

# Chapter 5

## Graphical programming using *wxWidgets*

C++ is a horrible language.  
It's made more horrible  
by the fact that a lot  
of substandard programmers use it.

Linus Torvalds <sup>a</sup>

### 5.1 Graphic objects. Operating by events

*WxDev-C++* uses wide and flexible library of typical objects. Data determines the object properties. For example, in a graphic object data determines size, position, colour, inscriptions, if it is visible or not and other properties. Some data about the object could be changed by users. In order to change the data we use *properties*. Object functions are called *methods*. Various operations can be performed with objects properties using methods. For example, form object has methods which make it possible to hide and show it, change the size, renew etc. Most *wxDev-C++* objects conform to the Windows graphic objects: windows, buttons and others.

We will not create any new class, but only apply typical objects for our requirements by changing their properties. We will change them while preparing and designing a program and also while executing it. That is why our objects during the performances could change the colour, disappear, move and etc.

WxDev-C++ graphic objects are forms and controls. Forms are windows where controls: labels, buttons, text fields, etc. are placed. As we will see there can be invisible controls. The program could have a lot of forms part of which could be invisible during the operating time.

The object use syntax similar to structure syntax. Objects and methods are separated using -> signs.

---

<sup>a</sup>Linus Benedict Torvalds (born 1969) - a Finnish software engineer and hacker, who was the principal force behind the development of the *Linux* kernel. He later became the chief architect of the *Linux* kernel, and now acts as the project coordinator.

**Operating by events.** The action of a graphical program is supported by *events*. This is opposite to the console program operating where the execution starts with the first line then comes different loops, branches, the results are printed and the work of the program is finished. Programmer in this case is able to predict the work of the system, considering himself the master of the situation.

Operating by events the program execution depends on the user's actions and the state of the program. For example, working with the text editor, users action order and the work of the program can be very different, and hard to be predicted. Events are created not only by the user but also by the operating system, by the program itself and even by some separate program elements.

The alive nature also works according to this event oriented paradigm. While going somewhere we do not plan how many steps we are going to do forward, how many after turning right and if on our way we face an event "a closed door", we will execute the function open, if we come across an obstacle, we will overcome it.

An event is caused both by a user (for example, clicking a mouse or using a key on a keyboard) and a program (for example, if there is a mistake or if there is a program time signal). The objects (forms and regulators) reacting to the event implement some part of the program, called the event function. It is written by a programmer and the events recognition is made automatically, because the C++ system is responsible for it. If there is no event to which the object knows how to react, the program will wait for the following event. For example, when we click the mouse while the cursor is on the button with the inscription "Finish the program", the button click event appears. Such event function will have one line which will finish the program execution.

Operating by events the objects become like independent figures to which are determined how to react to this or that event as presented in Fig. 5.1. So the whole performance script stays very difficult for prediction. Because of great amount of objects, many different kinds of events and a great variety of users actions the traditional program creation becomes very complicated. Meanwhile, the use of events makes complex software product development much easier. The program code is divided into several separate event function creation.

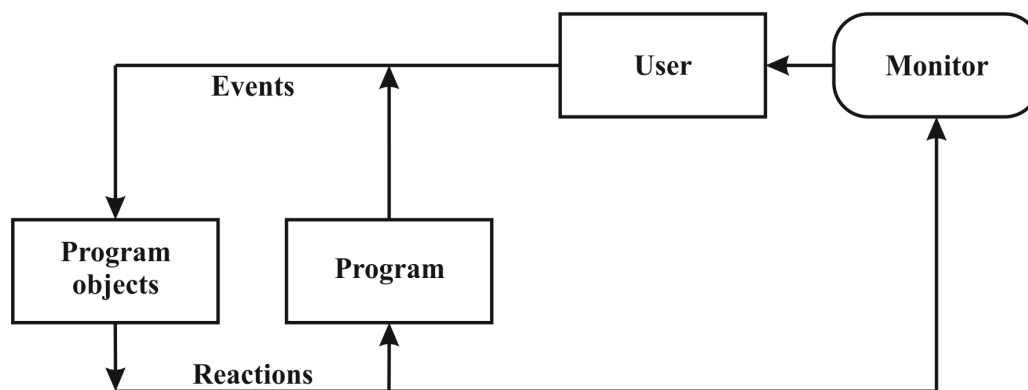


Fig. 5.1. The scheme of operating by events.

## 5.2 Creation of graphical project

We use the set of commands *File\New\Project\* to create a new project in *wxDev-C++*. We can use *File\Open Project* or *File\* to open an existing project. While creating a new project the dialogue window appears, where one must choose the type of the project and give it the name. We will use *wxWidgets* projects to create graphical programs. *WxDev-C++* contains three *wxWidgets* (GUI) project types:

- i) empty *wxWidgets* project a project that initially contains no source files but has the correct compiler and linker settings to build *wxWidgets* source code.
- ii) *wxWidgets Dialog* a project that contains skeleton source files for a *wxDialog*.
- iii) *wxWidgets Frame* a project that contains skeleton source files for a *wxFrame*.

We will use the second type of project in most of our programs. Some programs with *MDI* interface will be created using the third project type.

Selecting the second and the third project types, the dialogue window appears (as presented in Fig. 5.2). It is possible to set file and class names, project saving path, title, window style and other parameters in this window.

The view of the environment changes a bit using *wxWidgets* projects. Let us examine the parts of this IDE environment in more details (Fig. 5.3).

In the centre of the screen one can see the window named *Project1*. This is the form designing window where the regulators will be placed during the design process.

The form designer is the acting field for the design. Buttons and labels, pictures and other things can be situated on the form. If we create a complex program there is no need to limit ourselves by the only one form (Fig. 5.4).

The form by itself is one of the standard *wxDev-C++* objects. The buttons, scroll bars, drawings that will be situated in the form are also standard. So everything is made from standard objects. Don't you think that the program created with the *wxDev-C++* tools will be monotonous, as if it is a block of flats?

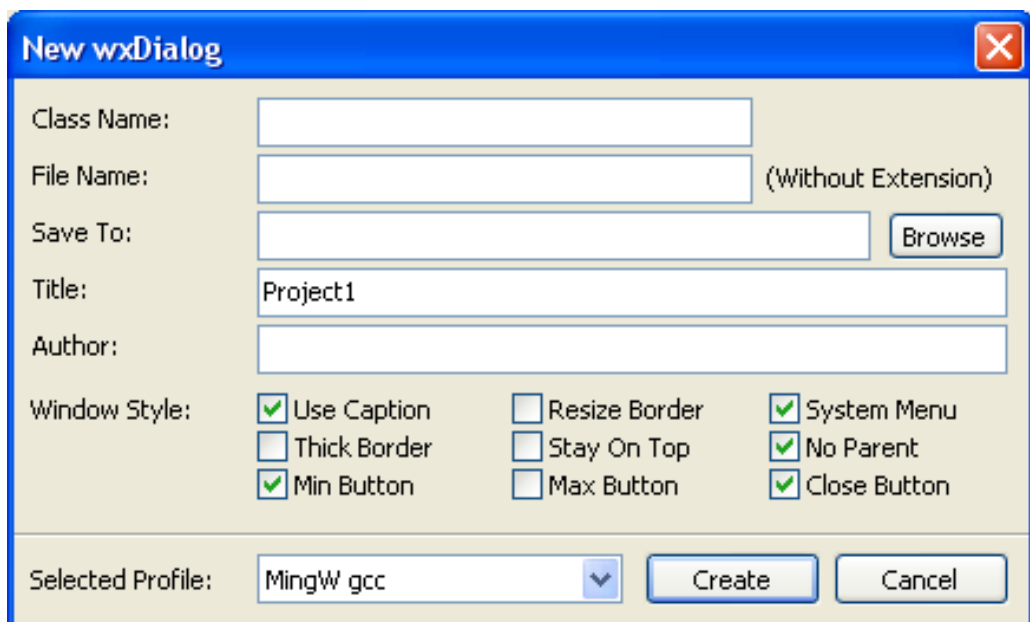


Fig. 5.2. Project *wxWidgets*: properties.

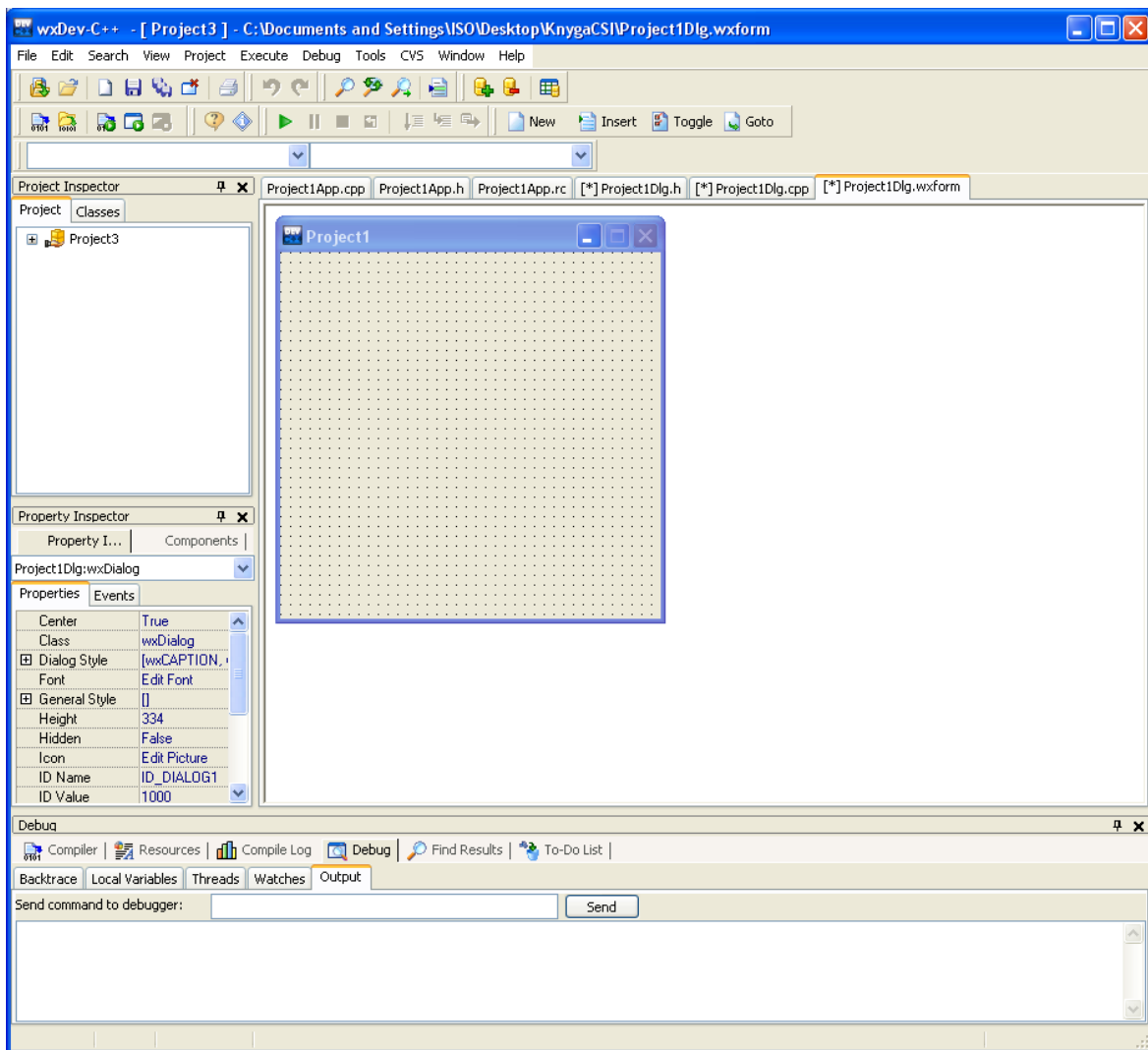


Fig. 5.3. The general view of the *wxWidgets**wxDev-C++* IDE environment.

This might happen if we didn't have a great possibility to change objects. Objects have a lot of changeable properties and it makes possible to create different programs and design graphic images of many various shapes.

We will use two concepts: *design* and *run*. While preparing a *wxDev-C++* program a design takes place when the program design is implemented, the property values are set and the event functions are written. When the program is started the project runs. These two stages sometimes are mixed because during the design the internal view of the program is almost the same as if it runs.

Working with *wxWidgets* project, *Property inspector* and *Components* panels are activated. The pictograms of the objects that can be used are seen in the *Components* toolbox which is usually on the right side of the screen (Fig. 5.5).

It is very easy to "take" an object from the toolbox and to "place" it into a necessary place in the form. It is enough to click the mouse on a desired object and then place it where you need by clicking the mouse on selected place. It is also easy to change the size of the object using the mouse. It is obvious that created objects can be copied, pasted and even deleted

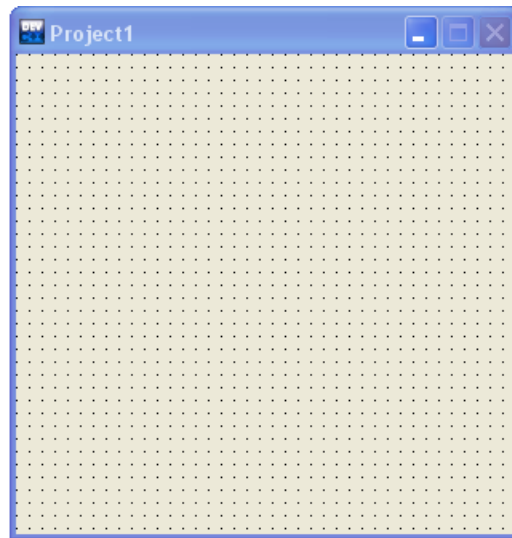


Fig. 5.4. Form design.

using the common operations of the *Windows* environment.

The example of the simplest object which pictogram *A* is in the toolbox is a static text. It is used exactly for the text writing.

The properties of the chosen object, in our case *wxBUTTON1*, are named in the window of *Property Inspector* (Fig. 5.6). Any property during the design period could be changed. All the objects have the property *Name*, most of the objects have the property *Label* or *Text*.

If it is possible to use only the fixed values of property, it is very convenient to choose the value from the list which appears by clicking with the mouse on a small button with a triangle at the right side of the editing line.

For the choice of the values of some properties it is possible to use the dialogue, for example, the dialogue of the choice of the colour, dialogue of the choice of font and its properties etc. Dialogue window is opened by clicking on appropriate button with the three-point.

Each object has its own list of properties. Window of *Project Inspector* shows the program modules and classes.

The program window appears only when we start to write event functions. For example, if we open the *Property Inspector's Events* area we can pick an event and choose *Add New Function*. We will see a program code window with our new function prepared. If we want, we can write a text in this function layout.

The most often used events are: *Click*, *Paint*, *MouseMotion*, *KeyDown*, *KeyUp*, etc.

The object may have a lot of events and also event processing functions. The list of all the regulator events can be seen in the *Event* area of *Property Inspector*. We can see that the form, for example, reacts to the movement of the mouse (*MouseMotion*) and other events.

There are a lot of different windows that are used in the designing process and not all of them are used at the same time. Almost all the time is used the form design window, often the *Property Inspector* and *Components window*. It is necessary to learn how to manipulate the windows and organize them in a convenient order for the use. If one has lost some window, it is necessary to appeal to the menu command *View\Show Panels* or *View\Show Plugin*.

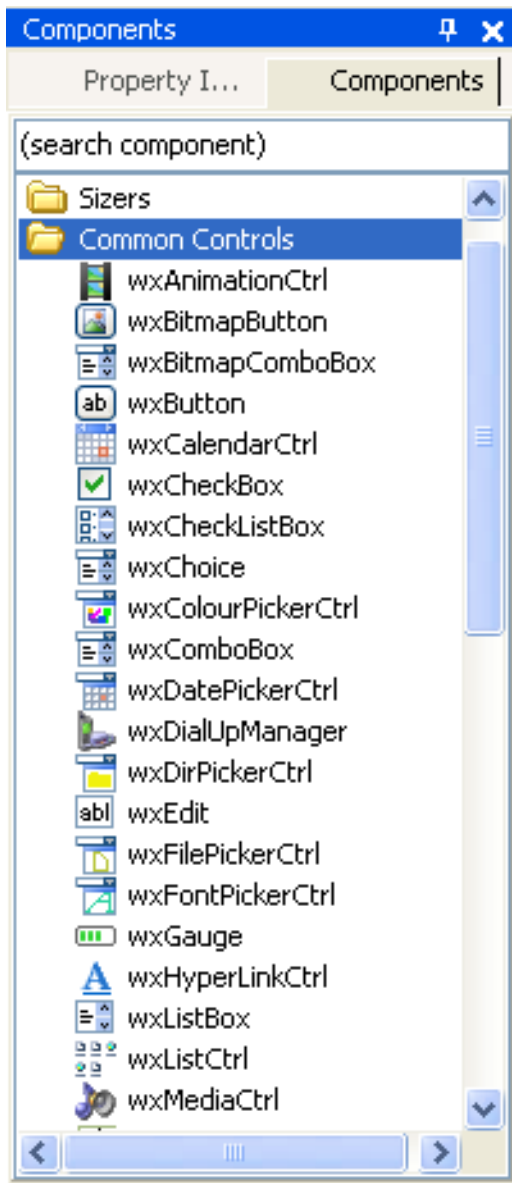


Fig. 5.5. Components toolbox.

