstDay" or any unused legal function name.

You might think this were enough, but it isn't quite so simple. If we stopped with just this code, it would never get executed. We also have to open the file GwxMain.h and add several lines. First, around line 36 you should find the first two of the following three lines.

```
        void OnAbout(wxCommandEvent& event);
        void OnMirandaBtnClick(wxCommandEvent& event);
        void OnCreate(wxWindowCreateEvent& event);
```

You add the third. You will notice that you are adding a member function to the class GwxFrame. These are prototypes for functions found over in the .cpp file. You are adding the prototype for the function you just added. As noted before, the word OnCreate could be OnWindowCreate or FirstDay or any unused legal function name, but it must of course be the same both here in the prototype and in the function's implementation over in the .cpp file.

You might suppose that surely we are now through, but not yet. Still more is needed to get our function called when the event occurs. There are two ways to proceed. One is to use the Connect command and the other is to use the Event Table. wxSmith uses the Connect command, and you can see a number of examples of it already in the code. Let's, however, use the Event Table, because it is both simpler and better documented but without examples in the code.

To use the Event Table, we have to post notice that we are going to do so by ensuring that the line

DECLARE_EVENT_TABLE()

is somewhere – anywhere, it seems – among the members in the class definition in the GwxMain.h file. New versions of wxSmith may have already auto-generated this line. We may as well put it at the end, so that we get something like this

```
class GwxFrame: public wxFrame {
    public:
        Tutorial_9Frame(wxWindow* parent,wxWindowID id = -1);
        virtual ~Tutorial_9Frame();
        void printg(char *fmt, ...);
    private:
        //(*Handlers(Tutorial_9Frame)
        void OnQuit(wxCommandEvent& event);
        . . .
        void OnCreate(wxWindowCreateEvent& event);
        //*)
        //(*Identifiers(Tutorial_9Frame)
        static const long ID_BUTTON1;
        static const long ID_COMBOBOX1;
        . . .
        //*)

        //(*Declarations(Tutorial_9Frame)
        wxButton* Button1;
        wxPanel* Panel1;
        . . .
        DECLARE_EVENT_TABLE()
     };
```

where the . . . show where I have removed for display here some lines that are in the actual code. Note that there is no semicolon after this added line. If wxSmith generated the line, well and good; leave it be.

The Event Table itself, however, is back in the GwxMain.cpp file. wxSmith did not use the event table but it marked the place for it (at about line 55) by two comments. When we put it between these comments we get this:

```
BEGIN_EVENT_TABLE(GwxFrame,wxFrame)
    //(*EventTable(GwxFrame)
    EVT_WINDOW_CREATE(GwxFrame::OnCreate)
    //*)
END_EVENT_TABLE()
```

Note that the OnCreate name matches the name we gave to the function. If we had named the function FirstDay, then we should have had FirstDay here instead of OnCreate. Otherwise everything in these three lines is precisely the way it has to be. Also note the absence of a semicolon at the end of these lines.

Now compile and run and rejoice with Miranda. This is a good place to do a save and back up the files to GwxBack.

Getting Input from the User and Handling the Items List in a Combobox

So far, the only input our program takes from the user is the click of a button. We will now see how to use the combobox to give our program commands in words. We want to be able to write something in the edit control of the combobox, tap the Enter key, and have the program read what we have entered, display it in the Results window, and then do something with it, anything. Moreover, we want the combobox to insert what we have just written as the top item in its list and to push down the items already there. To keep the list of manageable size, we want to eliminate the bottom (oldest) item when there get to be more than 10 items in the list.

To get started, click on the combobox in the Code::Blocks Resources window. Then click on the {} icon in the top line of the Properties browser, so that it becomes the Events browser. Click the EVT_TEXT_ENTER event, drop down the arrow at the right end of the line, and pick “Add new handler”. This event occurs when the user has entered some text in the combobox and taps Enter. WxSmith will create the frame for us. Here is that frame and the code that needs to go in it.

```
void GwxFrame::OnCmdBoxTextEnter(wxCommandEvent& event){
    wxString textFCB = CmdBox->GetValue();
    CmdBox->Insert(textFCB, 0);
    CmdBox->SetValue(wxT(""));
    if(CmdBox->GetCount() == 10)
        CmdBox->Delete(9);
    Results->AppendText(textFCB);
    Results->AppendText(_("\n"));

    strncpy(inbuf, (const char*) textFCB.mb_str(wxConvUTF8),239);
    // anything(); // Go do something with the input.
    }
```

The first line,

```
     wxString textFCB = CmdBox->GetValue();
```

declares the variable textFCB to be a wxString and puts into that string whatever is in the text field (the visible display) of the combo box. It does NOT include in that string the newline character corresponding to the user's tap of the Enter key. The next line,

```
    CmdBox->Insert(textFCB, 0);
```

inserts that string into line 0 of the list of items in the combobox and pushes down all the items already in it. Then the line

```
    CmdBox->SetValue(wxT(""));
```

clears the text field of the combobox to get it ready for whatever the user may next type. The next two lines

```
    if(CmdBox->GetCount() == 10)
            CmdBox->Delete(9);
```

prevent the list of items from growing beyond 10. The items are indexed starting with 0.

Since the textFCB is already a wxString, we can append it directly to the Results window. But because it has no newline at its end, we will also append a newline to keep Results neat.

```
    Results->AppendText(textFCB);
    Results->AppendText(_("\n"));
```

Finally, we convert textFCB to a standard, zero-terminated C-string and copy it to inbuf, which was declared along with printout.

```
    strncpy(inbuf, (const char*) textFCB.mb_str(wxConvUTF8), MAXLINE -
1);
```

For the moment, we will leave the call to "anything()" as just a comment, so that we can compile, run and test the program at this point. Try it! If it works, save it and back it up to Gwx2.

The unwritten function "anything()", however, is important. It is where the real work of the program gets done. It looks at what the user has put in "inbuf" and decides what to do. That may be to run a regression or draw a graph, or anything the program does. Our "inbuf" was declared globally, so if "anything()" does not use "printg()", it does not need to be a member of the class we have been building, But most likely it does use "printg()" – and the current version of "printg()" is a member of the class because it uses "Results" – so with this version of printg() "anything()" has to be a member of the class. That requirement would become a nuisance, so in Tutorial 5, we will make printg() globally accessible.

Let's review. We have started a wxWidgets project in Code::Blocks. Then we got practice with using wxSmith to build a window with box sizers, a panel, some buttons, a combo box, and a TextEdit control. We wrote printg() to work like C's printf() to write text to the TextEdit component called Results. We learned how to initialize variables before the program accepts input from the user. Finally, we learned how to get input text from the command box, CmdBox – a combo box – and how to add items to list of previous commands. In the next tutorial, we will begin writing "anything", though of course we won't call it that.

Remember to open a terminal window and do

```
cd Gwx
cp -r * ../Gwx1
```

to have a record of where we are at the end of this tutorial.

