WxSmith tutorial: Keyboard Input and Displaying Results

From Code::Blocks

Contents

Tutorial 9. Keyboard Input and Displaying Results

In programs for scientific calculations, a standard sort of interaction between the user and the program is that the user gives a command, the program executes it and displays the results, and then the user gives another command, and so on until the user gives the quit command. For example, the user instructs the program to perform a certain regression; the program performs it and displays the results. Such an interface was easy to write in days gone by when everyone used what is now called a terminal window or DOS window. Writing it for a GUI program, however, is not so simple. The old programs used *printf()* or *std::out* statements to display the results. In GUI programs, output from these statements just vanishes. In this tutorial we will see how to write an interface with wxSmith using a Combobox for input from the user and a wxTextCtrl for output. In the process, we will rescue *printf()*, or more correctly, will will write *printg()* which works just like *printf()* except that its output will go to a wxTextCtrl. Even if you are not interested in rescuing *printf()*, you will learn here some useful techniques for initializing global variables, for working with the wxWidgets Event Table, and working with a combobox.

We should first note that wxWidgets provides a command, *wxStreamToTextRedirector*, which will send output such as this

```
cout << setw(18) << left << name << setw(20) << left << surname << left << setw(15) << id << endl;
```

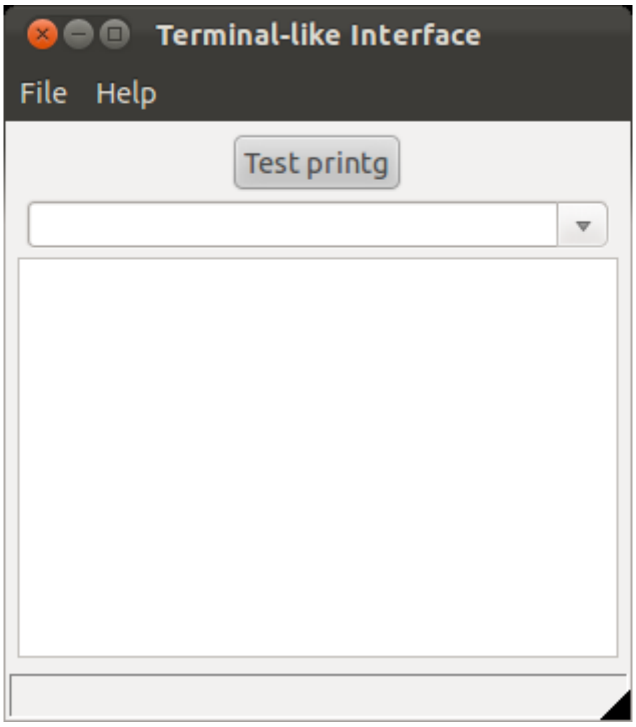
to a TextCtrl. But for those who fail to see the superiority of this way of formatting to the old-fashioned

```
printf("%18s %20s %15d \n", name, surname, id);
```

or for those who have thousands of lines of perfectly good code sprinkled with *printf()* statements, wxWidgets offers no simple device for using that code. We will develop one. All that you then have to do to use it is to globally change *printf(* to *printg(* with your code editor.

Rescuing printf(): Variable argument Lists, String conversions, and Appending lines to a wxTextCtrl

Open Code::Blocks and create a new project called Tutorial_9. We want a form that looks like the one shown below.



By now, you can probably create such a form in your sleep, but to be sure we are together, in the form's property browser you give it the Title property “Terminal-like Interface”, then you put a box sizer on the form, and in the sizer put a panel and check the panel's Expand property. On the panel put a box sizer and in its properties make it vertical. Into it drop successively a button, a Combo box, and a TextCtrl. Give the button the label “Test printg” and make is Proportion property 0. Mark the Expand property of the Combo Box and make its Proportion property 0. On the TextCtrl, 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. Give the Combo box the name CmdBox (for Command Box) and give the TextCtrl the name Results.

Now we go into new territory. Open the file Tutorial_9Main.h (Use File | Open ... to do so.) and add printg as a friend of the class, as shown here.

Tutorial_9Main.h

```
    public:

        Tutorial_9Frame(wxWindow* parent,wxWindowID id = -1);
        virtual ~Tutorial_9Frame();
        friend void printg(char *fmt, ...);
    private:
```

You will notice that we have made printg() a friend but not member of the interface class. That is because we want to declare printg() globally and to be able to call it from anywhere in a perhaps large program, not just from within the class which handles the user interface. This decision will cost us a bit of work at first, but will be well worth it in the long run.

Save and close that file. Double click on “Test printg” button. The file Tutorial_9Main.cpp opens and you find the frame for OnButton1Click and fill it in as follows:

Tutorial_9Main.cpp

```
void Tutorial_9Frame::OnButton1Click(wxCommandEvent& event)
{
    printg("%s", "O brave new world!\n");
}
```

You recognize the call to printg() as just like a printf() call. Just below it, you can now put this definition of the character arrays *printout* and *inbuf* and the function *printg()*.