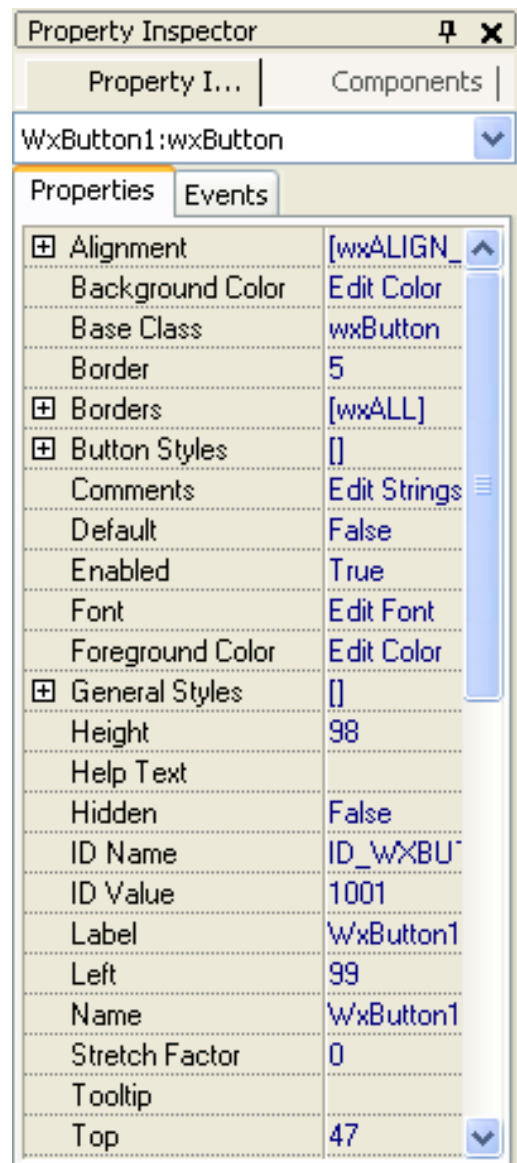


Fig. 5.6. Property Inspector.

### 5.3 Creation of the first program

The creation of the program could be divided into three main steps:

- i) creation of the user interface;
- ii) setting of the properties;
- iii) writing of the program text.

But this does not mean that in practice these steps or their fulfilment is always followed strictly. For example, it is possible to return back to the interface, to the setting of the properties because some new ideas could appear when the program is tested.

Table 5.1. The changes of the object properties.

| Object             | Property | Value of property |
|--------------------|----------|-------------------|
| <i>Project1Dlg</i> | Title    | My first program  |
| <i>WxEdit1</i>     | Text     | Empty             |
| <i>WxButton1</i>   | Label    | Click me          |

We will illustrate the steps of the creation of a program with some simple examples. At the beginning in the program window should be an empty text box (*wxEdit*) and a button (*wxButton*). If we click the button, "Hello world!!!" will appear in the text box.

First of all we will place our objects in the form using the *Component box*. Open *Common Controls* folder and click an appropriate pictogram in the *Component box*, then click the mouse on the wished place and change the object size using the mouse. The image of project interface is presented in Fig. 5.7.

The properties will be changed with the use of the *Property Inspector*. Let us make changes of the values of the objects properties presented in Table 5.1.

We write the text of the program in the program text window which is opened when we chose *Add New Function* in the *Event* area of *Property Inspector*. Such event function runs only when the particular event connected to the object happens. In our case it is event *Click*:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxEdit1 -> SetValue("Hello world!!!");
}
```

The program changes the text field *WxEdit1* property *Text* value from empty into "Hello world!!!". The value is changed bay using *SetValue* method. Fig. 5.8 illustrates the ouput of program, after the *Click me* button has been pressed.

If we want to save our project, we should not forget that it consists not of one file only. So the best way to save the projects is to create a separate catalogue for each particular project.

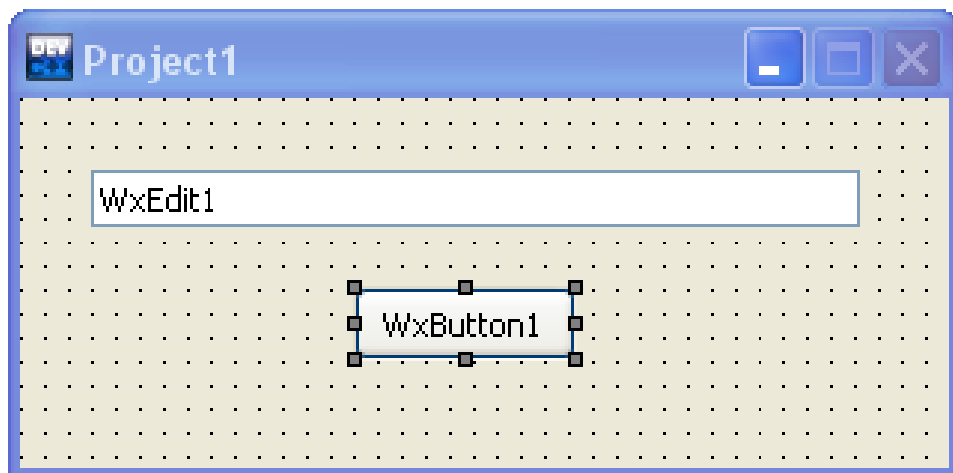


Fig. 5.7. The project interface.

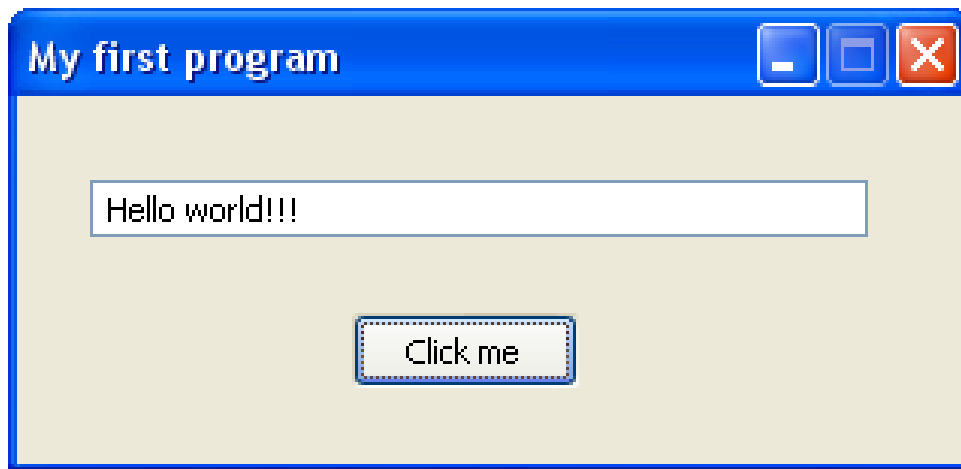


Fig. 5.8. Output of program: after button click.

Fig. 5.9a. Pictogram of a button among the *Common Controls* image of the button in the program.

Fig. 5.9b. The image of the textbox and static text.

## 5.4 Basic control. Input/output of text

We can find the button among the *Common Controls* in the *Components box*. The pictogram of a button and its image in the program are presented in Fig. 5.9a.

The click of the button causes the event *Click*. That is why the part of the program we want to run with the click of this button we will write into the button *Click* event function. Very often the button is used to finish the program. That button *Click* event function will be the following:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    exit(0);
}
```

In graphical programs text boxes are usually used for text input, they are also used for the text output. But otherwise for text output there is one more rather simple and convenient regulator *Static Text*. Pictograms of these regulators are presented in Fig.5.9b.

The text of the *Static Text* regulator (the property *Label*) is set mostly during the design time. This is one of the ways to output text to the form.

For example, the font of the *Static Text* label "Jonas" (italic or normal) could be changed with the buttons during the running time.

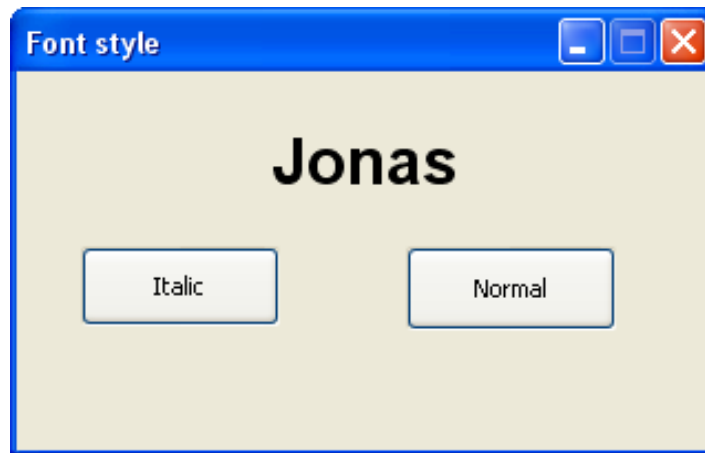


Fig. 5.10. Changing of the font.

Figure below presents the image of the form in which we can see the *WxStaticText1* with the inscription "Jonas" (the property *Label*) and two buttons *WxButton1* and *WxButton2* with the inscriptions "Italic" and "Normal" (see Fig. 5.10).

It is necessary to write the *Click* event function for the buttons (their first and last lines are always presented). The *SetFont* method is used to change the font of *StaticText*:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxStaticText1 -> SetFont(wxFont(22, wxSWISS, wxITALIC, wxBOLD, false));
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    WxStaticText1 -> SetFont(wxFont(22, wxSWISS, wxNORMAL, wxBOLD, false));
}
```

A font is an object which determines the appearance of the text, primarily when drawing text to a window or device context. A font is determined by the following parameters (not all of them have to be specified, of course):

|                    |                                                                                                                                                                            |
|--------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Point size</i>  | standard way of referring to text size;                                                                                                                                    |
| <i>Family</i>      | supported families are: <code>wxDEFAULT</code> , <code>wxDECORATIVE</code> , <code>wxROMAN</code> , <code>wxSCRIPT</code> , <code>wxSWISS</code> , <code>wxMODERN</code> ; |
| <i>Style</i>       | value can be <code>wxNORMAL</code> , <code>wxSLANT</code> or <code>wxITALIC</code> ;                                                                                       |
| <i>Weight</i>      | value can be <code>wxNORMAL</code> , <code>wxLIGHT</code> or <code>wxBOLD</code> ;                                                                                         |
| <i>Underlining</i> | value can be <code>TRUE</code> or <code>FALSE</code> ;                                                                                                                     |
| <i>Face name</i>   | an optional string specifying the actual typeface to be used. If <code>NULL</code> , a default typeface will chosen based on the family;                                   |
| <i>Encoding</i>    | font encoding.                                                                                                                                                             |

Specifying a family, rather than a specific typeface name, ensures a degree of portability across platforms because a suitable font will be chosen for the given font family.

The text boxes are usually used for the text input, but also they can be used for the output of it. The most important thing is their property *Text* in which we can find the entered text. The property *Edit Style* `wxTE_MULTILINE` makes it possible to create a text box that consists of a lot of lines (if the value of the `wxTE_MULTILINE` is equal to `TRUE`). The property `MaxLength` specifies the maximum amount of symbols that can be entered into text box.

To enter the password, we shall use the text box property *Edit Style* - `wxTE_PASSWORD`. We use the event *Updated*, when some changes happen in the text.

**Text box methods.** Some text box methods are presented below.

`AppendText(const wxString& text)` - appends the text to the end of the text box. The parameter `text` specifies the text to write to the text control. After the text is appended, the insertion point will be at the end of the text control. If this behaviour is not desired, the programmer should use `GetInsertionPoint()` and `SetInsertionPoint()`.

`Clear()` - clears the text in the control. Note, that this function will generate a text updated event.

`Copy()` - copies the selected text to the clipboard.

`Cut()` - copies the selected text to the clipboard and removes the selection.

`GetInsertionPoint()` - returns the insertion point. This is defined as the zero based index of the character position to the right of the insertion point. For example, if the insertion point is at the end of the text control, it is equal to both `GetValue().Length()` and `GetLastPosition()`.

`GetLastPosition()` - returns the zero based index of the last position in the text control, which is equal to the number of characters in the control.

`GetLineLength(long lineNo)` - gets the length of the specified line, not including any trailing newline character(s). The parameter `lineNo` is the line number (starting from zero). The length of this line is returned, or (-1) if `lineNo` was invalid.

`GetLineText(long lineNo)` - returns the content of a given line in the text control, not including any trailing newline character(s). The parameter `lineNo` is the line number (starting from zero). The content of the specified line is returned.

`GetStringSelection()` - gets the text currently selected in the control. If there is no selection, the returned string is empty.

`GetValue()` - gets the contents of the control. Note, that for a multiline text control, the lines will be separated by (*UNIX*-style) `\n` characters, even under *Windows* where they are separated by a `\r\n` sequence in the native control.

`IsEditable()` - returns `TRUE` if the controls contents may be edited by user (note that it always can be changed by the program), i.e. if the control has not been put in read-only mode by a previous call to `SetEditable`.

`IsEmpty()` - returns `TRUE` if the control is currently empty. This is the same as `GetValue().empty()` but can be much more efficient for the multiline controls containing big amounts of text.

`IsMultiLine()` - returns `TRUE` if this is a multi line edit control and `FALSE` otherwise.

`IsSingleLine()` - returns `TRUE` if this is a single line edit control and `FALSE` otherwise.

`LoadFile(const wxString& filename, int fileType=wxTEXT_TYPE_ANY)` - loads and displays the named file, if it exists. The parameter `filename` specifies the filename of the file to load. The parameter `fileType` specifies the type of file to load.

`Paste()` - pastes text from the clipboard to the text item.

`Remove(long from, long to)` - removes the text starting at the first given position (parameter `from`) up to (but not including) the character at the last position (parameter `to`).

`Replace(long from, long to, const wxString& value)` - replaces the text starting at the first position (parameter `from`) up to (but not including) the character at the last position (parameter `to`) with the given text (parameter `value`).

`SaveFile(const wxString& filename, int fileType=wxTEXT_TYPE_ANY)` - saves the contents of the control in a text file specified by the `filename` parameter. The parameter `fileType` specifies the type of file to save.

`SetEditable(const bool editable)` - makes the text item editable or read-only (Boolean parameter `editable`). If the parameter value is `TRUE`, the control is editable. If `FALSE`, the control is read-only.

`SetInsertionPoint(long pos)` - sets the insertion point at the given position (parameter `pos`).

`SetInsertionPointEnd()` - sets the insertion point at the end of the text control. This is equivalent to `SetInsertionPoint(GetLastPosition())`.

`SetMaxLength(unsigned long len)` - this function sets the maximum number of characters the user can enter into the control. In other words, it allows to limit the text value length to `len` not counting the terminating `NULL` character. If `len` is equal to 0, the previously set maximal length limit, if any, is discarded and the user may enter as much text as the underlying native text control widget supports (typically at least 32Kb). If the user tries to enter more characters into the text control when it already is filled up to the maximal length, a `wxEVT_COMMAND_TEXT_MAXLEN` event is sent to notify the program about it (giving it the possibility to show an explanatory message, for example) and the extra input is discarded.

`SetSelection(long from, long to)` - selects the text starting at the first position (parameter `from`) up to (but not including) the character at the last position (parameter `to`). If both parameters are equal to (-1) all text in the control is selected.

`SetValue(const wxString& value)` - sets the text value and marks the control as not-modified (which means that `IsModified` would return `FALSE` immediately after the call to `SetValue`).

`ChangeValue(const wxString& value)` - sets the text value and marks the control as not-modified (which means that `IsModified` would return `FALSE` immediately after the call to `SetValue`). This function is new since *wxWidgets* version 2.7.1.

`ShowPosition(long pos)` - makes the line containing the given position (parameter `pos`) visible.

`WriteText(const wxString& text)` - writes the text (parameter `text`) into the text control at the current insertion position. After the write operation, the insertion point will be at the end of the inserted text, so subsequent write operations will be appended. To append text after the user may have interacted with the control, call `wxTextCtrl::SetInsertionPointEnd` before writing.

For example, let us create the input field that will not allow starting the text with any wrong word.