Gwx Tutorial 2: The Select-Chop Interaction

The fundamental mechanism of G7 is that a main program reads a command from the CmdBox and calls a program, gselect(), to decide what to do with it. The gselect() program, in tern, calls a routine called chop() which chops up the command into “bite-sized” pieces that select can understand. For example, if the user gives the command

r gdp = gdp[1], time

successive calls to chop() will return:

r   gdp   =   gdp   [1   ]   ,   time

Ultimately, the gselect() routine will be developed enough to deduce from the r that the user is asking for a regression and call an appropriate routine to read the rest of the line and act on it. Initially, however, we will make gselect() just print the return from chop() to the Results window and continue calling chop() until a new line is encountered. It will then go back to the main program to get another command from the user. We will also develop the chop() routine gradually. Initially, it will break only on blanks and new lines. Thus, when processing the above line, successive calls to the first version of chop() will return

r   gdp   =   gdp[1],   time

The first step is to expand the OnCmdBoxTextEnter() function as follows:

```
void GwxFrame::OnCmdBoxTextEnter(wxCommandEvent& event){
    // Get the command from the CmdBox into textFCB (FCB = From Command Box)
    wxString textFCB = CmdBox->GetValue();
    // Push textFCB into the top of the combo box's items list
    CmdBox->Insert(textFCB,0);
    // Clear the window of the CmdBox
    CmdBox->SetValue(wxT(""));
    // Limit the number of items in the Combo Box drop-down list
    // Numbering of items begins at 0.
    if(CmdBox->GetCount() == 10) CmdBox->Delete(9);
    // Display what was read in Results
    Results->AppendText(textFCB);
    Results->AppendText(_("\n"));
    // Put what was read into inbufAsRead as a standarc C string
    strncpy(inbufAsRead,(const char*) textFCB.mb_str(wxConvUTF8),MAXLINE-1);
    // Copy inbufAsRead into inbuf.
    // This copy will be expanded to do all the text subsitutions.
    short i,j, k;
    i = 0; j= 0; k = 0;
    while(inbufAsRead[i] != '\0'&& j < MAXLINE)
        inbuf[j++] = inbufAsRead[i++];
    if(j >= MAXLINE) {
        printg("%s\n","That line was too tong!");
        j--;
        }
    inbuf[j] = '\0';
    pinbuf = inbuf;
    gselect();
    }
```

The added code is in bold type. The line

```
    strncpy(inbufAsRead,(const char*) textFCB.mb_str(wxConvUTF8),MAXLINE-1);
```

converts the wxString textFCB to a standard, null-terminated C string and then copies up to

MAXLINE characters – namely characters 0 through MAXLINE-1 – to inbufAsRead. This inbufAsRead is then copied over to inbuf. Why make this copy? Why not just read into inbuf in the first place? G makes much use of substitution of text strings in place of %1, %2, … %9 in the command. The plan is to make those substitutions right here, so that `inbuf` will have the expanded text in it. This is a change from the present code, which does the substitution in a routine called feed(), which is called by chop(). That coding was chosen when space was extremely scarce in the computer memory, and wasting space with a big dimension for inbuf was unseemly. The coding, however, has become complex while space has become plentiful, so clearer, simpler code seems desirable.

The string in inbuf is given a standard null ending and pinbuf, defined as a pointer to a character, is set to point to the beginning of inbuf. Both inbuf and pinbuf were declared globally, so they can be accessed as needed. Finally, gselect() is called to decide what to do with the command. Current G uses select() as the name of this function. That name seemed to cause some confusion with another part of the program written by wxSmith, so I changed to gselect().

This initial version of gselect() is short and simple:

```
short GwxFrame::gselect(){
    char s[100];
    while (chop(s) == 'a'){
        printg("%s\n",s);
        }
    return OK;
    }
```

It simply calls chop() and prints the string returned. It does so over and over as long as the return value from chop() is the letter a.

The chop() routine, even at this stage, is a bit more involved.

```
char chop(char *s){
    short i;
    unsigned short cu;
    i = 0;

    while(i < MAXLINE){
        cu = *pinbuf++;
        if(cu == ' '){ // found a space
            if(i== 0) continue; // eat up white space
            s[i] = '\0'; // end the word being returned
            return 'a';
            }
        if(cu == 0){ // found an end of line
            s[i] = '\0'; // end the word being built
            if(i > 0) { // if a word has been started
                pinbuf--; // back up one character
                return 'a'; // return the word that had been started
                }
            else return 'e';
            }
        s[i++] = cu;
        }
}
```

This code is perhaps most quickly understood by first concentrating on the statements

```
i = 0;
while(i < MAXLINE){
    cu = *pinbuf++;
     . . .
    s[i++] = cu;
    }
}
```

These lines say, “Remember that `pinbuf` starts off pointing to position 0 of `inbuf.` Start `i` off at 0, and then copy characters one by one from `inbuf` to `s`, incrementing `pinbuf` and `i` as we go.” If that were all there was to chop(), it would simply copy MAXLINE characters from one place to another. It is the two **`if`** statements that make it stop and return sooner.

The first **`if`** statement deals with what to do if we hit a white space, a `' '` character.

```
if(cu == ' '){ // found a space
    if(i== 0) continue; // eat up white space
    s[i] = '\0'; // end the word being returned
    return 'a';
    }
```

If i is zero, nothing has been started in the s array; and we simply jump to the end of the **while** loop to increment `i`; `pinbuf` has already been incremented. Thus, this white space character has been effectively ignored or, as the comment says, eaten up. If, on the other hand, i is greater than 0, something has been started in s, and we now terminate it with a null and return the character 'a' to signal to the calling program what was found. The chopped, bite-sized morsel of input is in the character string `s`.

The second **`if`** statement deals with what to do when the null that marks the end of a standard C string is encountered.

```
if(cu == 0){ // found an end of line
    s[i] = '\0'; // end the string being built
    if(i > 0) { // if a word has been started
        pinbuf--; // back up one character
        return 'a'; // return the word that had been started
        }
    else return 'e';
    }
```

We first of all end any string which has been started in the s array. Note that we do not increment i. Now if some string has been started in s, we want to return it just as if it had been terminated by a space in the input. But if no string has been started, we want to return the character 'e' for end of line. So we look at the value of i; if it is greater than 0, a string has been been started and we will return the value 'a'. But before doing so, we back up the value pinfbuf pinbuf by one so that on the next call to chop() we will immediately find the null while i is still zero.

You may reasonably ask why I used an unsigned short for cu instead of simply a char.

The code of G7 does so and has the comment “to keep 255 from turning to -1”. I think this comment reflects some experience that I no longer remember precisely, but I thought it wise to maintain the practice.

A prototype for gselect() must be added to public members of the GwxFrame class, as found in the file GwxFrame.h. Here is how the beginning of the class definition should look.

```
class GwxFrame: public wxFrame
{
    public:

        GwxFrame(wxWindow* parent,wxWindowID id = -1);
        virtual ~GwxFrame();
        void printg(char *fmt, ...);
        short gselect();
```

The chop() routine, however, does not use printg() and does not need to be a member of the GwxFrame class. If you put the code for chop() before the code for gselect(), you do not yet need a prototype for chop().