Tutorial_9Main.cpp

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

void Tutorial_9Frame::OnButton1Click(wxCommandEvent& event)
{
    printg("%s","O brave new world!\n");
}
#define MAXPRINTOUT 500
#define MAXLINE 500
#define OK 1
#define ERR -1

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

void printg(char *fmt, ...){
    char buf[240];
    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 >= MAXPRINTOUT-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 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(). In a real application, you should do so.)

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

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

What is MainFrame? Remember that printg() is being defined globally, not as a member of of the class which contains Results. So MainFrame is the pointer to this class. How does the program know what MainFrame is? Up just below the initial #include commands, we need to put the line

```
extern Tutorial_9Frame* MainFrame;
```

That tells routines in this file that MainFrame will be a pointer to the Tutorial_9Frame class. It will be given a value somewhere else before these routines are called. Where?

Open the file Tutorial_9App.cpp. There you will see this code:

```
IMPLEMENT_APP(wxSmithTuts_Tutorial_9App);
```

```
bool Tutorial_9App::OnInit()
{
    //(*AppInitialize
    bool wxsOK = true;
    wxInitAllImageHandlers();
    if ( wxsOK )
    {
        Tutorial_9Frame* Frame = new wxSmithTuts_Tutorial_9Frame(0);
        Frame->Show();
        SetTopWindow(Frame);
    }
    //*)
    return wxsOK;
}
```

As you can see, this code gets executed before our form gets created, or more precisely, this is what creates the form. That local variable **Frame** is the pointer to the interface class of which Results is a member. We have to grab it and put it somewhere where other routines, such as our printg() can find it. That is, we need to declare it globally. In the process, let's give it a more distinctive name than just Frame. Let's call it MainFrame. So we edit this code to make it read like this.

```
IMPLEMENT_APP(wxSmithTuts_Tutorial_9App);

Tutorial_9Frame* MainFrame;

bool wxSmithTuts_Tutorial_9App::OnInit()
{
    //(*AppInitialize
    bool wxsOK = true;
    wxInitAllImageHandlers();
    if ( wxsOK )
    {
        MainFrame = new Tutorial_9Frame(0);
        MainFrame->Show();
        SetTopWindow(MainFrame);
    }
    //*)
    return wxsOK;
}
```

Finally, we set *printout*'s 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 (*The Tempest*, Act 5, Scene 1) 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”? 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 Tutorial_9Main.cpp if it is not already open. At the bottom, put the lines

Tutorial_9Main.cpp

```
void Tutorial_9Frame::OnCreate(wxWindowCreateEvent& event)
{
    printout[0] = '\0';
}
```

The window we are concerned with is precisely Tutorial_9Fame. 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 Tutorial_9Main.h and add several lines. First, around line 39 or 40 you should find the first two of the following three lines.

Tutorial_9Main.h

```
        void OnAbout(wxCommandEvent& event);
        void OnButton1Click(wxCommandEvent& event);
        //*)
        void OnCreate(wxWindowCreateEvent& event);
```

You add the third. You will notice that you adding a member function to the class Tutorial_9Frame. 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 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.

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

```
DECLARE_EVENT_TABLE()
```

somewhere – anywhere, it seems – among the members in the class definition in the Tutorial_9Main.h file (newer 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

Tutorial_9Main.h

```
class Tutorial_9Frame: 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.

The Event Table itself, however, is back in the Tutorial_9Main.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:

Tutorial_9Main.cpp

```
BEGIN_EVENT_TABLE(Tutorial_9Frame,wxFrame)
    //(*EventTable(Tutorial_9Frame)
    //*)
    EVT_WINDOW_CREATE(Tutorial_9Frame::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.

Now compile and run and rejoice with Miranda.

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, and 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. Double-click the EVT_TEXT_ENTER event. 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..

Tutorial_9Main.cpp

```
void Tutorial_9Frame::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(); //We will be using this later.
}
```

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), 239);
```

For the moment, we will comment out the call to “anything()”, so that we can compile, run and test the program at this point. Try it!

You may notice that nothing happens when you press Enter in CmdBox. That is because we have to set CmdBox to handle that event. Select this control in

Code::Blocks Resources window, then in Properties open Style section and check wxTE_PROCESS_ENTER. Now it's working as expected.

The 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 to calculate a comet's orbit, or to do anything computers can do. 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 *printg()* has to be a member of the class because it uses *Results* – so *anything()* has to be a member of the class.

We will write anything() as simply

Tutorial_9Main.cpp

```
void Tutorial_9Frame::anything()
{
    printg("Anything says: %s\n", inbuf);
}
```

so when we type something in the combobox and tap Enter, we should see what we wrote appear twice on Results, once as before and once preceded by the words "Anything says:" But we must not forget to put *anything()* into the definition of the class. So over in the Tutorial_9Main.h file we need to add a line as indicated here:

Tutorial_9Main.h

```
    public:

        Tutorial_9Frame(wxWindow* parent,wxWindowID id = -1);
        virtual ~Tutorial_9Frame();
        void printg(char *fmt, ...);
        void anything();

    private:
```

Now build and run. You should find our program a perfect parrot; whatever you say to it, it says back to you.

Maybe that seems like not much, but remember that *anything()* has infinite potential.

Previous | Index

Tutorial 10. Using wxGrid