```
// WxEdit1Updated
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    if (WxEdit1 -> GetValue() == "bad")
        WxEdit1 -> SetValue ("This is a wrong word!");
}
```

For example, let us create a simple calculator which multiplies two numbers and shows the result as soon as the data is entered. Implementing this task, we shall create two text boxes *WxEdit1* and *WxEdit2* for the variables and the *StaticText* for the results. Than we shall write the following program code:

```
// WxEdit1Updated
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    wxString answer;
    answer << wxAtof (WxEdit1 -> GetValue()) *
        wxAtof (WxEdit2 -> GetValue());
    WxStaticText1 -> SetLabel(answer);
}
```

The same we shall repeat for the *WxEdit2*. Note, that we enter text data into text boxes, so we must convert the text into number (*wxAtof()* function), and then we must convert the result back to the string format.

## 5.5 Properties

### 5.5.1 The geometrical properties

Every *wxDev-C++* object contains a great list of properties: most properties in the objects are analogical. Almost all objects have their position and size. These properties are called the geometrical properties. The position coordinates are determined by the properties: *Left* in horizontal axis and *Top* in vertical axis. The size is determined by the properties: *Width* and *Height* (see Fig. 5.11).

The initial values of the geometrical properties are determined automatically during the designing time changing with the mouse the size and the position of the objects. During the program working time it is possible to move regulators, change their size by changing these properties.

The coordinates system begins form the upper left corner of the form or other container. It is, so called, a container coordinates system, which does not change, if it is moved, because the container and its regulators move together. The form can enclose other containers in which there can be their own regulators. The coordinates system beginning of those regulators is the upper left corner of their container.

The unit of measurement of geometrical properties is usually a pixel.

The example. There are four buttons and a static text in the form. The position of the static text is changed by the buttons (Fig. 5.12).

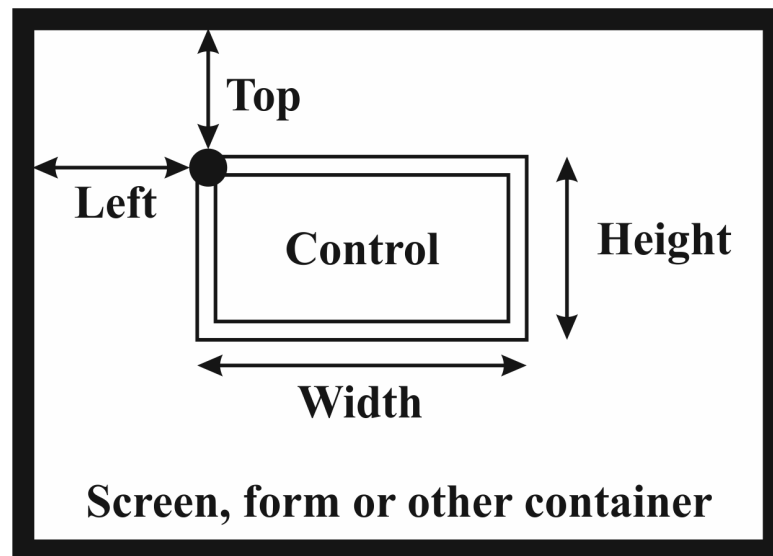


Fig. 5.11. The object properties that determine the position and size.

The property of the position of the label consists of two coordinates. During the execution of the program it is possible to get the values of control position (`GetPosition()` method) and it is also possible to set the new value of the coordinates (`Move()` method). So, let us write the following functions for the buttons:

```
void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x,y-5);
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x,y+5);
}

void Project1Dlg::WxButton3Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x-5,y);
}

void Project1Dlg::WxButton4Click(wxCommandEvent& event)
{
    int x, y;    WxStaticText1 -> GetPosition(&x,&y);
                WxStaticText1 -> Move(x+5,y);
}
```

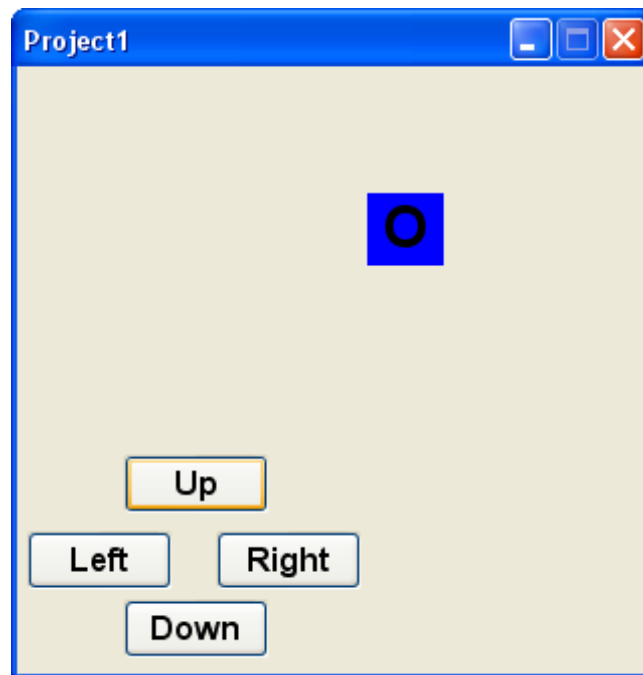


Fig. 5.12. Moving object program.

### 5.5.2 The properties of colours

Most of the objects have colour properties: *Background Color* as the colour of the background; *Foreground Color* as the colour of the text and drawings. It is easy to set the property of colour during the designing stage because we set it up in the special window (see Fig. 5.13). There are five possible ways to set the colour:

- i) it is possible to use default colour;
- ii) we can pick the colour from standard colours;
- iii) we can use system colours;
- iv) it is possible to define custom colour (using RGB format);
- v) it is possible to pick a colour from the palette (*Choose button*).

We can use `SetBackgroundColor()` and `SetForegroundColor()` methods to change control's colour properties during program run. The parameter of both methods is `wxColour`.

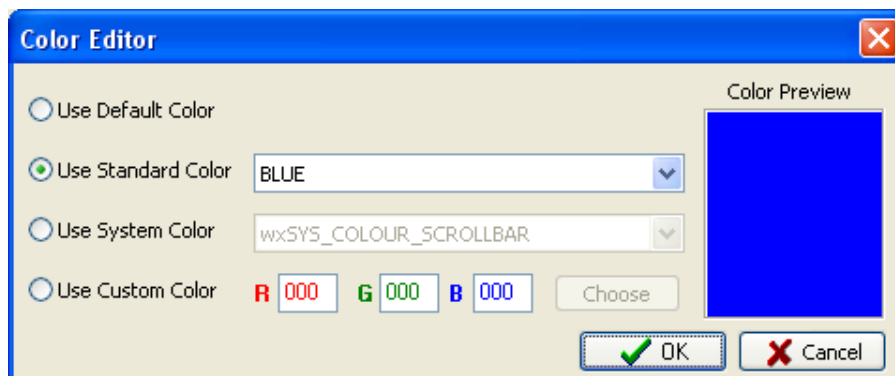


Fig. 5.13. Colour setting.



Fig. 5.14. Colour mixing.

Using RGB colour setting the meanings of the arguments can change from 0 (no colour) till 255 (the most intensive colour) (see Fig. 5.13).

The example. There are three text boxes in the form which corresponds three main colours (RGB). If we change the meaning of the intensity of the colour the static text will colour into a given tint (Fig 5.14).

Let us use the event *Updated* that happens just as we change the value of the text box. We shall change the text into numbers using the already known method `wxAtoi()`.

```
void Project1Dlg::WxEdit1Updated(wxCommandEvent& event)
{
    WxStaticText1->SetBackgroundColour(wxColour(wxAtoi (WxEdit1 -> GetValue()),
                                                wxAtoi (WxEdit2 -> GetValue()),
                                                wxAtoi (WxEdit3 -> GetValue())));
    WxStaticText1->Refresh();
}
```

The same code is copied and for the other text boxes. Note, that it is necessary to refresh the static text to see the changed colour.

### 5.5.3 Some most often used properties

The change of values of the logical property *Hidden* is very often. This property enables the programmer to hide or show the object. Method `Show()` can be used to hide or show the object (belonging on input parameter: `TRUE` show, `FALSE` hide).

The value of the logical property *Enabled* is usually equal to `TRUE`. The meaning `FALSE` is used if we want to make any of the objects out of work, because its work at that moment is not necessary. If the form is out of work, all the regulators are out of work too.

For example, the button that changes font into italic is not necessary, if that font is already in use at the moment.

The property *Font* determines the font style. It is the object itself with its own properties. Its properties:

- a) size (whole numbers);
- b) font family;
- c) font style;
- d) font weight;
- e) logical Underline;
- f) name (*Arial*, *Symbol* and etc.).

During the program design the font properties are determined through the dialogue.

```

wxFont(           int    pointSize,
                  wxFontFamily family,
                  int    style,
                  wxFontWeight weight,
                  const bool underline = false,
const wxString& faceName = "",
                  wxFontEncoding encoding = wxFONTENCODING\_DEFAULT)

```

Lets starts to describe all significant parameters.

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>pointSize</i> | Size in points.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <i>family</i>    | font family, a generic way of referring to fonts without specifying actual facename. One of:<br><b>wxFONTFAMILY_DEFAULT</b> , chooses a default font;<br><b>wxFONTFAMILY_DECORATIVE</b> , a decorative font;<br><b>wxFONTFAMILY_ROMAN</b> , a formal, serif font;<br><b>wxFONTFAMILY_SCRIPT</b> , a handwriting font;<br><b>wxFONTFAMILY_SWISS</b> , a sans-serif font;<br><b>wxFONTFAMILY_MODERN</b> , a fixed pitch font;<br><b>wxFONTFAMILY_TELETYPE</b> , a teletype font;                                                                                                                                                                       |
| <i>style</i>     | one of :<br><b>wxFONTSTYLE_NORMAL</b> ;<br><b>wxFONTSTYLE_SLANT</b> ;<br><b>wxFONTSTYLE_ITALIC</b> ;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <i>weight</i>    | Font weight, sometimes also referred to as font boldness. One of;;<br><b>wxFONTWEIGHT_NORMAL</b> , normal font;<br><b>wxFONTWEIGHT_LIGHT</b> , light font;<br><b>wxFONTWEIGHT_BOLD</b> , old font;                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <i>underline</i> | The value can be <b>TRUE</b> or <b>FALSE</b> ;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <i>faceName</i>  | An optional string specifying the actual typeface to be used. If it is an empty string, a default typeface will be chosen based on the family;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <i>encoding</i>  | An encoding which may be one of:<br><b>wxFONTENCODING_SYSTEM</b> , default system encoding;<br><b>wxFONTENCODING_DEFAULT</b> , default application encoding: this is the encoding set by calls to <b>SetDefaultEncoding</b> and which may be set to, say, KOI8 to create all fonts by default with KOI8 encoding. Initially, the default application encoding is the same as default system encoding;<br><b>wxFONTENCODING_ISO8859_1...15</b> , ISO8859 encodings;<br><b>wxFONTENCODING_KOI8</b> , standard Russian encoding for Internet;<br><b>wxFONTENCODING_CP1250...1252</b> , <i>Windows</i> encodings similar to ISO8859 (but not identical). |

While starting the program the first form is loaded and shown and the function **CreateGUIControls()** is run.

At the beginning the values of the numerical variables are usually equal to zero that is why it is necessary to describe the determination of the starting meanings in the `CreateGUIControls()` function.

The local variable values are accessed only in their own function. They are also declared in that function.

The class level variable values are accessed in all the functions of this class.

For example. Let us create an event function, that will proportionally change the position of the regulators in the form (for example, the position of the button in the form) while changing the size of the form (see Fig. 5.15).

Implementing this program, we should create two global variables to store the relative position of the button. We can do it in class description (*Project1Dlg.h* file). Two public real variables (`rx` and `ry`) are declared.

```
class Project1Dlg : public wxDialog
{
private:
    DECLARE_EVENT_TABLE();
public:
    Project1Dlg(        wxWindow    *parent,
                      wxWindowID    id    = 1,
                      const wxString &title = wxT("Project1"),
                      const wxPoint  &pos  = wxDefaultPosition,
                      const wxSize    &size = wxDefaultSize,
                      long            style = Project1Dlg_STYLE);
    virtual ~Project1Dlg();
    float rx, ry;
    void Project1DlgSize(wxSizeEvent& event);
    ...
}
```

Then, at the beginning of the program (`CreateGUIControls()` function in *Project1Dlg.cpp* file) it is necessary to calculate the relative position of the button in the form and to remember it. After the change of the size of the form (form event *Size*) recalculate the coordinates of the position of the button (Table 5.2).

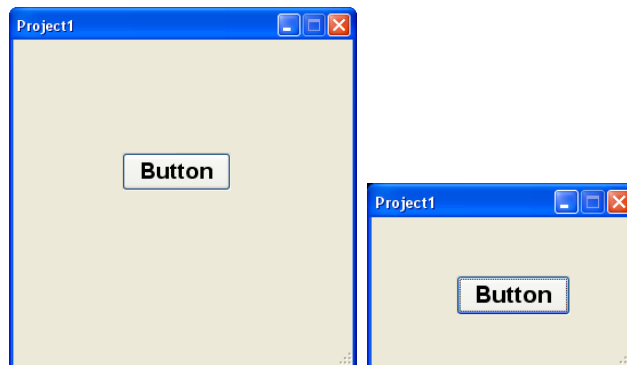


Fig. 5.15. Relative position.

---

Table 5.2. Relative position.

---

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    // //GUI Items Creation Start

    WxButton1 = new wxButton(this, ID_WXBUTTON1, _("Button"),
                             wxPoint(100, 130),
                             wxSize(100, 35), 0,
                             wxDefaultValidator, _("WxButton1"));
    WxButton1 -> SetFont(wxFont(16, wxSWISS, wxNORMAL, wxBOLD, false));

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End
    int x1, y1, x2, y2;
    GetSize(&x1, &y1); // The size of the form
    WxButton1 -> Move(100, 130); // Button position
    WxButton1 -> GetPosition(&x2, &y2); // Position values
    rx = (float) x2/x1; // Relative horizontal position
    ry = (float) y2/(y1-55); // Relative vertical position
                                // 55 - height of form header
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

void Project1Dlg::Project1DlgSize(wxSizeEvent& event)
{
    int x, y;
    GetSize(&x, &y); // New size of the form
    WxButton1 -> Move((int)(rx*x), (int)(ry*y)); // New position of button
}

```

---

## 5.6 Timer and scrollbars

Timer control is invisible during program execution, but it enables program to measure time and to perform different actions connected with time intervals.

Fig. 5.16. *wxTimer* view.

In *wxDev-C++* we have **wxTimer** function (Components folder) (Fig. 5.16). The timer component has no geometrical properties. The main purpose of this control is to execute code at specified intervals (*Timer* events are initiated). Time interval is set using *Interval* property. The timer can be enabled or disabled using methods **Start()** and **Stop()**.

For example, let us create a form, which slowly moves to the right. We will need timer to implement this program (*Interval* value 100, *AutoStart()* value **TRUE**).

---

Table 5.3a. Blinking rectangle.


---

```
void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    // //GUI Items Creation Start

    WxStaticText1 = new wxStaticText(this,
                                     ID_WXSTATICTEXT1, _("WxStaticText1"),
                                     wxPoint(105, 190), wxDefaultSize,
                                     0, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,255,0));

    WxTimer4 = new wxTimer(); WxTimer4 -> SetOwner (this, ID_WXTIMER4);
    WxTimer3 = new wxTimer(); WxTimer3 -> SetOwner (this, ID_WXTIMER3);
    WxTimer2 = new wxTimer(); WxTimer2 -> SetOwner (this, ID_WXTIMER2);
    WxTimer1 = new wxTimer(); WxTimer1 -> SetOwner (this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    // //GUI Items Creation End
    WxTimer1 -> Start(2000,wxTIMER_ONE_SHOT);
}
```

---

Table 5.3b. Blinking rectangle.

---

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    WxStaticText1->Hide();          WxTimer2 -> Start(1000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer2Timer(wxTimerEvent& event)
{
    WxStaticText1->Show();          WxTimer3 -> Start(3000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer3Timer(wxTimerEvent& event)
{
    WxStaticText1->Hide();          WxTimer4 -> Start(2000,wxTIMER_ONE_SHOT);
}
void Project1Dlg::WxTimer4Timer(wxTimerEvent& event)
{
    WxStaticText1->Show();          WxTimer1 -> Start(2000,wxTIMER_ONE_SHOT);
}

```

---

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    int x, y;
    GetPosition(&x, &y);
    Move(x+5,y);
}

```

We can use several timers to perform more complex tasks.

For example, let us create a program, where a red rectangle will blink according to some complex rhythm (2 seconds - visible, 1 second - disappeared, 3 seconds - visible, 2 seconds disappeared and then the rhythm is repeated). We will need four timers and a static text. Timer functions are shown in Tables 5.3a and 5.3b.

Start method has two parameters: timer interval and shooting mode:

`wxTIMER_CONTINUOUS` - start a normal, continuously running, timer;  
`wxTIMER_ONE_SHOT` - start a one shot timer.

We can refuse three timers using global variables (Tables 5.4a, 5.4b, 5.4c).

**Timer program.** Let us create a time measuring program (it will measure time between *Start* and *Stop* button presses). We will measure time in milliseconds using `wxTimer` control (Time 1) and using time functions (Time 2), local time value will be also shown in `wxEdit3` (Local time) after button *Show time* click (see Fig. 5.17). Time 2 will be counted using stop watch class. We must define the public `wxStopWatch` object `sw` and `long t` in project class declaration. The program code is presented in Table 5.5.

---

Table 5.4a. Blinking rectangle with one timer.


---

```

class Project1Dlg : public wxDialog
{
    private:
        DECLARE_EVENT_TABLE();

    public:
        Project1Dlg(        wxWindow    *parent,
                           wxWindowID   id = 1,
                           const wxString &title = wxT("Project1"),
                           const wxPoint& pos = wxDefaultPosition,
                           const wxSize& size = wxDefaultSize,
                           long style = Project1Dlg_STYLE);
        virtual ~Project1Dlg();
        int interval, step; // Two public variables are declared
        void Project1DlgSize (    wxSizeEvent& event);
        void WxButton1Click (wxCommandEvent& event);
        void WxTimer1Timer ( wxTimerEvent& event);
        ...
}

```

---



---

Table 5.4b. Blinking rectangle with one timer.