With this code in place, you should be able to click the Code::Blocks build-and-run icon, get the familiar opening window, and then type “one two three” in the command box and see the individual words played back on the Results window as:

```
one two three
one
two
three
```

That is enough for this tutorial. We will come back to refine chop() in Tutorial 4.

I well remember when I got to this stage with the original DOS-based G and then again with Windows-based G7 and realized that I would be able to do all the computing programming in a fairly straight-forward way. The remaining big uncertainty concerned graphs. So let us deal with them immediately in the next tutorial.

Remember to do “File | save everything” and then open a terminal window and do

```
cd Gwx
cp -r * ../Gwx2
```

to have a record of where we are at the end of this tutorial.

Gwx Tutorial 3: Drawing and Saving Graphs

When I was first trying to get G7 to draw graphs, the first job was how to get it to draw any graph at all, never mind whether it was a graph anyone might want. I was in Florence at the time, and happened to remember the story that Pope Boniface VIII sent a messenger to Giotto to ask for a sample of his work. Giotto simply took a brush, dipped it in red paint, and drew a freehand circle and gave it to the messenger. The messenger was not happy with so meager an example; but when it was clear that he could get no more, he took it back to the Pope. Fortunately, the Pope understood draftsmanship better than did the messenger, and gave Giotto the commission at once. So our first graph will be created by the Giotto button which just draws a circle.

Basic Drawing on the Screen

In wxWidgets, one always draws a graph on some kind of Device Context. From the point of view of a programmer using wxWidgets, a Device Context is a black box which spares us from having to know the details of how to send a graph to a printer, or to the screen, or to a bitmap and on to a .jpg file. Exactly the same code creates the graph for use on all three output devices.

The easiest and clearest part of the code we must write is the routine to draw on the device context. We will call it simply Giotto. Here is the code.

```
void Giotto(wxDC &dc){
      // clear the dc to white
      dc.SetBrush(*wxWHITE_BRUSH);
      dc.Clear();

      // Create the color Red
      wxColor Red(255,0,0);
      // Create a pen using it, 6 pixels wide and drawing a solid line
      wxPen myRedPen(Red,6,wxSOLID);
      // Tell dc to use it
      dc.SetPen(myRedPen);

      // Draw a circle with center at 100, 100 with radius 60
      dc.DrawCircle(100,100,80);
      }
```

A pointer to the device context, dc, is passed to this routine. This routine does not know nor care whether it is drawing on the screen or on a bitmap to be saved as a file or sent to a printer. Immediately it sets the dc's “brush” to the white brush, and then clears the dc, that is, it paints the whole thing with the current brush, which is white. Next we need to create a red pen to draw with. wxWidgets provides one, but it is very fine and creates an anemic circle, so we will make our own. wxWidgets also provides a number of colors, but let's mix our own just to see how.

A color on the computer screen is described by the intensities of its red, green, and blue components. Each intensity is described by a number from 0 to 255. Thus (255, 0, 0) is pure, intense red, while green is (0,255,0), and blue is (0,0,255). The line

```
      wxColor Red(255,0,0);
```

creates a wxColor object with the name Red and with the intensities indicated. The line

```
	wxPen myRedPen(Red,6,wxSOLID);
```

creates an wxPen object, named myRedPen, that draws a solid line 6 pixels wide in the color Red. Alternatives to wxSolid are are wxDOT, wxLONG_DASH, wxSHORT_DASH, wxDOT_DASH, and wxTRANSPARENT.

The line

```
	dc.SetPen(myRedPen);
```

tells dc to draw with myRedPed. Finally, the line

```
	dc.DrawCircle(100,100,80);
```

tells the dc to draw a circle of radius 60 pixels and with a center 100 pixels to the right of the left border and 100 pixels ***down*** from the ***top*** border. (Note that the meaning of the vertical coordinate is the opposite of the one customary in mathematics.)

Now that we have Giotto ready to draw on any device context, how do we make him draw on the screen? Start up Code::Blocks and open the Gwx project. We are going to need a panel in a Dialog for Giotto to draw on. On the main menu of Code::Blocks there is a wxSmith item. Click it and pick "Add wxDialog".

We could instead picked "Add wxFrame". There is one important reason for picking a wxDialog instead of a wxFrame. When G draws a graph, we want the program to pause and give the user a chance to look at it, not rush on with the next calculations in a command file, perhaps throwing scores of graphs on the screen. A wxDialog has a ShowModal function which will show the graph and stop the program until the user closes it; a wxFrame has no ShowModal function, but only a Show function, which does not stop the program to give the user a chance to look at the graph.

When the window comes up asking for the Class Name and suggesting "NewDialog", let's instead call it "GiottoDialog". Accept the other defaults suggested, and finish adding the dialog. You will then be greeted by another field of dots, but they represent the new GiottoDialog, not the main frame. In the dialog's properties browser, drop down the Style item and check the boxes for wxRESIZE_BORDER, wxMAXIMIZE_BOX, and wxMINIMIZE_BOX in addition to those already checked by wxSmith.

Before going further, we must fix up what happens to this dialog when the user tries to close it. Click on the {} icon at the top of the Properties browser to turn it into the Events browser and click the EVT_CLOSE item. Click the down-arrow at the right end of the line and click "Add new handler". The C++ code associated with the Dialog appears. At the bottom of the file you should see these lines:

```
void GiottoDialog::OnClose(wxCloseEvent& event)
	{
	}
```

This is where control comes when the user tries to close the GiottoDialog window. As you can plainly see, nothing will happen; and the window will hang around until the main window is closed. If, however, we put into the body between the braces the Close() command as before, closing this window will close the whole application. Instead, we must put Destroy(), which will wipe out the present window, but not kill the whole program. So we should have:

```
      void GiottoDialog::OnClose(wxCloseEvent& event)
      {
            Destroy();
      }
```

This is a good time to copy Giotto() from the above text and place it just below the OnClose block of code.

Now we must give Giotto something to draw on. Get back to the field of dots (click on GiottoFrame.wxs in the bar above the C++ code) and put on it a box sizer and in the sizer put a panel. This panel is where we will draw our picture for viewing on the screen. In its properties browser, check its Expand box, uncheck Default size, and fill in Width as 200 and Height as 200. Finally, find the Style property marked on the left by a little square inside it. Click on the + to drop down more properties and click on the last, Full_Repaint_on_Resize.

We now need to add a bit of code for the Paint event for this panel. So click on the panel, click on the {} above the Properties browser to turn it into the Event browser, find EVT_PAINT (it should be at the top of the list), click on it, then click on the down arrow at the right edge of the line, and pick "Add new handler". Accept the suggested name and click OK.

You find yourself right back in GiottoFrame.cpp just below where you put Giotto() and presented with the following frame for writing the code to handle this event:

```
void GiottoFrame::OnPanel1Paint(wxPaintEvent& event)
{
}
```

We need only add two lines in the middle of the frame, as shown here:

```
void GiottoDialog::OnPanel1Paint(wxPaintEvent& event)
{
          wxPaintDC dc( Panel1 );
          Giotto(dc);
}
```

The first of those two lines creates `dc` as a Device Context of the wxPaintDC variety that will draw on Panel1. The wxWidgets book says a wxPaintDC is "for drawing on the client area of a window during a paint event handler." In other words, it creates dc and sets up its connection to Panel1. The second line calls Giotto to draw on dc. These two lines will be executed whenever the operating system paints the panel. It will do so when the dialog containing it is first displayed, or moved, or resized, or uncovered after being covered.

Now we just have to make the Giotto button on the main window display the panel in GiottoDialog. So click on GwxDialog.wks in the line above the code editor to get back to the main window; go nearly to the top and add under the first group of #include statements

```
 #include "GiottoDialog.h"
```

so that with the neighboring statements it looks like this:

```
#include <wx/msgdlg.h>
#include "GiottoDialog.h"
```

```
//(*InternalHeaders(Tutorial_8Frame)
```

This “include” has to be added because otherwise the main program would not know about the panel it is supposed to open in GiottoDialog.cpp.

We already have the “Giotto” button; we just have to add a handler for its OnClick event. Double click on the button. The frame for adding the event handler for the button opens up and we fill it in as follows:

```
void GwxFrame::OnGiottoBtnClick(wxCommandEvent& event)
{
    GiottoDialog* dlg = new GiottoDialog(this);
    dlg->ShowModal();
}
```

At last, we have a program we can build and run. Click the Code::Blocks build-and-run icon. When the program starts, click the Giotto button and you should see the famous O appear on the screen.

You will notice that the Gwx window is “grayed” and clicking on it has no effect until we close the picture. That behavior is a consequence of the

```
    dlg->ShowModal();
```

line in the code. If instead we had written

```
    dlg->Show();
```

we could click the Miranda button, type into the command box, or even click the Giotto button over and over creating many copies of the graph.

You may recall that we asked for a minimize button on the Giotto Dialog window, but you will see that we did not get it. That also has to do with the fact that the window is being shown modally. What would happen if the user minimized a modal window?

Let's review what happens when you click the Giotto button. The OnGiottoBtnClick event handler is called and a new instance of a GiottoDialog is created and shown. Showing it requires that the panel be painted, and Giotto() is called to do the painting.

A Stretchable Canvas for Giotto

If, when our project is running and Giotto's O is on the screen, we drag the lower right corner of its frame down and to the right, we will find that the frame and the panel in it expand – thanks to the wxRESIZE_BORDER property of GiottoDialog and the Expand property of the panel – but the O remains the same size in the upper left corner. Similarly, if you click the Maximize icon in the top frame, the window resizes to fill the whole screen, but the circle remains the same size in the top left corner. You may recall that we asked for a Minimize button also, but it is not there. That is because we used ShowModal to show the dialog. That means nothing can be done until this window is closed, so if we minimized it, we would hang the computer.

It would be nice if the size and position adjusted. To make that happen, we need to get the dimensions of the panel Giotto will draw on, pass them to Giotto, and have him adjust his O accordingly. Here is the code for the new versions of OnPanel1Paint and Giotto:

```
void GiottoDialog::OnPanel1Paint(wxPaintEvent& event){
    wxPaintDC dc(Panel1);
    wxSize sz = GetClientSize();
    Giotto(dc,sz);
    }

void Giotto(wxDC &dc,wxSize &sz){
     dc.SetBrush(*wxWHITE_BRUSH);
     dc.Clear();

     // Create the color Red
     wxColor Red(255,0,0);
     // Create a pen using it, 6 pixels wide and drawing a solid line
     wxPen myRedPen(Red,6,wxSOLID);
     // Tell dc to use it
     dc.SetPen(myRedPen);
     // Find Center
     int x = sz.x/2;
     int y = sz.y/2;
     int r = 0.4*wxMin(sz.x,sz.y);

     // Draw a circle with center at (x, y) with radius r
     dc.DrawCircle(x,y,r);
     }
```

Now when we build and run and stretch the window while running, the circle will enlarge as well.

Drawing Straight Lines, an Octagon

In G, we really never need to draw circles, so let's draw some short straight lines. In keeping with our Florentine theme and the octagonal base of Brunelleschi's great cupola, let's draw an octagon around the circle. At the same time, and with an eye to the next step, let's use the variables x and y for the coordinates of the center of the circle and the variable r for its radius. We just need to add below the

```
     dc.DrawCircle(x, y, r);
```

the following lines:

```
    // Draw lines to form an octagon around the circle.
    // h is half the length of one side of the octagon.
    // tangent of 22.5 degrees = .414214;
    h = 0.414214*r;
    dc.DrawLine(x+r,y+h,x+r,y-h); // right vertical
    dc.DrawLine(x-r,y+h,x-r,y-h); // left vertical
    dc.DrawLine(x-h,y-r,x+h,y-r); // top horizontal
    dc.DrawLine(x-h,y+r,x+h,y+r); // bottom horizontal
    dc.DrawLine(x-r,y-h,x-h,y-r); // top left slant
    dc.DrawLine(x+h,y-r,x+r,y-h); // top right slant
    dc.DrawLine(x-r,y+h,x-h,y+r); // bottom left slant
    dc.DrawLine(x+h,y+r,x+r,y+h); // bottom right slant
```

The format of the DrawLine(x1,y1,x2,y2) function is clear; it draws a line from the point with (x, y) coordinates specified by the first two integers to the point specified by the last two integers. Virtually all of the G graphs – except their titles – are created with the DrawLine function. With this added code, you can again build and run Gwx and click the Giotto button. Be sure to try the maximize icon.

Putting Text on the Graph

Graphs need titles and often other text. We will illustrate by writing at the top of Giotto's drawing the text, "Give Boniface This!".

The first step is to set the color for the text. We will just use black and make use of wxWidgets predefined color wxBlack.

```
    // Set text color
    dc.SetTextForeground( *wxBLACK);
```

The next step is to decide on the point size for the text. After some experimentation, I found that setting the point size (an integer) to six percent of the window height in pixels gave a pleasing proportion. So we have:

```
    // Adjust the point size of the font to the height of the window
    int PointSize = 0.06*sz.y;
```

Now we need to create a wxFont object with this point size. The statement will be of the form

wxFont GiottoFont(*size, family, style, weight, underline*);

The *size* argument is just an integer, the point size. The *family* argument is more involved. To aid in making programs run on a variety of computers, wxWidgets distinguishes six basic "font families". For example, most systems will have Helvetica or Arial, but not necessarily both. If we specify that the font should be a member of the wxFONTFAMILY_SWISS, the program will use whichever of these it finds. The names of the font familes are wxFONTFAMILY_SWISS, wxFONTFAMILY_ROMAN, wxFONTFAMILY_SCRIPT, wxFONTFAMILY_MODERN, wxFONTFAMILY_DECORATIVE, wxFONTFAMILY_DEFAULT. MODERN might better be called MONO, because it is always a monospaced font, usually Courier. DEFAULT leaves the choice up to wxWidgets. We will use wxFONTFAMILY_ROMAN to get a nice serif font.

For the *style* argument, we have three possibilities: wxNORMAL, wxITALIC, and wxSLANT.