---

```

void Project1Dlg::CreateGUIControls()
{
    WxStaticText1 = new wxStaticText(this,
                                     ID_WXSTATICTEXT1, _("WxStaticText1"),
                                     wxPoint(105, 190), wxDefaultSize,
                                     0, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,255,0));

    WxTimer1 = new wxTimer();
    WxTimer1 -> SetOwner(this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End

    step=0;
    interval=2000;
    WxTimer1->Start(interval,wxTIMER_ONE_SHOT);
}

```

---

---

Table 5.4c. Blinking rectangle with one timer.


---

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    step++;
    if (step>4) step=1;
    switch (step)
    {
        case 1:
            WxStaticText1 -> Hide();    interval=1000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 2:
            WxStaticText1 -> Show();    interval=3000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 3:
            WxStaticText1 -> Hide();    interval=2000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
        case 4:
            WxStaticText1 -> Show();    interval=2000;
            WxTimer1 -> Start(interval,wxTIMER_ONE_SHOT); break;
    }
}

```

---

Timer control and stopwatch time measurements will differ because of timer precision. Its precision is platform-dependent, but, in general, will not be better than 1 ms or worse than 1 s. So, it is better to use `wxStopWatch` object to measure time (especially in milliseconds).

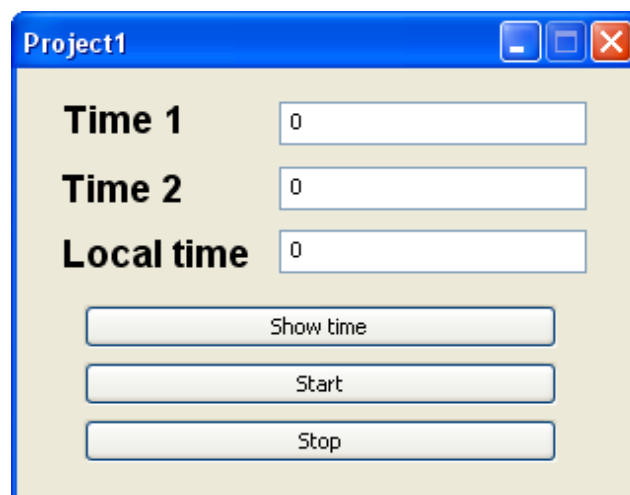


Fig. 5.17. Timer program interface.

| Table 5.5. | Timer program code.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | <pre> void Project1Dlg::WxButton1Click(wxCommandEvent&amp; event) {     WxEdit3-&gt;SetValue(wxNow()); } void Project1Dlg::WxButton2Click(wxCommandEvent&amp; event) {     t=0;     WxTimer1 -&gt; Start(1,wxTIMER_CONTINUOUS); // Timer is started     sw.Start();     WxEdit1 -&gt; SetValue("0");     WxEdit2 -&gt; SetValue("0"); } void Project1Dlg::WxTimer1Timer(wxTimerEvent&amp; event) {     t++; } void Project1Dlg::WxButton3Click(wxCommandEvent&amp; event) {     WxTimer1-&gt;Stop();     sw.Pause();     wxString ss;      ss &lt;&lt; sw.Time();  WxEdit2 -&gt; SetValue(ss);                       ss = ""; ss &lt;&lt; t;    WxEdit1 -&gt; SetValue(ss); } </pre> |

**Moving bars program.** In this program we will create two moving bars (Static text controls). Each bar will move with its own speed and will reflect from form's borders (see Fig. 5.18). Program code is presented in Tables 5.6a, 5.6b.

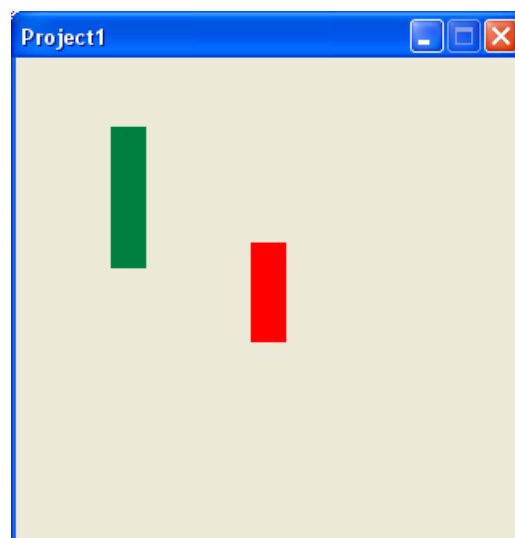


Fig. 5.18. Interface of Moving bars program.

Table 5.6a. Moving bars program code.

---

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    //// GUI Items Creation Start

    WxStaticText2 = new wxStaticText(this, ID_WXSTATICTEXT2, _(""),
                                     wxPoint(146, 115), wxSize(22, 62),
                                     wxST_NO_AUTORESIZE, _("WxStaticText2"));
    WxStaticText2 -> SetBackgroundColour (wxColour(255,0,0));

    WxStaticText1 = new wxStaticText(this, ID_WXSTATICTEXT1, _(""),
                                     wxPoint(59, 43), wxSize(22, 88),
                                     wxST_NO_AUTORESIZE, _("WxStaticText1"));
    WxStaticText1 -> SetBackgroundColour (wxColour(0,128,64));

    WxTimer2 = new wxTimer(); WxTimer2 -> SetOwner(this, ID_WXTIMER2);
    WxTimer1 = new wxTimer(); WxTimer1 -> SetOwner(this, ID_WXTIMER1);

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End

    WxTimer1 -> Start(100, wxTIMER_CONTINUOUS);
    WxTimer2 -> Start(100, wxTIMER_CONTINUOUS);
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

```

---

In order to enter the parameter values it is possible to use not only text box (`wxEdit`). Sometimes it is more convenient to use scroll bars or sliders (`wxScrollBar` or `wxSlider`). These are rather similar controls which allow to set the parameter value from fixed interval. Both of these controls can be either horizontal or vertical (Orientation property: `wxSB_HORIZONTAL` or `wxSB_VERTICAL`). The main event for both of them is `Scroll`.

`WxScrollBar` is a control that represents a horizontal or vertical scrollbar. A scrollbar has the following main attributes: range, thumb size, page size, and position. The range is the total number of units associated with the view represented by the scrollbar.

Table 5.6b.

Moving bars program code.

---

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    static int x, y, h, w, hb, wb, dx=5, dy=7;
    // dx is horizontal speed, and dy is vertical speed of bar
    WxStaticText1 -> GetPosition(&x,&y);
    GetSize(&w,&h);
    WxStaticText1 -> GetSize(&wb,&hb);
    if (((x<abs(dx)) && (dx<0)) || ((x>(w-wb-dx)) && (dx>0))) dx=-dx;
    if (((y<abs(dy)) && (dy<0)) || ((y>(h-hb-dy)) && (dy>0))) dy=-dy;
    WxStaticText1 -> Move(x+dx,y+dy);
}

void Project1Dlg::WxTimer2Timer(wxTimerEvent& event)
{
    static int x, y, h, w, hb, wb, dx=-7, dy=-3;
    // dx is horizontal speed, and dy is vertical speed of bar
    WxStaticText2 -> GetPosition(&x,&y);
    GetSize(&w,&h);
    WxStaticText2 -> GetSize(&wb,&hb);
    if (((x<abs(dx)) && (dx<0)) || ((x>(w-wb-dx)) && (dx>0))) dx=-dx;
    if (((y<abs(dy)) && (dy<0)) || ((y>(h-hb-dy)) && (dy>0))) dy=-dy;
    WxStaticText2 -> Move(x+dx,y+dy);
}

```

---

For a table with 15 columns, the range would be 15. The thumb size is the number of units that are currently visible.

For the table example, the window might be sized so that only 5 columns are currently visible, in which case the application would set the thumb size to 5. When the thumb size becomes the same as or greater than the range, the scrollbar will be automatically hidden on most platforms. The page size is the number of units that the scrollbar should scroll by, when 'paging' through the data. This value is normally the same as the thumb size length, because it is natural to assume that the visible window size defines a page. The scrollbar position is the current thumb position.

Some scroll bar methods are presented below.

`GetRange()` - returns the length of the scrollbar.

`GetPageSize()` - returns the page size of the scrollbar. This is the number of scroll units that will be scrolled when the user pages up or down. Often it is the same as the thumb size.

`GetThumbPosition()` - returns the current position of the scrollbar thumb.

`GetThumbSize()` - returns the thumb or 'view' size.

`SetThumbPosition(int viewStart)` - sets the position of the scrollbar. The parameter `viewStart` represents the position of the scrollbar thumb.

`SetScrollbar(int position, int thumbSize, int range, int pageSize, const bool refresh = TRUE)` - sets the scrollbar properties. Parameters are

presented below.

|                    |                                                                                                                                                                       |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>position</b> -  | the position of the scrollbar in scroll units.                                                                                                                        |
| <b>thumbSize</b> - | the size of the thumb, or visible portion of the scrollbar, in scroll units.                                                                                          |
| <b>range</b> -     | the maximum position of the scrollbar.                                                                                                                                |
| <b>pageSize</b> -  | the size of the page size in scroll units. This is the number of units the scrollbar will scroll when it is paged up or down. Often it is the same as the thumb size. |
| <b>refresh</b> -   | TRUE to redraw the scrollbar, FALSE otherwise.                                                                                                                        |

**Slider.** A slider is a control with a handle which can be pulled back and forth to change the value. Slider events are handled in the same way as a scrollbar. Some useful methods:

**GetMax()** - gets the maximum slider value.

**GetMin()** - gets the minimum slider value.

**GetPageSize()** - returns the page size.

**GetValue()** - gets the current slider value.

**SetLineSize(int lineSize)** - sets the line size for the slider. The parameter **lineSize** specifies the number of steps the slider moves when the user moves it up or down a line.

**SetPageSize(int pageSize)** - sets the page size for the slider. The parameter **pageSize** specifies the number of steps the slider moves when the user pages up or down.

**SetRange(int minValue, int maxValue)** - sets the minimum and maximum slider values.

**SetValue(int value)** - sets the slider position. The parameter **value** specifies the slider position.

For example, this is a simple program with scroll bar, slider and gauge. We can set scroll bar position using slider, and we can change slider position using scroll bar.

The values are presented using gauge and text box (see Fig. 5.19). The function code is presented in Tables 5.7a, 5.7b.

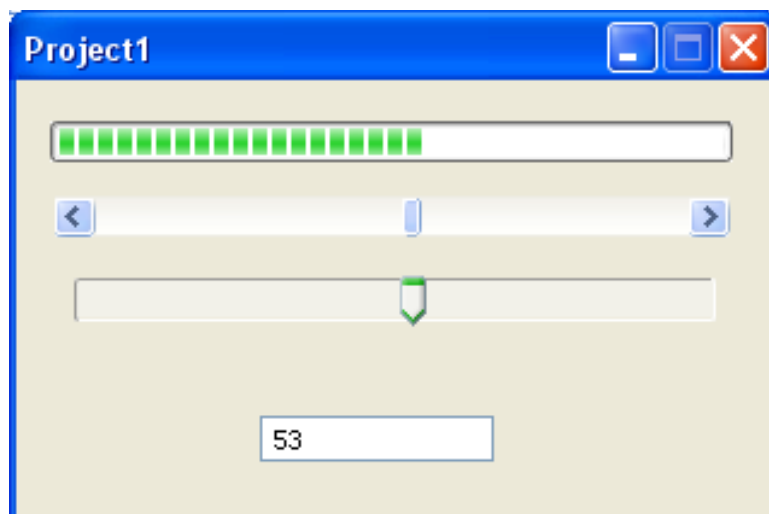


Fig. 5.19. Slider and scrollbar sample program.

---

|             |                                                             |
|-------------|-------------------------------------------------------------|
| Table 5.7a. | Slider and scrollbar sample program<br>event function code. |
|-------------|-------------------------------------------------------------|

---

```

void Project1Dlg::CreateGUIControls()
{
    WxGauge1 = new wxGauge(this, ID_WXGAUGE1, 100,
                           wxPoint(14, 17), wxSize(283, 17),
                           wxGA_HORIZONTAL,
                           wxDefaultValidator, _("WxGauge1"));
    WxGauge1 -> SetRange(100);
    WxGauge1 -> SetValue(0);

    WxEdit1 = new wxTextCtrl(this, ID_WXEDIT1, _(""),
                             wxPoint(101, 139), wxSize(97, 19),
                             0,
                             wxDefaultValidator, _("WxEdit1"));

    WxSlider1 = new wxSlider(this, ID_WXSLIDER1, 0, 0, 10,
                             wxPoint(16, 78), wxSize(282, 45),
                             wxSL_HORIZONTAL | wxSL_SELRANGE ,
                             wxDefaultValidator, _("WxSlider1"));
    WxSlider1 -> SetRange(0,10);
    WxSlider1 -> SetValue(0);

    WxScrollBar1 = new wxScrollBar(this, ID_WXSCROLLBAR1,
                                    wxPoint(16, 48), wxSize(280, 17),
                                    wxSB_HORIZONTAL,
                                    wxDefaultValidator, _("WxScrollBar1"));

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,212);
    Center();

    //// GUI Items Creation End
    WxScrollBar1 -> SetScrollbar(0, 1, 100, 10, true);
    WxSlider1 -> SetRange(0,100);
    WxSlider1 -> SetValue(0);
}

```

---

**Colour mixing program.** Let us modify previously created colour mixing program using sliders (see Fig. 5.20). We need three sliders with minimum value 0 and maximum value 255 each. Slider1 scroll event function code is presented in Table 5.8. The same code should be copied to WxSlider2 and WxSlider3 scroll event functions.

Table 5.7b.

Slider and scrollbar sample program  
event function code.

---

```

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::Project1DlgActivate(wxActivateEvent& event)
{
    // insert
    // your
    // code
    // here
}
void Project1Dlg::WxScrollBar1Scroll(wxScrollEvent& event)
{
    WxSlider1 -> SetValue (WxScrollBar1 -> GetThumbPosition());
    WxGauge1  -> SetValue (WxScrollBar1 -> GetThumbPosition());
    wxString ss;
    ss << WxScrollBar1 -> GetThumbPosition();
    WxEdit1 -> SetValue(ss);
}
void Project1Dlg::WxSlider1Scroll(wxScrollEvent& event)
{
    WxScrollBar1 -> SetThumbPosition (WxSlider1 -> GetValue());
    WxGauge1      -> SetValue          (WxSlider1 -> GetValue());
    wxString ss;
    ss << WxSlider1 -> GetValue();
    WxEdit1 -> SetValue(ss);
}

```

---

Table 5.8.

Colour mixing program  
scroll event function code.

---

```

void Project1Dlg::WxSlider1Scroll(wxScrollEvent& event)
{
    WxStaticText4 -> SetBackgroundColour (wxColour (WxSlider1 -> GetValue(),
                                                    WxSlider2 -> GetValue(),
                                                    WxSlider3 -> GetValue()));
    WxStaticText4 -> Refresh();
}

```

---

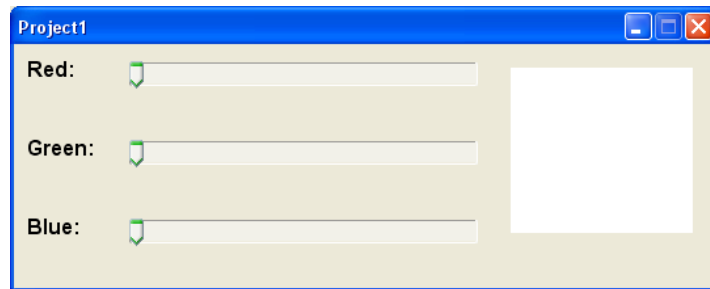


Fig. 5.20. Colour mixing program interface.

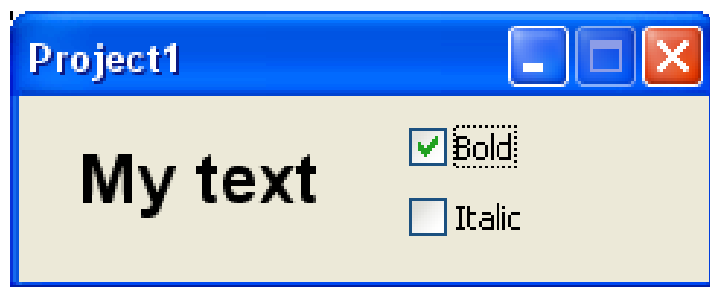


Fig. 5.21. Font style changing program interface.

## 5.7 Selection controls

It is convenient to use check boxes (`wxCheckBox`) or radio buttons (`wxRadioButton`) while working with binary data. If there are several check boxes in the form, they work independently from each other. If there are several radio buttons they compound a group, where only one radio button from the group can be checked. If we want to have several groups of radio buttons, we should use radio box control (`wxRadioBox`).

We can set check box or radio button value by using `SetValue()` method. The argument of this method is boolean constant (`TRUE` or `FALSE`). In order to get the value of check box or radio button we use `GetValue()` method. We get a boolean value which indicates, whether the component is checked or not.

For example, let us use check boxes to change the font stile (bold and italic). The program interface is shown in Fig. 5.21. `CheckBox1` click event function code is presented in Table 5.9. The same code should be copied into `CheckBox2` click event function.

We can use list box (`wxListBox`) or combo box (`wxComboBox`), if we want to select one or several options from a fixed list. Combo box enables us not only to select a value from a list, but also to enter a new value.

It is possible to fill in a list box using `Strings` property during the program design. We can also edit the content of a list box during the program execution. We can use `Insert()` or `Append()` methods to add an item into a list box or a combo box and `Delete()` method to remove an item. Method `Clear()` removes all items from the control. `GetSelections()` method allows to determine, which items are selected in a list box. `GetValue()` method is used to determine combo box value. `GetCount()` method returns the number of items in the control.

Table 5.9. Font style changing program code.

---