We will use wxNORMAL. For the *weight* argument, we again have three possibilities: wxNORMAL, wxBOLD, and wxLIGHT. We will choose wxNORMAL. Finally, the *underline* argument can be *true* or *false;* we will use *false.* Actually, there are two more optional arguments, *face name* and *encoding*. If present, *face name* overrides the font-family mechanism and tries to use the specified font. The *encoding* argument relates to internationalization. I have no experience with either of these two optional arguments. So finally, here is our declaration of the GiottoFont:

```
//Create a PointSize serif font, that is not bold, not italic, not underlined
wxFont GiottoFont(PointSize,wxFONTFAMILY_ROMAN,wxNORMAL,wxNORMAL,false);
```

Now of course we have to tell dc to use this font.

```
// Tell dc to use this font
dc.SetFont(GiottoFont);
```

Now we want to find out how big our text will be so we can center it and reduce the available space vertically that Giotto has to draw in. Of course, we will have to convert the text to a wxString to do so. So here we create the wxString *text*:

```
wxString text("Give Boniface This!", wxConvUTF8);
```

How big will *text* be? The GetTextExtent() function of the dc will tell us, but we have have to pass to it the text itself and pointers to two variables of the type wxCoord. So we have:

```
wxCoord textwidth, textheight;
dc.GetTextExtent(text, &textwidth, &textheight);
```

Now we figure out where the text should begin if it is to look centered, but we want to take care that we do not start at a negative position. So we have:

```
// left is where the text should start
int left = wxMax((sz.x – textwidth)/2,0);
```

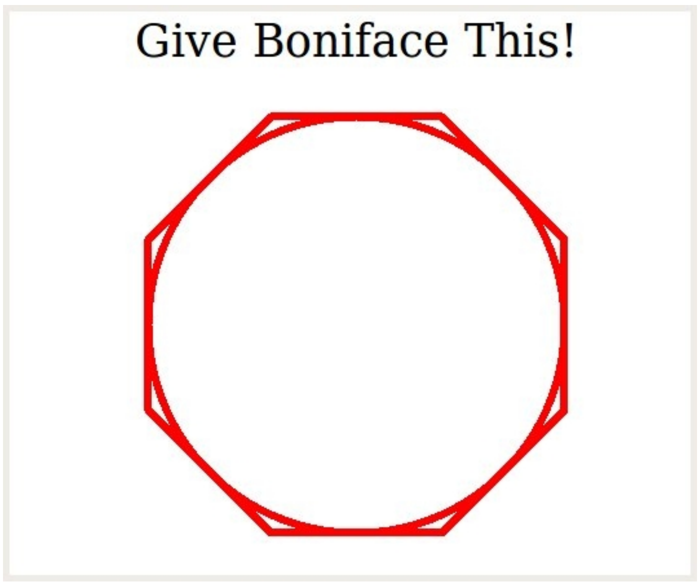
Finally, we are ready to draw the text on the dc. The second and third arguments are the coordinates of the upper left corner of the text.

```
dc.DrawText(text,left,2);
```

The drawing of the circle and the octagon proceed as before, but we must take into account the reduction of the available vertical space caused by putting the text across the top. It affects both the vertical limit on the radius and the location of the center of the circle. Here's the code:

```
int x = sz.x/2;// Horizontal limit on radius
int y = (sz.y - textheight)/2; //Vertical limit on radius
int r = 0.8*wxMin(x,y);
y = y + textheight - 2; // Vertical center of the circle
dc.DrawCircle(x, y, r);
```

Here is the final result as obtained by a screen shot.



The final version of Giotto() follows.

```
void Giotto(wxDC &dc,wxSize &sz){
      // clear the dc to white
      dc.SetBrush(*wxWHITE_BRUSH);
      dc.Clear();

      // Put some text on the screen
      dc.SetTextForeground( *wxBLACK);
      // Adjust the point size of the font to the height of the window
      int PointSize = 0.06*sz.y;
      //Create a PointSize serif font, that is not bold, not italic, not underlined
      wxFont GiottoFont(PointSize,wxFONTFAMILY_ROMAN,wxNORMAL,wxNORMAL,false);
      // Tell dc to use this font
      dc.SetFont(GiottoFont);
      wxCoord textwidth, textheight;
      wxString text("Give Boniface This!", wxConvUTF8);
      dc.GetTextExtent(text, &textwidth, &textheight);
      // left is where the text should start
      int left = wxMax((sz.x - textwidth)/2,0);

      dc.DrawText(text,left,2);

      // Create the color Red
      wxColor Red(255,0,0);
      // Create a pen using it, 6 pixels wide and drawing a solid line
      wxPen myRedPen(Red,6,wxSOLID);
      // Tell dc to use it
      dc.SetPen(myRedPen);

      int x = sz.x/2;// Horizontal limit on radius
      int y = (sz.y - textheight)/2; //Vertical limit on radius
      int r = 0.8*wxMin(x,y);
      y = y + textheight - 2; // Vertical center of the circle
      dc.DrawCircle(x, y, r);
      //
      // Draw an octagon around the circle
      // tangent of 22.5 degrees = .414214;
      int h = 0.414214*r;
      dc.DrawLine(x+r,y+h,x+r,y-h); // right vertical
      dc.DrawLine(x-r,y+h,x-r,y-h); // left vertical
      dc.DrawLine(x-h,y-r,x+h,y-r); // top horizontal
      dc.DrawLine(x-h,y+r,x+h,y+r); // bottom horizontal
      dc.DrawLine(x-r,y-h,x-h,y-r); // top left slant
      dc.DrawLine(x+h,y-r,x+r,y-h); // top right slant
      dc.DrawLine(x-r,y+h,x-h,y+r); // bottom left slant
      dc.DrawLine(x+h,y+r,x+r,y+h); // bottom right slant
      }
```

Saving Graphs as .PNG and .JPG Files

When the user closes the GiottoDialog, we will save the drawing to a PNG and to a JPG file. More correctly said, we will create a memory Device Context (called "memDC") and a bitmap (called "paper"), assign "paper" to "memDC" to write on, have Giotto draw on memDC, then free "paper" from memDC and save it to a .png and to a .jpg file. Saving to both types of files is an experiment to look at the size of the file and the quality of the result. The .png file is expected to win, that is, to be smaller and to reproduce better, because, in the words of Wikipedia, "when storing images that contain text, line art, or graphics – images with sharp transitions and large areas of solid color – the PNG format can compress image data more than JPEG can, and without the noticeable visual artifacts which JPEG produces around high-contrast areas." Let's see what happens.

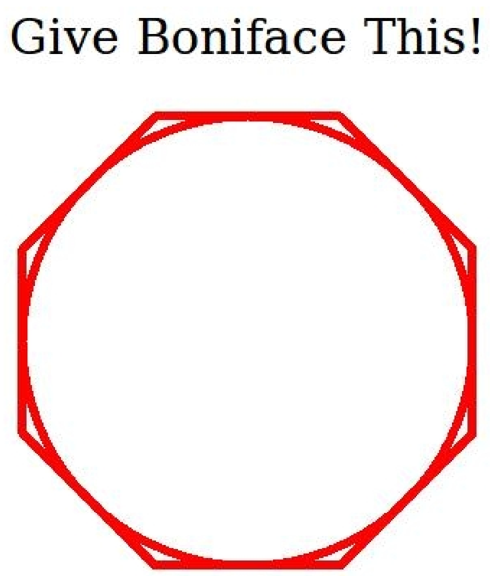
Because the Giotto program is over in a different .cpp file, we will need a prototype, like this:

```
void Giotto(wxDC &dc,wxSize &sz);
```

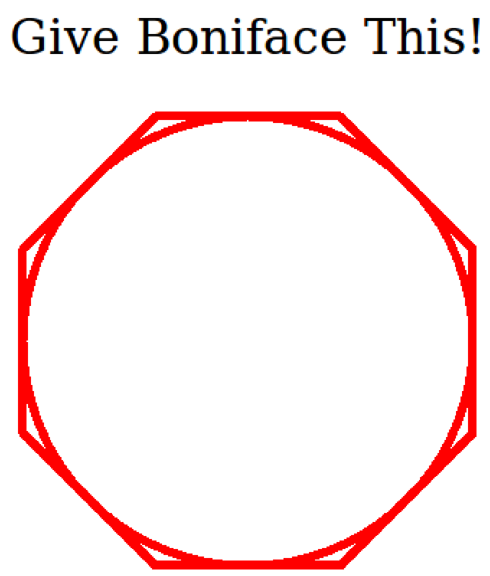
Then the expanded response to the Giotto button click is the following.

```
void GwxFrame::OnGiottoBtnClick(wxCommandEvent& event){
      GiottoDialog *dlg = new GiottoDialog(this);
      dlg->ShowModal();

/*    To save our drawing to a file, we first create a bitmap, then
      a memory DC, then hand the bitmap to the memory DC to use as
      paper to draw on, then have Giotto draw on it, then free the
      bitmap from the DC and make it write itself as a .png file.
*/
      // Create a bitmap 400 pixels wide and 430 pixels high.
      // Call it "paper" because Giotto will draw on it.
      wxBitmap *paper = new wxBitmap( 400,430);

      // Create a memory Device Context
      wxMemoryDC memDC;

      // Tell memDC to write on “paper”.
      memDC.SelectObject( *paper );

      // Create a wxSize object with the size of “paper”.
      wxSize sz(400,430);

      // Call Giotto to draw on memDC
      Giotto(memDC,sz);
// Tell memDC to write on a fake bitmap;
// this frees up "paper" so that it can write itself to a file.
memDC.SelectObject( wxNullBitmap );

// Put the contents of "paper" into a png and into a jpeg file.
paper->SaveFile( _T("Giotto.png"), wxBITMAP_TYPE_PNG,
      (wxPalette*)NULL );
paper->SaveFile( _T("Giotto.jpg"), wxBITMAP_TYPE_JPEG,
      (wxPalette*)NULL );

delete paper;
delete sz;
}
```

Click the build-and-run icon, click the Save button. Then start your word processor and insert Giotto.jpg and Giotto.png, as shown below. I see no difference in the quality of the result. The Giotto.jpg file, however, is 16.1 kilobytes, while the Giotto.png is only 9.3 kilobytes. So the experiment came out as expected, and we will save Gwx graphs as .png files.



Giotto from the Giotto.jpg file.



Giotto from the Giotto.png file.

At this point, my major questions about the usability of wxWidgets for writing a cross-platform version of G7 have been largely resolved. It can be done. Of course, lots of work remains. Every form displayed by G7 must be rewritten, and there are about twenty of them. The use of grids may prove to be quite different. Various operating system calls will have to be changed. I need to get a better understanding of the debugger under Code::Blocks. And my project to record progress as a sequence of tutorials is going to be time consuming. But I think all this is doable and will give us a firm basis for G in the future.

Gwx Tutorial 4: The Debugger, an Improved Chop(), and Text Substitution

As long as you make no mistakes, you don't need the debugger; but in working on this tutorial, I made a simple mistake which might have held me up for days had it not been for the debugger. So it's time to have a look at it. We will then refine the chop() routine so that it distinguishes:

numbers (including those in scientific notation)
words beginning with a letter but including the digits and $ and _
arithmetic operators (= + - * and /) # and " and other punctuation.
Finally, we will program text substitution, so the %0, %1, %2, … %9 in input lines will be replaced by specified text.

The Code::Blocks Debugger with GNU GCC

For the debugger to work effectively, the program must be compiled with a flag (-g) set so that the compiler generates a symbol table. In Code::Blocks, to set the flag, click Project | Build options … on the main menu. The first option is “Produce debugging symbols [-g]”. Be sure that its box is checked, along with that of “Enable all compiler warnings.” Be sure that the box for “Strip all symbols from binary [-s] is NOT checked. Click Build | Rebuild to ensure that the entire project has been compiled with these options. After doing this once, it should be sufficient to use just the “build and run” icon as usual.

If you want to have the convenient feature of the the debugger showing the value of an expression when you hover the the cursor over it, you will need to change the default debugger settings. On the Code::Blocks main menu, click Settings. The click “Compiler and debugger settings.” Click the big “Debugger Settings” icon in the left column. Then put a check mark in the box of the option “Evaluate expression under the cursor”.

Another device for showing the values of variables is to set “watches.” To be able to do so, click “Debug” on the Code::Blocks main window; then click “Debugging windows” and check the box for Watches.

Decide where you want a breakpoint and click just a tiny bit to the right of the line number. A red circle should appear.

Start the debugger. Click Debug | Start. The program should run to the breakpoint and put a white triangle inside the red circle. Now click Debug | Edit watches and in the window which comes up, click the “Add” button, and enter the name of the variable you want to watch. You should see its name and current value in the watch window. Or, if you have set up “cursor hovering” you can use it.

Supposedly F8 will start the debugger and F7 will go to the next line of the program, while F4 will run to the cursor and Shift F7 will step into a function call. In my experience, however, I found it worked best to use Debug | Start to get started and to use Debug | Next line the first time I needed the Next line. After that, the F7 worked.

That should get you started. It is a good idea to try this out before you actually need it. The maintainers of Code::Blocks say that the “nightly builds” since 10.05 are a big improvement in the debugging area. I have not tried them.

Refining chop()

In the second tutorial, we introduced a simple version of chop(0) to illustrate the interaction of (1) the user's entering a line in the Command Box and tapping the Enter key (2) the gselect() routine, and (3) the chop() routine. That version of chop() recognized only spaces and tabs as separators of components in the line. G requires a more sensitive division. The prototype for chop() is

    char chop(char *s);

where s is a pointer to space where chop can put null-terminated cstring. Thus, two things are returned by chop: (1) the value of the function and (2) the contents of string pointed to by s.

Here is an example of how it should chop up a line.

Line: f abc=2.5*(ghi+jkl)

Successive calls to chop() should return:

```
value of function:   a  a    =    n    *  (  a    +  a    )  e
string in s[]     :  f  abc  =   2.5   *  (  ghi  +  jkl  )
```

The string in s[] is always a proper, null-terminated C string.