```

void Project1Dlg::WxCheckBox1Click(wxCommandEvent& event)
{
    if      ((WxCheckBox1 -> GetValue()) && (WxCheckBox2 -> GetValue()))
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxITALIC, wxBOLD,
                                         false));

    else if (WxCheckBox1 -> GetValue())
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxNORMAL, wxBOLD,
                                         false));

    else if (WxCheckBox2 -> GetValue())
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxITALIC, wxNORMAL,
                                         false));

    else
        WxStaticText1 -> SetFont(wxFont(18, wxSWISS, wxNORMAL, wxNORMAL,
                                         false));
}

```

---

For example, let us create a program with list box. It should be possible to add new items into a list box, to remove selected ones (Fig. 5.22). Function code is presented in Table 5.10.

## 5.8 Keyboard events

Keyboard is a common and very convenient input device. **WxWidgets** programs are able to respond to keyboard events. There are three keyboard events, that are identified by **WxWidgets** program: **KeyDown**, **KeyUp** and **Char**.

The difference between the first two is clear – the first corresponds to a key press and the second to a key release – otherwise they are identical. Just note, that if the key is maintained in a pressed state you will typically get a lot of (automatically generated) down events but only one up. So, it is wrong to assume, that there is one up event corresponding to each down one. **Char** event does not distinguish, whether the key is up or down. It reacts to key press and carries the translated key code.

Both key events provide untranslated key codes, while the **char** event carries the translated one. The untranslated code for alphanumeric keys is always an upper case value. For the other keys it is one of **WXK\_XXX** values from the keycodes table. The translated key is, in general, the character, the user expects to appear as the result of the key combination when typing the text into a text entry zone, for example.

A few examples to clarify this (all assume that **CAPS LOCK** is unpressed and the standard US keyboard is used).

When the 'A' key is pressed, the key down event key code is equal to ASCII **A == 65**. But the **char** event key code is ASCII **a == 97**. On the other hand, if you press both **SHIFT** and 'A' keys simultaneously, the key code in key down event will still be just 'A' while the **char** event key code parameter will now be 'A' as well.

Table 5.10.

List box sample program event function code.

---

```

void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    if (WxEdit1 -> GetValue() != "")
    {
        int count;
        wxString ss;
        count = wxAtoi(WxEdit2 -> GetValue());
        WxListBox1 -> Append(WxEdit1 -> GetValue());
        count++;
        ss << count;
        WxEdit2 -> SetValue(ss);
        WxEdit1 -> SetValue("");
    }
}

void Project1Dlg::WxListBox1Selected(wxCommandEvent& event)
{
    WxEdit3 -> SetValue (WxListBox1 ->
                        GetString (WxListBox1 -> GetSelection()));
}

void Project1Dlg::WxButton2Click(wxCommandEvent& event)
{
    if (WxListBox1 -> GetSelection() != wxNOT_FOUND)
    {
        WxListBox1 -> Delete(WxListBox1 -> GetSelection());
        wxString ss;
        ss << WxListBox1 -> GetCount();
        WxEdit2 -> SetValue(ss);
        WxEdit3 -> SetValue("");
    }
}

```

---

Although in this simple case it is clear that the correct key code could be found in the key down event handler by checking the value returned by `ShiftDown`, in general, you should use `EVT_CHAR` for these as for non alphanumeric keys or non-US keyboard layouts the translation is keyboard-layout dependent and can only be done properly by the system itself.

Another kind of translation is done, when the control key is pressed.

For example, for *CTRL-A* key press the key down event still carries the same key code 'A', as usual, but the char event will have key code of 1, the ASCII value of this key combination.

Key codes are presented in Table 5.11.

Summary of most common methods is presented below.

|                              |                                                                                        |
|------------------------------|----------------------------------------------------------------------------------------|
| <code>AltDown()</code> -     | returns <code>TRUE</code> if the <i>Alt</i> key was down at the time of the event.     |
| <code>ControlDown()</code> - | returns <code>TRUE</code> if the <i>Control</i> key was down at the time of the event. |
| <code>GetKeyCode()</code> -  | returns the virtual key code.                                                          |

| Table 5.11.          | Key codes.    |                   |
|----------------------|---------------|-------------------|
| WXK_NUMPAD_SPACE     | WXK_BACK      | WXK_WINDOWS_LEFT  |
| WXK_NUMPAD_TAB       | WXK_EXECUTE   | WXK_WINDOWS_RIGHT |
| WXK_NUMPAD_ENTER     | WXK_SNAPSHOT  | WXK_WINDOWS_MENU  |
| WXK_NUMPAD_HOME      | WXK_TAB       | WXK_RBUTTON       |
| WXK_NUMPAD_END       | WXK_RETURN    | WXK_LBUTTON       |
| WXK_NUMPAD_PAGEUP    | WXK_INSERT    | WXK_MBUTTON       |
| WXK_NUMPAD_PAGEDOWN  | WXK_CLEAR     | WXK_PAUSE         |
| WXK_NUMPAD_NEXT      | WXK_CONTROL   | WXK_CAPITAL       |
| WXK_NUMPAD_PRIOR     | WXK_ALT       | WXK_MENU          |
| WXK_NUMPAD_BEGIN     | WXK_ESCAPE    | WXK_HELP          |
| WXK_NUMPAD_RIGHT     | WXK_SHIFT     | WXK_CANCEL        |
| WXK_NUMPAD_LEFT      | WXK_MULTIPLY  | WXK_SEPARATOR     |
| WXK_NUMPAD_UP        | WXK_SUBTRACT  | WXK_PRIOR         |
| WXK_NUMPAD_DOWN      | WXK_ADD       | WXK_NEXT          |
| WXK_NUMPAD_INSERT    | WXK_DIVIDE    | WXK_END           |
| WXK_NUMPAD_DELETE    | WXK_NUMLOCK   | WXK_HOME          |
| WXK_NUMPAD_EQUAL     | WXK_DECIMAL   | WXK_LEFT          |
| WXK_NUMPAD_MULTIPLY  | WXK_SCROLL    | WXK_RIGHT         |
| WXK_NUMPAD_DIVIDE    | WXK_PAGEUP    | WXK_DOWN          |
| WXK_NUMPAD_ADD       | WXK_PAGEDOWN  | WXK_UP            |
| WXK_NUMPAD_SUBTRACT  | WXK_SELECT    | WXK_SPACE         |
| WXK_NUMPAD_SEPARATOR | WXK_DELETE    | WXK_PRINT         |
| WXK_NUMPAD_DECIMAL   |               |                   |
| WXK_NUMPAD_F1 ...    | WXK_NUMPAD_F4 |                   |
| WXK_SPECIAL1 ...     | WXK_SPECIAL20 |                   |
| WXK_NUMPAD0 ...      | WXK_NUMPAD9   |                   |

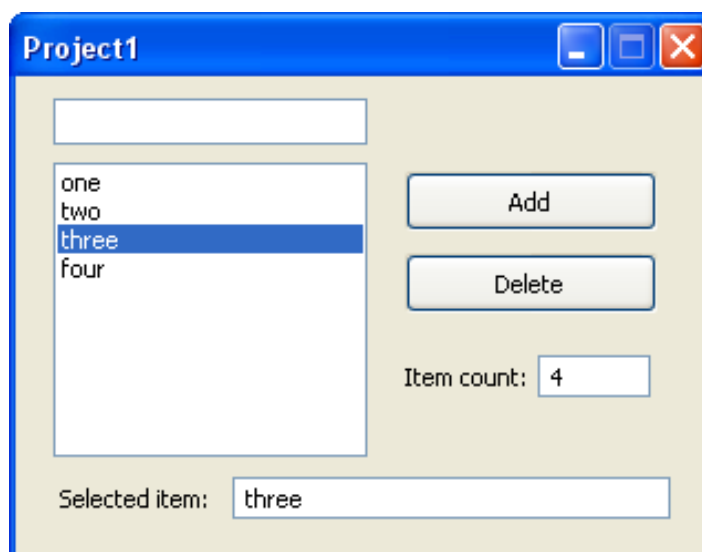


Fig. 5.22. List box sample program interface.

Table 5.12. *KeyDown* event function code.

---

```

void Project1Dlg::Project1DlgKeyDown(wxKeyEvent& event)
{
    wxString ss;
    ss << " KeyCode=" << event.GetKeyCode()
        << " Alt=" << event.AltDown();
    ss << " Control=" << event.ControlDown()
        << " Shift=" << event.ShiftDown();
    ss << " Key character=" << (char)event.GetKeyCode();
    WxStaticText1 -> SetLabel(ss);
}

```

---

**HasModifiers()** - returns TRUE if either *CTRL* or *ALT* keys was down at the time of the key event.

**ShiftDown()** - returns TRUE if the *Shift* key was down at the time of the event.

Let us create a simple program demonstrating keyboard event handling (see Fig. 5.23). We must set *General Style* - **wxWANTS\_CHARS** property value TRUE to get all key codes. The event function code is presented in Table 5.12.

## 5.9 Mouse events

**WxWidgets** program can react to different mouse events: **LeftDown**, **LeftUp**, **LeftDclick** (left mouse button), **MiddleDown**, **MiddleUp**, **MiddleDclick** (middle mouse button), **RightDown**, **RightUp**, **RightDclick** (right mouse button), **MouseMotion**, **EnterWindow**, **LeaveWindow**, **MouseWheel** and **MouseEvents** . The last one processes all the mouse events.

Handling mouse events we will need some additional methods.

**AltDown()** - returns TRUE if the *Alt* key was down at the time of the event.

**Button(button)** - returns TRUE if the identified mouse button is changing state.

Valid values of **button** are presented below:

|                             |                                     |
|-----------------------------|-------------------------------------|
| <b>wxMOUSE_BTN_LEFT</b> -   | check if left button was pressed;   |
| <b>wxMOUSE_BTN_MIDDLE</b> - | check if middle button was pressed; |
| <b>wxMOUSE_BTN_RIGHT</b> -  | check if right button was pressed;  |
| <b>wxMOUSE_BTN_ANY</b> -    | check if any button was pressed.    |

**ButtonDClick(button)** - returns TRUE if the event was a mouse double click event. The argument specifies, which double click event was generated.

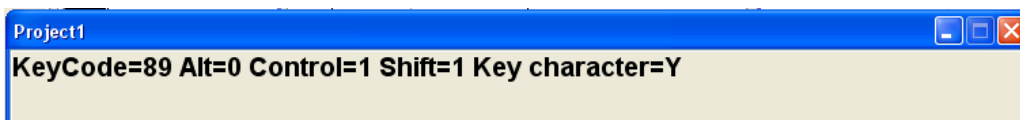


Fig. 5.23. Keyboard events handling program.

`ButtonDown(button)` - returns `TRUE` if the event was a mouse button down event. The argument specifies, which button-down event was generated.

`ButtonUp(button)` - returns `TRUE` if the event was a mouse button up event. The argument specifies, which button-up event was generated.

`ControlDown()` - returns `TRUE` if the control key was down at the time of the event.

`Dragging()` - returns `TRUE` if this was a dragging event (motion, while a button is depressed).

`Entering()` - returns `TRUE` if the mouse was entering the window.

`GetButton()` - returns the mouse button, which generated this event or `wxMOUSE_BTN_NONE` if no button is involved. For mouse move, enter or leave event, for example. Otherwise, `wxMOUSE_BTN_LEFT` is returned for the left button down, up and double click events, `wxMOUSE_BTN_MIDDLE` and `wxMOUSE_BTN_RIGHT` for the same events for the middle and the right buttons respectively.

`GetPosition(long* x, long* y)` - sets `*x` and `*y` to the position, at which the event occurred on the window. Returns the physical mouse position on the window in pixels.

`GetX()` - returns the *X* coordinate of the physical mouse event position on the window.

`GetY()` - returns the *Y* coordinate of the physical mouse event position on the window.

`IsButton()` - returns `TRUE` if the event was a mouse button event (not necessarily a button down event - that may be tested using `ButtonDown`).

`Leaving()` - returns `TRUE` if the mouse was leaving the window.

`LeftDClick()`, `MiddleDClick()`, `RightDClick()` - returns `TRUE` if the event was a left or middle or right double click.

`LeftDown()`, `MiddleDown()`, `RightDown()` - returns `TRUE` if the left or middle or right mouse button changed to down.

`LeftIsDown()`, `MiddleIsDown()`, `RightIsDown()` - returns `TRUE` if the left or middle or right mouse button is currently down, independent of the current event type.

`LeftUp()`, `MiddleUp()`, `RightUp()` - returns `TRUE` if the left or middle or right mouse button changed to up.

`Moving()` - returns `TRUE` if this was a motion event and no mouse buttons were pressed. If any mouse button is held pressed, this method returns `FALSE` and `Dragging()` returns `TRUE`.

`ShiftDown()` - returns `TRUE` if the shift key was down at the time of the event.

Let us create a sample mouse event illustrating program. The position of the green rectangle in the program form will be set using mouse click (see Fig. 5.24). Event function code is presented in Table 5.13.

Now we will improve previous program. The green rectangle will go after the mouse pointer (see Table 5.14).

---

|             |                            |
|-------------|----------------------------|
| Table 5.13. | Mouse event function code. |
|-------------|----------------------------|

---

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    int x, y;
    event.GetPosition(&x, &y);
    WxStaticText1 -> Move(x,y);
}
```

---

Table 5.14.

*MouseMotion* event function code.

```
void Project1Dlg::Project1DlgMouseMotion(wxMouseEvent& event)
{
    // variable indicating which part of the distance
    // the rectangle will cover
    float ko = 0.05;
    int x1, y1, x2, y2, dx, dy;

    // the position of static text control
    // (our rectangle)
    WxStaticText1 ->GetPosition(&x1, &y1);

    // the position of mouse pointer
    event.GetPosition(&x2, &y2);
    dx = x2 - x1;
    dy = y2 - y1;
    WxStaticText1 -> Move(x1+(int)(dx*ko),y1+(int)(dy*ko));
    // new rectangle position
}
```

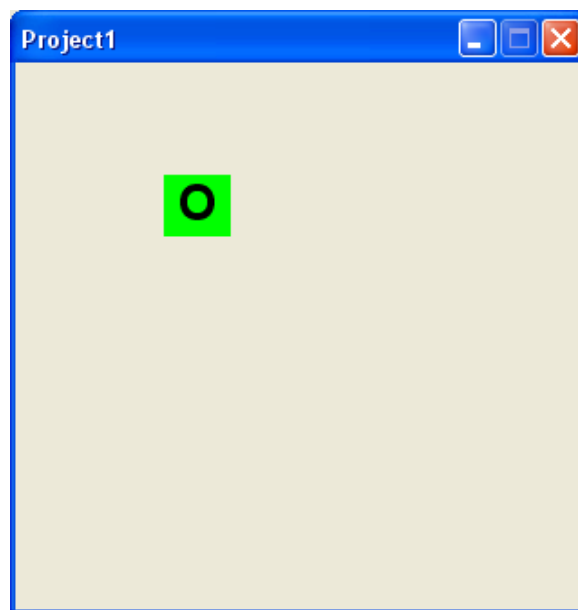


Fig. 5.24. Sample mouse event illustrating program interface.

## 5.10 Graphical controls and methods

It is possible to operate with images and graphics in `wxWidgets` programs. We can insert image into `wxWidgets` window using `wxStaticBitmap` control. A static bitmap control displays a bitmap. We can assign the image to show using *Picture* property.

Some useful methods are presented below.

`GetBitmap()` - returns the bitmap currently used in the control.

`SetBitmap(wxBitmap)` - sets the bitmap.

To set static bitmap picture during the program run, we must use `wxBitmap` object and its methods.

`LoadFile(name, wxBitmap type)` - loads a bitmap from a file or resource. Function parameters are as follows:

i) **name** - either a filename or a Windows resource name. The meaning of name is determined by the type parameter.

ii) **type** - one of the following values:

|                                            |                                                                     |
|--------------------------------------------|---------------------------------------------------------------------|
| <code>wxBITMAP_TYPE_BMP</code> -           | load a Windows bitmap file;                                         |
| <code>wxBITMAP_TYPE_BMP_RESOURCE</code> -  | load a Windows bitmap resource from the executable;                 |
| <code>wxBITMAP_TYPE_PICT_RESOURCE</code> - | load a PICT image resource from the executable. <i>Mac</i> OS only; |
| <code>wxBITMAP_TYPE_GIF</code> -           | load a GIF bitmap file;                                             |
| <code>wxBITMAP_TYPE_XBM</code> -           | load an X bitmap file;                                              |
| <code>wxBITMAP_TYPE_XPM</code> -           | load an XPM bitmap file.                                            |

In addition, `wxBitmap` can read all formats that `wxImage` can such as following: `wxBITMAP_TYPE_JPEG`, `wxBITMAP_TYPE_PNG`, `wxBITMAP_TYPE_GIF`, `wxBITMAP_TYPE_PCX`, `wxBITMAP_TYPE_PNM`. (Of course you must have `wxImage` handlers loaded.)