Any string beginning with a letter returns the character 'a' as value of the function and the characters in s[] should extend until something other than a letter, a digit, a '.', '$' or a '_' is encountered. These strings are typically G commands or variable names, which must begin with a letter but may include digits and '$', '_', and '.' characters. Any other character breaks the buildup of the string and causes chop() to return.

Any string of characters beginning with a digit returns as value of the function the character 'n' (for number) with the number as an ASCII string in s[]. This string may include, besides digits, a '.' and the expressions E+ and E- to allow for scientific notation of numbers – e.g. 2.72E+05. The string extends until a character other than a digit, a '.', or E+ or E- is encountered, whereupon chop() returns.

Spaces and tabs break the build up of the type 'a' and type 'n' strings, but are then just "eaten up" by chop() which passes on looking for some other kind of character.

A string of characters beginning with a quotation mark, " , returns the " as the value of the function and everything inside the quotes – but not the quotes themselves – in s[]. These may be titles for graphs or regressions.

A string of characters beginning with a # returns the # as the value of the function and the # and everything following it on the line in s[]. Essentially, the # and everything following on the line will be considered a comment. These characters will be shown on the Results screen, but not considered a command.

A null – the '\0' at the end of a line – breaks off and returns any string that was being built and then, on the next call, returns 'e' for End of Line.

Any other character – and these include = + - * / ( ) [ ] – returns the character itself as the value of the function and as the content of the string in s[]. More precisely, s[] will have the character in position 0 and '\0' in position 1.

Here is a chop() which does all this. Note that the value of the function, chopreturn, is set as soon as the type of string is known. Then, when we hit the end of the string, we just do

```
return chopreturn;
```

Among the #include statements near the top of GwxMain.cpp we need to have:

```
#include <ctype.h>
```

for the use of the isalnum() and isdigit() functions.

```
char chop(char *s){
    short i;
    unsigned short cu;
    i = 0;
    char chopreturn = 'a';

    while(i < MAXLINE){
        cu = *pinbuf++;

        // Is it a space or tab?
        if(cu == ' ' || cu == '\t'){ // found a space or tab
            if(i== 0) continue; // eat up white space
            s[i] = '\0'; // end the word being returned
            return chopreturn;
            }

        // Is it an End of line
        if(cu == 0){ // found an end of line
            s[i] = '\0'; // end the word being built
            if(i > 0) { // if a word has been started
                pinbuf--;     // back up one character
                              // so the EOL will be read next time
                return chopreturn; // return the word that had been started
                }
            else return 'e';
            }

        // Is it a number?
        if(i == 0 && (isdigit(cu) || cu == '.')){
            // found a digit or decimal point
            chopreturn = 'n';
            s[i++] = cu;
            while(isdigit(*pinbuf) || *pinbuf == '.'
                  || *pinbuf == 'E' // allow for scientific notation
                  || (*pinbuf == '+' && s[i-1] == 'E')
                  || (*pinbuf == '-' && s[i-1] == 'E')){
                s[i++] = *pinbuf++;
                }
            s[i] = '\0';
            return chopreturn;
            }

        // Is it a string in quotes?
        if (i == 0 && cu == '"'){ // Found something in quotes
            chopreturn = '"';
            while(*pinbuf != '"' && *pinbuf != '\0'){
                s[i++] = *pinbuf++;
                }
            s[i] = '\0';
            return chopreturn;
            }

        // Is it a # ? If so, the rest of the line is a comment
        if (i == 0 && cu == '#'){
            chopreturn = '#';
            s[i++] = '#';
            while(*pinbuf != '\0'){
                s[i++] = *pinbuf++;
                }
            s[i] = '\0';
            return chopreturn;
            }

        // Is it a letter?
        if(i == 0 && (isalnum(cu))){
            chopreturn = 'a';
            s[i++] = cu;
            while(isalnum(*pinbuf) || isdigit(*pinbuf) || *pinbuf == '.'
                  || *pinbuf == '_' || *pinbuf == '$' ){
                s[i++] = *pinbuf++;
                }
            s[i] = '\0';
            return chopreturn;
            }
        // Operators and other punctuation
        s[i++] = cu;
        s[i] = '\0';
        chopreturn = cu;
        return chopreturn;
        }
    // Should not get here, but if we do
    s[MAXLINE-1] = '\0';
    return ERR;
    }
```

Replace the previous chop() with this one and test it out. It may have to undergo modification later, but right now it seems to work well and is reasonably simple.

Making Text Substitutions

One of G's most useful abilities is text substitution. When it is executing an “add” or “fadd” file, it may come upon the symbol %0 or %1, or %2 and so on up to %9. These symbols indicate that previously established text strings should be substituted for these symbols in the input line. For example, if %0 has been defined as Aaa and the command

    f %0 = x + y

is given, chop() will see the command as

    f Aaa = x + y

The current coding in G7 is rather complex, and I want to try something much simpler as shown here.

First of all, we have to reserve some space for storing the ten possible strings to be substituted. We will do it right along with the reservation of space for inbuf and inbufAsRead.

```
#define MAXPRINTOUT 500
#define MAXLINE 500
#define MAXSUBSTITUTES 100 //Maximum length of any one susbstitute
string
#define OK 1
#define ERR -1

char printout[MAXPRINTOUT], inbufAsRead[MAXLINE], inbuf[MAXLINE],
*pinbuf;
char substitutes[10][MAXSUBSTITUTES];
```

We also need to get some text into at least the first two rows of the *substitutes* array so we can test our program for making the substitutions. In the current G7, this step is usually done by arguments on an *add* command or from the argument file of a *fadd* command. We don't have that machinery set up yet for Gwx, so – just for testing the substitution code – we will initialize the first two rows of *substitutes* when the GwxFrame is created, like this:

```
void GwxFrame::OnCreate(wxWindowCreateEvent& event){
    printout[0] = '\0';
    // Just for testing substitutes
    strcpy(substitutes[0],"Aaa");
    strcpy(substitutes[1],"Bbb");
    }
```

Now we turn to what is done in OnCmdBoxTextEnter right after the user types something and taps the *Enter* key. This is where the substitution is actually done. Here is the code, with the new part in bold.

```
void GwxFrame::OnCmdBoxTextEnter(wxCommandEvent& event){
    wxString textFCB = CmdBox->GetValue();
    CmdBox->Insert(textFCB, 0);
    CmdBox->SetValue(wxT(""));
    if(CmdBox->GetCount() == 10) CmdBox->Delete(9);
    Results->AppendText(textFCB);
    Results->AppendText(_("\n"));
    // Put what was read from the CmdBox into inbufAsRead as a standard C string
    strncpy(inbufAsRead,(const char*) textFCB.mb_str(wxConvUTF8),MAXLINE-1);
    // Copy inbufAsRead into inbuf with text substitution.
    short i,j, srow, scol;
    i = 0; j= 0; srow = 0; scol = 0;
    // i keeps track of where we are in inbufAsRead.
    // j keeps track of where we are in inbuf.
    while(inbufAsRead[i] != '\0'&& j < MAXLINE && i < MAXLINE){
        // check for text substitution signals: %0, %1, %2 etc.
        if(inbufAsRead[i]== '%' && isdigit(inbufAsRead[i+1])){
           // make text substitution
            srow = inbufAsRead[i+1]-48; //'0' is ASCII 48
            i += 2;
            scol = 0;
            while(substitutes[srow][scol] != '\0' && j < MAXLINE &&
                scol < MAXSUBSTITUTES){
                inbuf[j++] = substitutes[srow][scol++];
                }
            }
        else inbuf[j++] = inbufAsRead[i++];
        }
    if(j >= MAXLINE) {
        printg("%s\n","That line was too tong!");
        j--;
        }
    inbuf[j] = '\0';
    pinbuf = inbuf;
    gselect();
    }
```

With the comments, the program is pretty self-explanatory. The string *inbufAsRead* is what was read from the command box. The string *inbuf* is *inbufAsRead* expanded by replacing %0, %1, etc. by the appropriate text strings.