`SaveFile(name, wxBitmap type, wxPalette* palette = NULL)` - saves a bitmap in the named file. Function parameters are follows:

i) **name** - a filename. The meaning of name is determined by the type parameter;

ii) **type** - one of the following values:

|                                  |                             |
|----------------------------------|-----------------------------|
| <code>wxBITMAP_TYPE_BMP</code> - | save a Windows bitmap file; |
| <code>wxBITMAP_TYPE_GIF</code> - | save a GIF bitmap file;     |
| <code>wxBITMAP_TYPE_XBM</code> - | save an X bitmap file;      |
| <code>wxBITMAP_TYPE_XPM</code> - | save an XPM bitmap file.    |

In addition, `wxBitmap` can save all formats that `wxImage` can such as following: (`wxBITMAP_TYPE_JPEG`, `wxBITMAP_TYPE_PNG`). (Of course you must have `wxImage` handlers loaded.)

iii) **palette** - an optional palette used for saving the bitmap.

**Simple animation.** Let us create simple animation program: a butterfly flies from the bottom left corner of the program window to the upper right corner after *Start* button is pressed (see Fig. 5.25). Animation effect occurs due to moving `wxStaticBitmap1` control and changing pictures in this bitmap. Function code is presented in Tables 5.15a, 5.15b.

We can use graphic methods to draw on the program window. A `wxDC` object is a device context onto which graphics and text can be drawn.

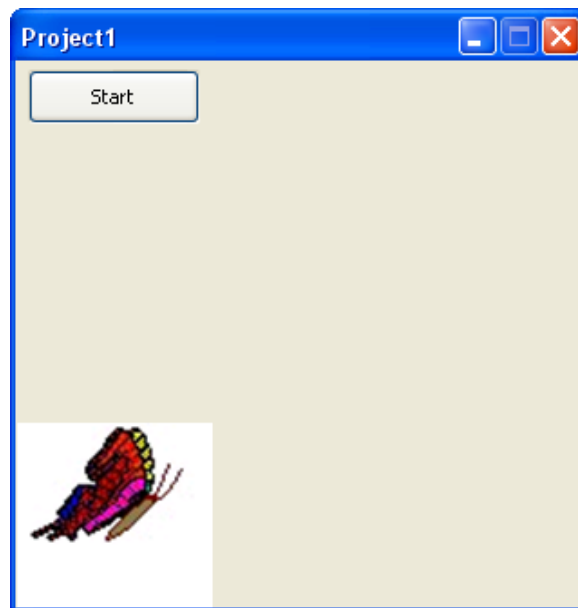


Fig. 5.25. Interface of animation program.

The device context is intended to represent a number of output devices in a generic way, with the same API being used throughout.

Some device contexts are created temporarily in order to draw on a window. This is TRUE of `wxScreenDC`, `wxClientDC`, `wxPaintDC`, and `wxWindowDC`. The following describes the difference between these device contexts and situation when you should use them.

`wxScreenDC` - it is used to paint on the screen, as opposed to an individual window.

`wxClientDC` - it is used to paint on the client area of window (the part without borders and other decorations), but do not use it from within an `wxPaintEvent`.

`wxPaintDC` - it is used to paint on the client area of a window, but only from within a `wxPaintEvent`.

`wxWindowDC` - it is used to paint on the whole area of a window, including decorations. This may not be available on non-Windows platforms. To use a client, paint or window device context, create an object on the stack with the window as argument.

We can draw different shapes and print out text using following methods.

`Clear()` - clears the device context using the current background brush.

`DrawArc(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2, wxCoord xc, wxCoord yc)` - draws an arc of a circle, centred on  $(xc, yc)$ , with starting point  $(x1, y1)$  and ending at  $(x2, y2)$ . The current pen is used for the outline and the current brush for filling the shape. The arc is drawn in an anticlockwise direction from the start point to the end point.

`DrawCircle(wxCoord x, wxCoord y, wxCoord radius)` or

`DrawCircle(const wxPoint& pt, wxCoord radius)` - draws a circle with the given centre and radius.

`DrawEllipse(const wxRect& rect)` or

`DrawEllipse(const wxPoint& pt, const wxSize& size)` or

`DrawEllipse(wxCoord x, wxCoord y, wxCoord width, wxCoord height)` - draws an ellipse contained in the rectangle specified either with the given top left corner and the given size or directly. The current pen is used for the outline and the current brush for filling the shape.

---

Table 5.15a. Animation program code.

---

```

void Project1Dlg::CreateGUIControls()
{
    // Do not add custom code between
    // GUI Items Creation Start and GUI Items Creation End.
    // wxDev-C++ designer will remove them.
    // Add the custom code before or after the blocks
    ////GUI Items Creation Start

    wxInitAllImageHandlers();    //Initialize graphic format handlers

    WxButton1 = new wxButton(this, ID_WXBUTTON1, _("Start"), wxPoint(7, 5),
        wxSize(95, 30), 0, wxDefaultValidator, _("WxButton1"));

    WxTimer1 = new wxTimer();
    WxTimer1->SetOwner(this, ID_WXTIMER1);

    WxStaticBitmap1 = new wxStaticBitmap(this, ID_WXSTATICBITMAP1,
        wxNullBitmap, wxPoint(1, 188), wxSize(111, 111) );

    SetTitle(_("Project1"));
    SetIcon(wxNullIcon);
    SetSize(8,8,320,334);
    Center();

    ////GUI Items Creation End
    WxStaticBitmap1 -> Move(1, 200);
    wxBitmap pic;
    pic.LoadFile("Butterfly1.bmp", wxBITMAP_TYPE_BMP);
    WxStaticBitmap1 -> SetBitmap(pic);
}

void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}

```

---

`DrawEllipticArc(wxCoord x, wxCoord y, wxCoord width, wxCoord height, double start, double end)` - draws an arc of an ellipse. The current pen is used for drawing the arc and the current brush is used for drawing the pie. `x` and `y` specify the coordinates of the upper-left corner of the rectangle that contains the ellipse. `width` and `height` specify the width and height of the rectangle that contains the ellipse, `start` and `end` specify the start and end of the arc relative to the three-o'clock position from the center of the rectangle. Angles are specified in degrees (360 is a complete circle). Positive values mean counter-clockwise motion. If `start` is equal to `end`, a complete ellipse will be drawn.

`DrawLine(wxCoord x1, wxCoord y1, wxCoord x2, wxCoord y2)` - draws a line from the first point to the second. The current pen is used for drawing the line.

Table 5.15b.

Animation program code.

---

```

void Project1Dlg::WxTimer1Timer(wxTimerEvent& event)
{
    static int i=1, j=1;    int x, y;
    WxStaticBitmap1 -> GetPosition(&x, &y);
    if (i==1)
    {
        wxBitmap pic;
        pic.LoadFile("Butterfly1.bmp", wxBITMAP_TYPE_BMP);
        WxStaticBitmap1 -> SetBitmap(pic);
        i=2; j++;
    }
    else
    {
        wxBitmap pic;
        pic.LoadFile("Butterfly2.bmp", wxBITMAP_TYPE_BMP);
        WxStaticBitmap1 -> SetBitmap(pic);
        i=1;
    };
    WxStaticBitmap1 -> Move(x+5, y-5);
    if (j==20) WxTimer1 -> Stop();
}

void Project1Dlg::WxButton1Click(wxCommandEvent& event)
{
    WxStaticBitmap1 -> Move(1, 200);
    WxTimer1         -> Start(100);
}

```

---

Note that the point (x2, y2) is not part of the line and is not drawn by this function (this is consistent with the behaviour of many other toolkits).

`DrawLines(int n, wxPoint points[], wxCoord xoffset=0, wxCoord yoffset=0)` or  
`DrawLines(wxList *points, wxCoord xoffset=0, wxCoord yoffset=0)` - draws lines using an array of points of size `n`, or list of pointers to points, adding the optional offset coordinate. The current pen is used for drawing the lines. The programmer is responsible for deleting the list of points.

`DrawPolygon(int n, wxPoint points[], wxCoord xoffset=0, wxCoord yoffset=0, int fill_style = wxODDEVEN_RULE)` or

`DrawPolygon(wxList *points, wxCoord xoffset=0, wxCoord yoffset=0, int fill_style = wxODDEVEN_RULE)` - draws a filled polygon using an array of points of size `n`, or list of pointers to points, adding the optional offset coordinate. The last argument specifies the fill rule: `wxODDEVEN_RULE` (the default) or `wxWINDING_RULE`. The current pen is used for drawing the outline. The current brush is used for filling the shape. The use of a transparent brush suppresses filling. The programmer is responsible for deleting the list of points.

`DrawPolyPolygon(int n, int count[], wxPoint points[], wxCoord xoffset = 0, wxCoord yoffset = 0, int fill_style = wxODDEVEN_RULE)` - draws two or more filled poly-

gons using an array of points, adding the optional offset coordinates. Notice, that for the platforms providing a native implementation of this function (*Windows* and *PostScript*-based `wxDC` currently), this is more efficient than using `DrawPolygon` in a loop. The number of polygons to draw is specified by parameter `n`, the array count of size `n` specifies the number of points in each of the polygons in the points array. The last argument specifies the fill rule: `wxODDEVEN_RULE` (the default) or `wxWINDING_RULE`. The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling. The polygons may be disjoint or overlapping. Each polygon specified in a call to `DrawPolyPolygon` must be closed. Unlike polygons created by the `DrawPolygon` member function, the polygons created by `DrawPolyPolygon` are not closed automatically.

`DrawPoint(wxCoord x, wxCoord y)` - draws a point using the colour of the current pen. Note, that the other properties of the pen, such as width etc. are not used.

`DrawRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height)` - draws a rectangle with the given top left corner, and with the given size. The current pen is used for the outline and the current brush for filling the shape.

`DrawRotatedText(const wxString& text, wxCoord x, wxCoord y, double angle)` - draws the text rotated by angle degrees. Note, that under *Windows9x* only *TrueType* fonts can be drawn by this function. In particular, a font different from `wxNORMAL_FONT` should be used as the latter is not a textitTrueType font. `wxSWISS_FONT` is an example of suitable font.

`DrawRoundedRectangle(wxCoord x, wxCoord y, wxCoord width, wxCoord height, double radius)` - draws a rectangle with the given top left corner and with the given size. The corners are quarter-circles using the given radius. The current pen is used for the outline and the current brush for filling the shape. If the radius is positive, the value is assumed to be the radius of the rounded corner. If radius is negative, the absolute value is assumed to be the proportion of the smallest dimension of the rectangle. This means that the corner can be a sensible size relative to the size of the rectangle and avoids the strange effects X produces when the corners are too big for the rectangle.

`DrawText(const wxString& text, wxCoord x, wxCoord y)` - draws a text string at the specified point, using the current text font, the current text foreground and background colours. The coordinates refer to the top-left corner of the rectangle bounding the string.

`FloodFill(wxCoord x, wxCoord y, const wxColour& colour, int style=wxFLOOD_SURFACE)` - flood fills the device context starting from the given point, using the current brush colour and using a style: `wxFLOOD_SURFACE` - the flooding occurs until a colour other than the given colour is encountered. `wxFLOOD_BORDER` - the area to be flooded is bounded by the given colour.

`GetBackground()` - gets the brush used for painting the background.

`GetBackgroundMode()` - returns the current background mode: `wxSOLID` or `wxTRANSPARENT`.

`GetBrush()` - gets the current brush.

`GetFont()` - gets the current font. Notice, that even although each device context object has some default font after creation, this method would return a `wxNullFont` initially and only after calling `wxDC::SetFont` a valid font is returned.

`GetPen()` - gets the current pen.

`GetPixel(wxCoord x, wxCoord y, wxColour *colour)` - gets the colour at the specified location.

`GetTextBackground()` - gets the current text background colour.

`GetTextForeground()` - gets the current text foreground colour.

`GradientFillConcentric(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour)` or

`GradientFillConcentric(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour, const wxPoint& circleCenter)` - fill the area specified by `rect` with a radial gradient, starting from `initialColour` at the centre of the circle and fading to `destColour` on the circle outside. `circleCenter` are the relative coordinates of centre of the circle in the specified `rect`. If not specified, the circle is placed at the centre of `rect`.

`GradientFillLinear(const wxRect& rect, const wxColour& initialColour, const wxColour& destColour, wxDirection nDirection = wxEAST)` - fill the area specified by `rect` with a linear gradient, starting from `initialColour` and eventually fading to `destColour`.

The `nDirection` specifies the direction of the colour change, default is to use `initialColour` on the left part of the rectangle and `destColour` on the right one.

`SetBackground(const wxBrush& brush)` - sets the current background brush for the DC. `SetBackgroundMode(int mode)` - mode may be one of `wxSOLID` and `wxTRANSPARENT`. This setting determines whether text will be drawn with a background colour or not.

`SetBrush(const wxBrush& brush)` - sets the current brush for the DC. `SetFont(const wxFont& font)` - sets the current font for the DC. It must be a valid font, in particular you should not pass `wxNullFont` to this method.

`SetPen(const wxPen& pen)` - sets the current pen for the DC.

`SetTextBackground(const wxColour& colour)` - sets the current text background colour for the DC.

`SetTextForeground(const wxColour& colour)` - sets the current text foreground colour for the DC.

Graphic methods draw directly on program window or other DC. Graphic methods are executed during program run. So, we see the result of these methods only during program execution.

Let us create a simple program drawing a circle in that place, where mouse left button is clicked. Mouse event function code is presented in Table 5.16.

We can copy the code into `MouseMotion` event function. In this case circles will appear continuously while moving mouse cursor on top of the program window (see Fig. 5.26).

It is also possible to draw a line using mouse events (see Table 5.17, Fig. 5.27). We can add circle in to click point (see Table 5.18).

Table 5.16.

Mouse event function code.

---

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    dc.DrawCircle(event.GetPosition(), 3);
    dc.SetPen(wxNullPen);
}
```

---

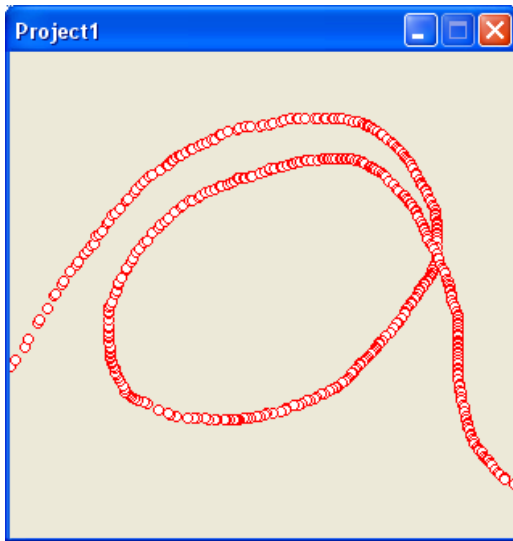


Fig. 5.26. Drawing program.

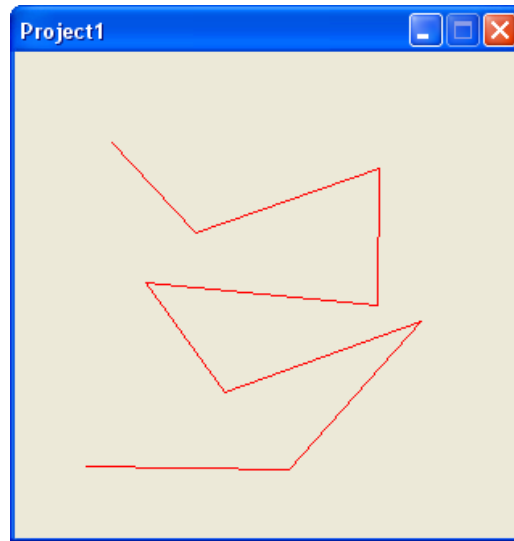


Fig. 5.27. Line drawing program.

---

Table 5.17. Line drawing code.


---

```
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    static int x1=0, y1=0;
    int x2, y2;
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    event.GetPosition(&x2, &y2);
    if ((x1==0)&&(y1==0)) dc.DrawLine(x2, y2, x2, y2);
    else dc.DrawLine(x1, y1, x2, y2);
    x1=x2; y1=y2;
    dc.SetPen(wxNullPen);
}
```

---

## 5.11 Menu, dialog boxes

It is very convenient to use menus, while implementing a large program. We can not use menu bar component while working with *wxWidgetsDialog* project type. This component will be observed later. It is possible to use popup menu while working with *wxWidgetsDialog* project. We must add *wxPopupMenu* component to the form to use this type of menu. We use *Menu Items* property to fill in the menu. Menu items are managed using special editor (see Fig. 5.28).

Table 5.18.

Line drawing with circles.

---

```

void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    static int x1=0, y1=0;
    int x2, y2;
    wxClientDC dc(this);
    wxPen pen(*wxRED, 1); // red pen of width 1
    dc.SetPen(pen);
    event.GetPosition(&x2, &y2);
    dc.DrawCircle(x2,y2,3);
    if ((x1==0)&&(y1==0))    dc.DrawLine(x2, y2, x2, y2);
    else                    dc.DrawLine(x1, y1, x2, y2);
    x1=x2; y1=y2;
    dc.SetPen(wxNullPen);
}

```

---

Menu items field of the editor presents already created items. It is possible to add a new menu item using *Add Item* button. We can set a type of the item (*Menu Item*, *Separator*, *Check Item*, *Radio Item* or *File History*) in the *Type* field. We can also set other parameters (such as *Caption*, *ID Name*, *Enabled*, *Hint* and others) and set event processing functions in the *Events* area. We should press *Apply* to save the created item. It is possible to edit or to delete earlier created items, or to create submenu using this editor (see Fig. 5.28). We can close the editor window pressing *OK* or *Cancel* button. We can show the popup menu using `PopupMenu()` method during the execution of the program.

It is possible to use different dialogues to improve the program behaviour. *WxWidgets* offer wide variety of dialogues: *wxColourDialog*, *wxDirDialog*, *wxFindReplaceDialog*, *wxFontDialog*, *wxMessageDialog*, *wxMultiChoiceDialog*, *wxOpenFileDialog*, *wxPageSetupDialog*, *wxPasswordEntryDialog*, *wxPrintDialog*, *wxProgressDialog*, *wxSaveFileDialog*, *wxSingleChoiceDialog* and *wxTextEntryDialog* (see Fig. 5.29).

Some methods are suitable for all dialogues:

`Centre(int direction = wxBOTH)` - centres the dialog box on the display. Parameter *direction* may be `wxHORIZONTAL`, `wxVERTICAL` or `wxBOTH`.

`Show(const bool show)` - hides or shows the dialog. If the parameter *show* value is `TRUE`, the dialog box is shown and brought to the front; otherwise the box is hidden. If the parameter *show* value is `FALSE` and the dialog is modal, control is returned to the calling program.

`ShowModal()` - shows a modal dialog. Program flow does not return until the dialog has been dismissed with `EndModal()`. The purpose of dialogues is clear from their names. Let us briefly review most common properties of some dialogues.

`wxMessageDialog` - this class represents a dialog that shows a single or multi-line message, with a choice of *OK*, *Yes*, *No* and *Cancel* buttons. This class supports the following styles.

- `wxOK` - puts an *OK* button in the message box. May be combined with `wxCANCEL`.
- `wxCANCEL` - puts a *Cancel* button in the message box. Must be combined with either `wxOK` or `wxYES_NO`.

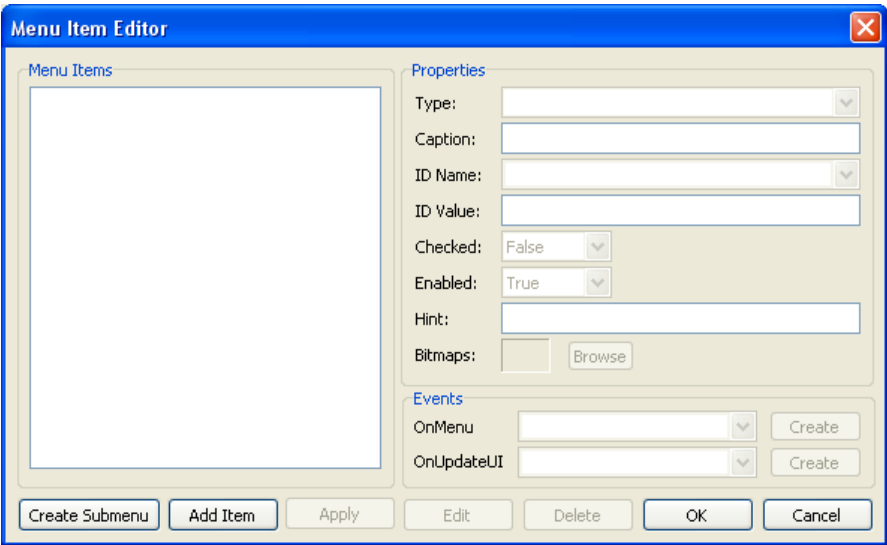


Fig. 5.28. Menu item editor.

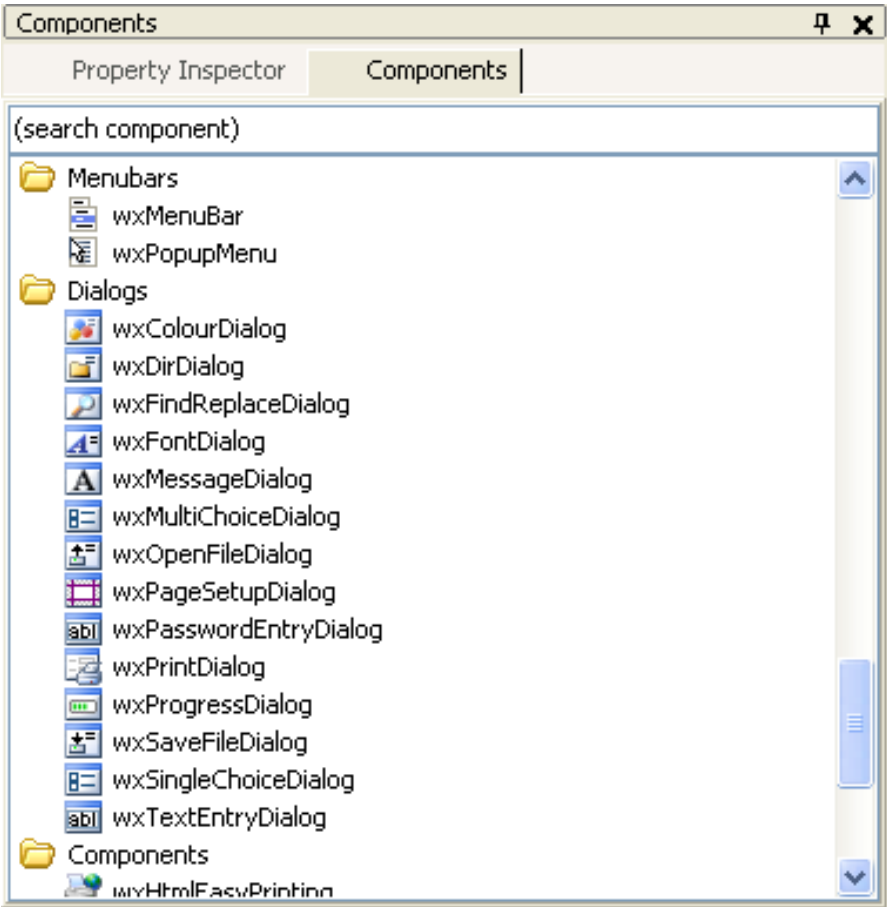


Fig. 5.29. Dialogue components.

|                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>wxYES_NO</code> -           | puts <i>Yes</i> and <i>No</i> buttons in the message box. It is recommended to use <code>wxCANCEL</code> always with this style. Otherwise the message box won't have a close button under <code>wxMSW</code> and the user will be forced to answer it.                                                                                                                                                                                                                             |
| <code>wxHELP</code> -             | puts a <i>Help</i> button to the message box. This button can have special appearance or be specially positioned if its label is not changed from the default one.                                                                                                                                                                                                                                                                                                                  |
| <code>wxNO_DEFAULT</code> -       | makes the <i>No</i> button default, can only be used with <code>wxYES_NO</code> .                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>wxCANCEL_DEFAULT</code> -   | makes the <i>Cancel</i> button default, can only be used with <code>wxCANCEL</code> .                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>wxYES_DEFAULT</code> -      | makes the <i>Yes</i> button default, this is the default behaviour and this flag exists solely for symmetry with <code>wxNO_DEFAULT</code> .                                                                                                                                                                                                                                                                                                                                        |
| <code>wxOK_DEFAULT</code> -       | makes the <i>OK</i> button default, this is the default behaviour and this flag exists solely for symmetry with <code>wxCANCEL_DEFAULT</code> .                                                                                                                                                                                                                                                                                                                                     |
| <code>wxICON_NONE</code> -        | displays no icon in the dialog if possible (an icon might still be displayed if the current platform mandates its use). This style may be used to prevent the dialog from using the default icon based on <code>wxYES_NO</code> presence as explained in <code>wxICON_QUESTION</code> and <code>wxICON_INFORMATION</code> documentation below.                                                                                                                                      |
| <code>wxICON_EXCLAMATION</code> - | displays exclamation or warning icon in the dialog.                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>wxICON_ERROR</code> -       | displays an error icon in the dialog.                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>wxICON_HAND</code> -        | displays an error symbol, this is a MSW-inspired synonym for <code>wxICON_ERROR</code> .                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>wxICON_QUESTION</code> -    | displays a question mark symbol. This icon is automatically used with <code>wxYES_NO</code> so it is usually unnecessary to specify it explicitly. This style is not supported for message dialogs under <code>wxMSW</code> when a task dialog is used to implement them (i.e. when running under <i>Windows Vista</i> or later) because <i>Microsoft</i> guidelines indicate that no icon should be used for routine confirmations. If it is specified, no icon will be displayed. |
| <code>wxICON_INFORMATION</code> - | displays an information symbol. This icon is used by default if <code>wxYES_NO</code> is not given so it is usually unnecessary to specify it explicitly.                                                                                                                                                                                                                                                                                                                           |
| <code>wxSTAY_ON_TOP</code> -      | makes the message box stay on the top of all other windows and not only just its parent (currently implemented only under MSW and GTK).                                                                                                                                                                                                                                                                                                                                             |
| <code>wxCENTRE</code> -           | centres the message box on its parent or on the screen if parent is not specified. Setting this style under MSW makes no differences as the dialog is always centered on the parent.                                                                                                                                                                                                                                                                                                |

Some useful methods are presented below.

`SetExtendedMessage (const wxString &extendedMessage)` - sets the extended message for the dialog. This message is usually an extension of the short message specified in the constructor or set with `SetMessage()`.

`wxTextEntryDialog` - this class represents the dialog that requests one-line text string from the user. `wxPasswordEntryDialog` is very similar to `wxTextEntryDialog`, but the input characters are masked. Some useful methods:

`GetMessage ()` - gets the message from the dialog.

`SetMessage (const wxString &message)` - sets the message shown by the dialog.

`GetValue()` - returns the text that the user has entered if the user has pressed *OK*, or the original value if the user has pressed *Cancel*.

`SetValue(const wxString& value)` - sets the default text value.

`wxSingleChoiceDialog` and `wxMultiChoiceDialog` - enable to use dialogues with radio buttons and to check boxes respectively. We can get the selected items using `GetSelections()` and `GetSelection()` methods.

`wxColourDialog` - represents the colour chooser dialog. We can get the colour value using `GetColourData()` method.

`wxDirDialog` - represents the directory chooser dialog. Some useful methods:

`GetPath()` - returns the default or user-selected path.

`GetMessage()` - returns the message that will be displayed on the dialog.

`SetMessage(const wxString& message)` - sets the message that will be displayed on the dialog.

`SetPath(const wxString& path)` - sets the default path.

`wxFontDialog` - represents the font chooser dialog. We can get the font properties using `GetFontData()` method.

`wxOpenFileDialog` and `wxSaveFileDialog` - enables to choose the file. Some useful methods:

`GetDirectory()` - returns the default directory.

`GetFilename()` - returns the default filename.

`GetFileNames (wxArrayString &filenames)` - fills the array filenames with the names of the files chosen.

`GetMessage ()` - returns the message that will be displayed on the dialog.

`GetPath ()` - returns the full path (directory and filename) of the selected file.

`GetPaths (wxArrayString &paths)` - fills the array paths with the full paths of the files chosen.

`SetDirectory (const wxString &directory)` - sets the default directory.

`SetFilename (const wxString &setfilename)` - sets the default filename.

`SetMessage (const wxString &message)` - sets the message that will be displayed on the dialog.

`SetPath (const wxString &path)` - sets the path (the combined directory and filename that will be returned when the dialog is dismissed).

`SetWildcard (const wxString &wildCard)` - sets the wildcard, which can contain multiple file types, for example: "BMP files (\*.bmp)—\*.bmp—GIF files (\*.gif)—\*.gif".

Let us create a sample program illustrating popup menu and different dialogues. The popup menu will appear after click of left mouse button on the form. The view of the program window during design stage is shown in Fig. 5.30.

Popup menu editor with menu items is shown in Fig. 5.31. Program window is shown in Fig. 5.32. Program code is presented in Table 5.19a, 5.19b, 5.19c, 5.19d, 5.19e, 5.19f.

**Menu bar.** Menu bar is a very convenient and frequently used control. We can use menu bar control while working with `wxWidgetsFrame` project type.

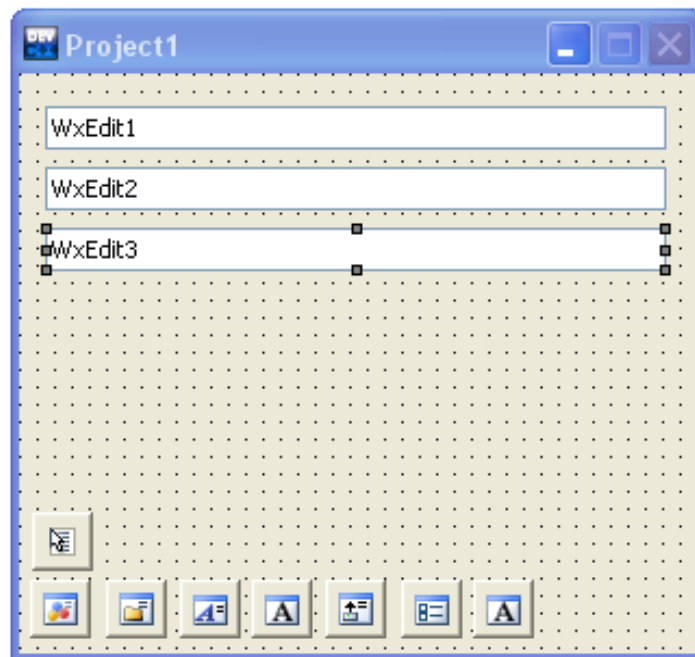


Fig. 5.30. Sample program design.

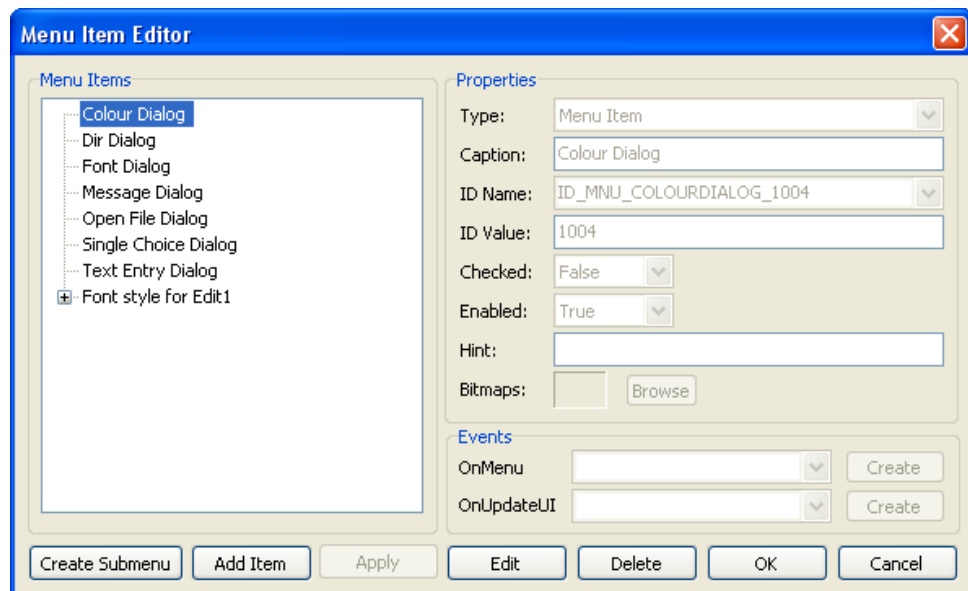


Fig. 5.31. Menu item editor with items.

A menu is a pull down (or popup while using popup menu) list of items, one of which may be selected before the menu goes away (clicking elsewhere dismisses the menu). Menus may be used to construct either menu bars or popup menus.

---

Table 5.19a. Program code.

---

```

/// @file      Project1Dlg.cpp
///-----

#include "Project1Dlg.h"
//-----
// Project1Dlg
//-----
// Add Custom Events only in the appropriate block.
// Code added in other places will be removed by wxDev-C++
//// Event Table Start
BEGIN_EVENT_TABLE(Project1Dlg,wxDialog)
    ///Manual Code Start
    Menu bar///Manual Code End

    EVT_CLOSE(Project1Dlg::OnClose)
    EVT_LEFT_DOWN(Project1Dlg::Project1DlgLeftDown)
    EVT_MENU(ID_MNU_COLOURDIALOG_1004 ,
              Project1Dlg::Mnucolourdialog1004Click)
    EVT_MENU(ID_MNU_DIRDIALOG_1005 ,
              Project1Dlg::Mnudirdialog1005Click)
    EVT_MENU(ID_MNU_FONTDIALOG_1006 ,
              Project1Dlg::Mnufontdialog1006Click)
    EVT_MENU(ID_MNU_MESSAGEDIALOG_1007 ,
              Project1Dlg::Mnumessagedialog1007Click)
    EVT_MENU(ID_MNU_OPENFILEDIALOG_1008 ,
              Project1Dlg::Mnuopenfiledialog1008Click)
    EVT_MENU(ID_MNU_SINGLECHOICEDIALOG_1009 ,
              Project1Dlg::Mnusinglechoicedialog1009Click)
    EVT_MENU(ID_MNU_TEXTENTRYDIALOG_1010 ,
              Project1Dlg::Mnutextentrydialog1010Click)
    EVT_MENU(ID_MNU_IALIC_1012 ,
              Project1Dlg::Mnuialic1012Click)
    EVT_MENU(ID_MNU_BOLD_1014 ,
              Project1Dlg::Mnubold1014Click)