The line

```
  srow = inbufAsRead[i+1]-48; //'0' is ASCII 48
```

perhaps needs a word of explanation. The ASCII character in inbufAsRead[i+1] will be a 0, or a 1, or a 2, … or a 9. The numeric values of these ASCII characters are 48, 49, 50, etc., so by subtracting 48 we get the numeric value of the row in *substitutes.* Perhaps it would be better practice to use the atoi() function to do the conversion, but the result would be the same.

In my testing of it, this code works fine and is much simpler than what I and others wrote for G7. It is perhaps more extravagant with space than I would have dared to be back in the days of DOS and the 640 kilobyte limit on programs.

Tutorial 5: Making printg() Global.

Back in Tutorial 1, when we introduced the printg() function, we made it a member of the GwxFrame class, the class I think of as the interface, the class which puts on the screen the form with the command box and the Results area for displaying output text. By doing so, we saved ourselves some complexity at the time and could concentrate on printg(), which had enough complexity. But as we now go further, the decision to make it a member function would soon come back to haunt us. Every time we wanted to print "Hello!" from a function that was not a member of the interface class, we would have to write something like

MainFrame->printg("Hello!\n");

instead of just

printg("Hello!\n");

Worse still, the value of MainFrame was created as a local variable and would not be available, so essentially our printg() is usable only from members of the interface class. That is too limiting. We need for printg() to be available everywhere.

The first step in the solution to this problem is to make printg() a "friend" rather than a member of the interface class. So edit GwxMain.h and change

```
class GwxFrame: public wxFrame
{
    public:

        GwxFrame(wxWindow* parent,wxWindowID id = -1);
        virtual ~GwxFrame();
        void printg(char *fmt, ...);
        short gselect();
```

to

```
class GwxFrame: public wxFrame
{
    public:

        GwxFrame(wxWindow* parent,wxWindowID id = -1);
        virtual ~GwxFrame();
        friend void printg(char *fmt, ...);
```

Notice both the "friend" keyword and the fact that all reference to gselect() has been removed. It needed to be in the class only because it used printg(), but printg() is going global.

Now let's look at printg() itself. It needs the following code

```
#define MAXPRINTOUT 500
#define MAXLINE 500
#define MAXSUBSTITUTES 100 //Maximum length of any one susbstitute
string
#define OK 1
#define ERR -1
```

```
char printout[MAXPRINTOUT], inbufAsRead[MAXLINE], inbuf[MAXLINE],
*pinbuf;
char substitutes[10][MAXSUBSTITUTES];

void printg(char *fmt, ...){
    char buf[MAXLINE];
    va_list args;
    va_start(args, fmt);
    vsprintf(buf, fmt, args);
    va_end(args);

    short lenb, lenp;
    lenb = strlen(buf);
    lenp = strlen(printout);
    if(lenb+lenp >= MAXLINE-2){
        wxString wxPO(printout, wxConvUTF8);
        MainFrame->Results->AppendText(wxPO);
        printout[0] = '\0';
        }
    strcat(printout, buf);
    lenp = strlen(printout);
    // If the last character in printout is \n, then it is time to
    // print
    if(printout[lenp-1] == '\n'){
        // convert printout to a wxString, which we will call wxPO
        wxString wxPO(printout, wxConvUTF8);
        MainFrame->Results->AppendText(wxPO);
        printout[0] = '\0';
        }
     }
```

The only changes in the whole routine are, first, to change

```
void GwxFrame::printg(char *fmt, ...){
```

to

```
void printg(char *fmt, ...){
```

and to insert **MainFrame->** in front of Results in two places. But what is this MainFrame business? MainFrame is the pointer to our interface class.

Open the file GwxApp.cpp; use File | Open to do so. You will see

```
bool GwxApp::OnInit()
{
    //(*AppInitialize
    bool wxsOK = true;
    wxInitAllImageHandlers();
    if ( wxsOK )
    {
      GwxFrame* Frame = new GwxFrame(0);
      Frame->Show();
      SetTopWindow(Frame);
    }
    //*)
    return wxsOK;
}
```

See that variable Frame? That is the precious pointer to our interface class. But it is a local variable and we need to use it elsewhere. So we need to make it a global variable. And in the process, let's give it a more distinctive name, like MainFrame. The code we want is:

```
IMPLEMENT_APP(GwxApp);

GwxFrame* MainFrame;

bool GwxApp::OnInit() {
    //(*AppInitialize
    bool wxsOK = true;
    wxInitAllImageHandlers();
    if ( wxsOK ){
      MainFrame = new GwxFrame(0);
      MainFrame->Show();
      SetTopWindow(MainFrame);
     }
    //*)
    return wxsOK;
    }
```

Note that **MainFrame** is now a global variable. Note further that we do **NOT** have

```
     GwxFrame* MainFrame = new GwxFrame(0);
```

That would have created a second MainFrame variable, this one local, and it would have hidden the global MainFrame variable. (I made this mistake.)

Now that we have made MainFrame a global variable in GwxApp.cpp, we need to let printg() over in GwxMain.cpp know about it. So just above the code for printg(), we insert the line

```
     extern GwxFrame* MainFrame;
```

Finally, since gselect is no longer a member of the interface class, we need to change its the top line of its code from

```
short GwxFrame::gselect(){
```

to just

```
short gselect(){
```

Make all of those changes and, if I have not forgotten to tell you something I did, you should be able to compile and run and get a Gwx that seems to work just like the version at the end of the previous tutorial. The difference, however, is enormous. We can now go forward with introducing the various functions of Gwx without having to make them all members of the interface class just to use the printg() function.

* * *

This is where work was during the 20th Inforum Conference in Florence in September of 2012. It is perhaps a bit disappointing that the program will not yet do anything useful. On the other hand, some of the biggest question marks in my mind about the usability of wxWidgets with Code::Blocks for this project have been favorably resolved. One of the most fundamental pieces of the code, chop(), has been rewritten and clarified. The next tutorial will tackle the problem of creating and assigning a G data bank, putting data into it, and displaying data numerically. Then we can take up writing the f command, and then the r command and graphing time series data.