    END_EVENT_TABLE()
//// Event Table End

```

---

Table 5.19b.

Program code.

---

```

void Project1Dlg::CreateGUIControls()
{
    //// GUI Items Creation Start
    WxTextEntryDialog1 = new wxTextEntryDialog( this, _(""),
                                                _("Please enter text"),
                                                _("") ,wxOK | wxCANCEL | wxCENTRE);

    wxString arrayStringFor_WxSingleChoiceDialog1;
    arrayStringFor_WxSingleChoiceDialog1.Add(_("Winter"));
    arrayStringFor_WxSingleChoiceDialog1.Add(_("Spring"));
    arrayStringFor_WxSingleChoiceDialog1.Add(_("Summer"));
    arrayStringFor_WxSingleChoiceDialog1.Add(_("Autumn"));
    WxSingleChoiceDialog1 = new wxSingleChoiceDialog(this, _(""),
                                                _("Single Choice List"),
                                                arrayStringFor_WxSingleChoiceDialog1, NULL);

    WxOpenFileDialog1 = new wxFileDialog(this, _("Choose a file"),
                                                _(""), _(""), _("%.*)" , wxFD_OPEN);

    WxMessageDialog1 = new wxMessageDialog(this, _(""), _("Message box"));

    WxFontDialog1 = new wxFontDialog(this);

    WxDirDialog1 = new wxDirDialog(this, _("Choose a directory"), _(""));

    WxPopupMenu1 = new wxMenu(_(""));
    WxPopupMenu1 -> Append (ID_MNU_COLOURDIALOG_1004,
                            _("Colour Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_DIRDIALOG_1005,
                            _("Dir Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_FONTDIALOG_1006,
                            _("Font Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_MESSAGEDIALOG_1007,
                            _("Message Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_OPENFILEDIALOG_1008,
                            _("Open File Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_SINGLECHOICEDIALOG_1009,
                            _("Single Choice Dialog"), _(""), wxITEM_NORMAL);
    WxPopupMenu1 -> Append (ID_MNU_TEXTENTRYDIALOG_1010,
                            _("Text Entry Dialog"), _(""), wxITEM_NORMAL);
    // continued to Table 5.19c

```

---

| Table 5.19c. | Program code.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <pre> // continued from Table 5.19b wxMenu *ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj = new wxMenu(); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -&gt; Append (ID_MNU_IALIC_1012,                                              _("Italic"), _(""), wxITEM_CHECK); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -&gt; Check (ID_MNU_IALIC_1012, false); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -&gt; Append (ID_MNU_BOLD_1014,                                              _("Bold"), _(""), wxITEM_CHECK); ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj -&gt; Check (ID_MNU_BOLD_1014, false); WxPopupMenu1 -&gt; Append(ID_MNU_FONTSTYLEFOREEDIT1_1011,                       _("Font style for Edit1"),                       ID_MNU_FONTSTYLEFOREEDIT1_1011_Obj);  WxColourDialog1 = new wxColourDialog(this); WxEdit3 = new wxTextCtrl(this, ID_WXEDIT3, _("WxEdit3"), wxPoint(12, 71),                         wxSize(286, 20),0, wxDefaultValidator, _("WxEdit3")); WxEdit2 = new wxTextCtrl(this, ID_WXEDIT2, _("WxEdit2"), wxPoint(12, 43),                         wxSize(286, 20),0, wxDefaultValidator, _("WxEdit2")); WxEdit1 = new wxTextCtrl(this, ID_WXEDIT1, _("WxEdit1"), wxPoint(12, 15),                         wxSize(286, 20),0, wxDefaultValidator, _("WxEdit1")); SetTitle(_("Project1")); SetIcon(wxNullIcon); SetSize(8,8,320,301); Center(); /////GUI Items Creation End } </pre> |

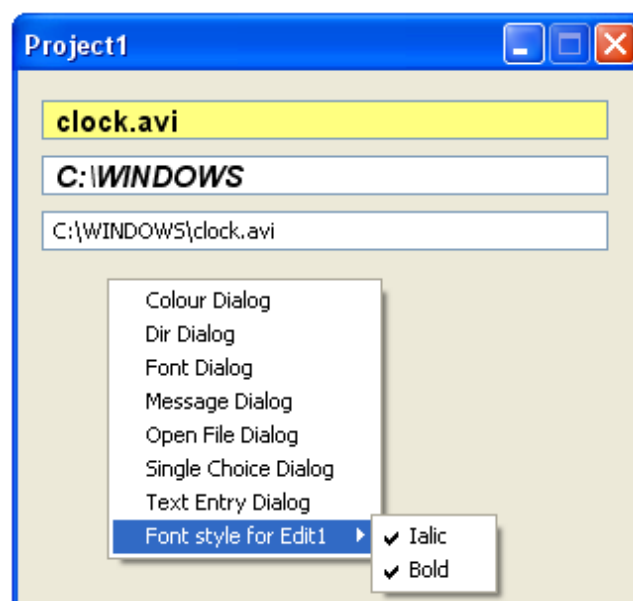


Fig. 5.32. Sample program during the execution stage.

Table 5.19d

Program code.

---

```

Project1Dlg::Project1Dlg(wxWindow *parent,
                        wxWindowID id,
                        const wxString &title,
                        const wxPoint &position,
                        const wxSize &size,
                        long style)
    : wxDialog( parent, id, title, position, size, style)
{
    CreateGUIControls();
}
Project1Dlg::~Project1Dlg()
{
}
void Project1Dlg::OnClose(wxCloseEvent& /*event*/)
{
    Destroy();
}
void Project1Dlg::Mnucolourdialog1004Click(wxCommandEvent& event)
{
    WxColourDialog1 -> ShowModal();
    WxEdit1 -> SetBackgroundColour(WxColourDialog1 ->
                                GetColourData().GetColour());
    WxEdit1 -> Refresh();
}
void Project1Dlg::Mnudirdialog1005Click(wxCommandEvent& event)
{
    WxDirDialog1 -> ShowModal();
    WxEdit1 -> SetValue(WxDirDialog1 -> GetPath());
}
void Project1Dlg::Mnufontdialog1006Click(wxCommandEvent& event)
{
    WxFontDialog1 -> ShowModal();
    WxEdit1 -> SetFont(WxFontDialog1 -> GetFontData().GetChosenFont());
}
void Project1Dlg::Mnumessagedialog1007Click(wxCommandEvent& event)
{
    WxMessageDialog1 -> SetMessage("This is a message dialogue.");
    WxMessageDialog1 -> ShowModal();
}
void Project1Dlg::Mnuopenfiledialog1008Click(wxCommandEvent& event)
{
    WxOpenFileDialog1 -> ShowModal();
    WxEdit1 -> SetValue(WxOpenFileDialog1 -> GetFilename());
    WxEdit2 -> SetValue(WxOpenFileDialog1 -> GetDirectory());
    WxEdit3 -> SetValue(WxOpenFileDialog1 -> GetPath());
}

```

---

| Table 5.19e | Program code.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|             | <pre> void Project1Dlg::Mnusinglechoicedialog1009Click(wxCommandEvent&amp; event) {     wxString ss;     WxSingleChoiceDialog1-&gt;ShowModal();     ss &lt;&lt; "It is " &lt;&lt; WxSingleChoiceDialog1 -&gt; GetStringSelection() &lt;&lt; " now";     WxEdit1 -&gt; SetValue(ss);     ss &lt;&lt; "The number of selected option is: "         &lt;&lt; WxSingleChoiceDialog1 -&gt; GetSelection();     WxEdit2 -&gt; SetValue(ss); }  void Project1Dlg::Mnutextentrydialog1010Click(wxCommandEvent&amp; event) {     wxString ss;     WxTextEntryDialog1 -&gt; ShowModal();     ss &lt;&lt; "Hi, " &lt;&lt; WxTextEntryDialog1 -&gt; GetValue();     WxEdit1 -&gt; SetValue(ss); }  void Project1Dlg::Mnuialic1012Click(wxCommandEvent&amp; event) {     if ((WxPopupMenu1 -&gt; IsChecked(1012))&amp;&amp;(WxPopupMenu1 -&gt; IsChecked(1014)))         WxEdit2 -&gt; SetFont(wxFont(11, wxSWISS, wxITALIC, wxBOLD, false));     else if (WxPopupMenu1 -&gt; IsChecked(1014))         WxEdit2 -&gt; SetFont(wxFont(11, wxSWISS, wxNORMAL, wxBOLD, false));     else if (WxPopupMenu1 -&gt; IsChecked(1012))         WxEdit2 -&gt; SetFont(wxFont(11, wxSWISS, wxITALIC, wxNORMAL, false));     else         WxEdit2 -&gt; SetFont(wxFont(11, wxSWISS, wxNORMAL, wxNORMAL, false)); } </pre> |

A menu item has an integer ID associated with it, which can be used to identify the selection, or to change the menu item in some way. A menu item with a special identifier -1 is a separator item and doesn't have an associated command but just makes a separator line appear in the menu. Menu items may be either normal items, check items or radio items.

Normal items do not have any special properties, while the check items have a boolean flag associated to them and they show a checkmark in the menu when the flag is set. `wxWidgets` automatically toggles the flag value when the item is clicked and its value may be retrieved using either `IsChecked` method of `wxMenu` or `wxMenuBar` itself or by using `wxEvent::IsChecked` when you get the menu notification for the item in question.

The radio items are similar to the check items except that all the other items in the same radio group are unchecked when one radio item is checked. The radio group is formed by a contiguous range of radio items, i.e. it starts at the first item of this kind and ends with the first item of a different kind (or the end of the menu).

Table 5.19f.

Program code.

---

```

void Project1Dlg::Mnubold1014Click(wxCommandEvent& event)
{
    if ((WxPopupMenu1 -> IsChecked(1012)) && (WxPopupMenu1 -> IsChecked(1014)))
        WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxBOLD, false));
    else if (WxPopupMenu1 -> IsChecked(1014))
        WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxBOLD, false));
    else if (WxPopupMenu1 -> IsChecked(1012))
        WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxITALIC, wxNORMAL, false));
    else
        WxEdit2 -> SetFont(wxFont(11, wxSWISS, wxNORMAL, wxNORMAL, false));
}
void Project1Dlg::Project1DlgLeftDown(wxMouseEvent& event)
{
    PopupMenu(WxPopupMenu1);
}

```

---

Notice, that because the radio groups are defined in terms of the item positions inserting or removing the items in the menu containing the radio items risks not to work correctly.

All menus, except the popup ones, must be created on the heap. All menus, attached to a menu bar or to another menu, will be deleted by their parent when it is deleted. As the frame menu bar is deleted by the frame itself, it means, that normally all menus used are deleted automatically.

We can use menu item editor (see Fig. 5.31) to create and manage menu bar items. Some useful menu bar methods:

`wxMenuItem* Append(int id, const wxString& item = "", const wxString& helpString = "", wxItemKind kind = wxITEM_NORMAL)` - adds a string item to the end of the menu.

`wxMenuItem* Append(int id, const wxString& item, wxMenu *subMenu, const wxString& helpString = "")` - adds a pull-right submenu to the end of the menu. Append the submenu to the parent menu after you have added your menu items, or accelerators may not be registered properly.

`wxMenuItem* Append(wxMenuItem* menuItem)` - adds a menu item object. This is the most generic variant of `Append()` method because it may be used for both items (including separators) and submenus. You can also specify various extra properties of a menu item in this way, such as bitmaps and fonts. Parameters are presented below:

|                           |                                                                                                                                                                    |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>id</code> -         | the menu command identifier;                                                                                                                                       |
| <code>item</code> -       | the string to appear on the menu item;                                                                                                                             |
| <code>menu</code> -       | pull-right submenu;                                                                                                                                                |
| <code>kind</code> -       | may be <code>wxITEM_SEPARATOR</code> , <code>wxITEM_NORMAL</code> , <code>wxITEM_CHECK</code> or <code>wxITEM_RADIO</code> ;                                       |
| <code>helpString</code> - | an optional help string associated with the item. By default, the handler for the <code>wxEVT_MENU_HIGHLIGHT</code> event displays this string in the status line; |

**menuItem** - a menuItem object. It will be owned by the wxMenu object after this function is called, so do not delete it yourself.

This command can be used after the menu has been shown, as well as on initial creation of a menu or menu bar. The item string for the normal menu items (not submenus or separators) may include the accelerator which can be used to activate the menu item from the keyboard. The accelerator string follows the item label and is separated from it by a TAB character ('\t'). Its general syntax is any combination of "CTRL", "ALT" and "SHIFT" strings (case doesn't matter) separated by either '-' or '+' characters and followed by the accelerator itself.

The accelerator may be any alphanumeric character, any function key (from F1 to F12) or one of the special characters listed in the table below (again, case doesn't matter):

|                 |                                  |
|-----------------|----------------------------------|
| DEL or DELETE   | <i>Delete key</i>                |
| INS or INSERT   | <i>Insert key</i>                |
| ENTER or RETURN | <i>Enter key</i>                 |
| PGUP            | <i>PageUp key</i>                |
| PGDN            | <i>PageDown key</i>              |
| LEFT            | <i>Left cursor arrow key</i>     |
| RIGHT           | <i>Right cursor arrow key</i>    |
| UP              | <i>Up cursor arrow key</i>       |
| DOWN            | <i>Down cursor arrow key</i>     |
| HOME            | <i>Home key</i>                  |
| END             | <i>End key</i>                   |
| SPACE           | <i>Space</i>                     |
| TAB             | <i>Tab key</i>                   |
| ESC or ESCAPE   | <i>Escape key (Windows only)</i> |

`wxMenuItem* AppendCheckItem(int id, const wxString& item, const wxString& helpString = "")` - adds a checkable item to the end of the menu.

`wxMenuItem* AppendRadioItem(int id, const wxString& item, const wxString& helpString = "")` - adds a radio item to the end of the menu. All consequent radio items form a group and when an item in the group is checked, all the others are automatically unchecked.

`wxMenuItem* AppendSeparator()` - adds a separator to the end of the menu.

`wxMenuItem* AppendSubMenu(wxMenu *submenu, const wxString& text, const wxString& help = wxEmptyString)` - adds the given submenu to this menu. Parameter `text` is the text, shown in the menu for it. Parameter `help` is the help string, shown in the status bar when the submenu item is selected.

`Break()` - inserts a break in a menu, causing the next appended item to appear in a new column.

`Check(int id, const bool check)` - checks or unchecks the menu item. Parameter `id` is the menu item identifier. If parameter `check` value is `TRUE`, the item will be checked, otherwise it will be unchecked.

`Delete(int id)` or `Delete(wxMenuItem *item)` - deletes the menu item from the menu. If the item is a submenu, it will not be deleted. Use `Destroy` if you want to delete a submenu. Parameter `id` is the id of the menu item to be deleted. Parameter `item` is the menu item to be deleted.

`Destroy(int id)` or `Destroy(wxMenuItem *item)` - deletes the menu item from the menu. If the item is a submenu, it will be deleted. Use *Remove* if you want to keep the submenu (for example, to reuse it later). Parameter `id` is the id of the menu item to be deleted. Parameter `item` is the menu item to be deleted.

`Enable(int id, const bool enable)` - enables or disables (greys out) a menu item. Parameter `id` is the menu item identifier. Parameter `enable` value should be `TRUE` to enable the menu item, `FALSE` to disable it.

`wxString GetHelpString(int id)` - returns the help string associated with a menu item. Parameter `id` is the menu item identifier.

`size_t GetMenuItemCount()` - returns the number of items in the menu.

`wxMenuItemList& GetMenuItems()` - returns the list of items in the menu. `wxMenuItemList` is a pseudo-template list class containing `wxMenuItem` pointers, see `wxList`.

`wxMenuItem* Insert(size_t pos, wxMenuItem *item)` or

`wxMenuItem* Insert(size_t pos, int id, const wxString& item = "", const wxString& helpString = "", wxItemKind kind = wxITEM_NORMAL)` - inserts the given item before the position `pos`. Inserting the item at position `GetMenuItemCount` is the same as appending it.

`wxMenuItem* InsertCheckItem(size_t pos, int id, const wxString& item, const wxString& helpString = "")` - inserts a checkable item at the given position.

`wxMenuItem* InsertRadioItem(size_t pos, int id, const wxString& item, const wxString& helpString = "")` - inserts a radio item at the given position.

`wxMenuItem* InsertSeparator(size_t pos)` - inserts a separator at the given position.

`bool IsChecked(int id)` - determines whether a menu item is checked. Parameter `id` is the menu item identifier.

`bool IsEnabled(int id)` - determines whether a menu item is enabled. Parameter `id` is the menu item identifier.

`wxMenuItem * Remove(int id)` or `wxMenuItem * Remove(wxMenuItem *item)` - removes the menu item from the menu but doesn't delete the associated C++ object. This allows to reuse the same item later by adding it back to the menu (especially useful with submenus). Parameter `id` is the identifier of the menu item to remove. Parameter `item` is the menu item to remove.

`void SetHelpString(int id, const wxString& helpString)` - sets an item's help string. Parameter `id` is the menu item identifier.

Parameter `helpString` is the help string to set.