OCTAVO
An Object Oriented GUI Framework

Federico de Ceballos
Universidad de Cantabria
federico.ceballos@unican.es

November, 2004

Abstract

This paper presents a framework for building Window applications using the Object Oriented tools available (in different forms) in various Forth compilers. Its aims are to simplify the programming of GUI applications and to isolate the user from the operating system internal details. As it would be expected in a Forth approach, the package tries to be as light and efficient as possible, while at the same time allowing those interested to understand what is taking place "under the bonnet".

INTRODUCTION

Octavo is a simple GUI framework for the Windows 2000 Operating System, to be used with a Forth system with minimal Object Oriented extensions.

It is based in the John English Window System [1], an Ada package aimed at novices.

The version presented here has been developed over SwiftForth [2], the system normally used by the author. However, the framework could be easily reproduced in other Forth systems, both commercial and public domain.

This paper concentrates on the requirements of a Forth system in order to be able to implement the package. It also gives an outlook of the functionality of Octavo.

USING CLASSES AND OBJECTS

In order to implement the Octavo package, an Object-Oriented Forth system is needed. However, there is not a common standard (not even a consensus) about what options an OO package should contain or how they should be used.

Most Forth systems can be considered as supersets of the ANS standard, and it is possible to write a complete OO package in ANS Forth (gForth [3] comes with three of them). But if the Forth system you use comes with a different one you are probably at ease with it and are not willing to try something new.

The SWOOP [4] package is used in the examples that follow, for no better reason that this is the package built into the system the author uses.

Defining classes

The following example shows how to define a class that can hold the information of a two-dimensional point.

```
class 2D-point
   variable x
   variable y

   : show             ." X = " x @ .  ." Y = " y @ . ;
   : set ( x y )      y !  x ! ;
   : get ( - x y )    x @  y @ ;
end-class
```

We could do a similar thing in order to work with three dimensional points. However, it is easier to take advantage of the fact that a `3D-point` is a `2D-point` with something added and something changed.

```
2D-point subclass 3D-point
   variable z

   : show               show  ." Z = " z @ . ;
   : set ( x y z )      z !  set ;
   : get ( - x y z )    get  z @ ;
end-class
```

Defining objects

The following example show how a couple of instances (one for each of the two classes defined) are created, and how messages are sent to them.

```
2D-point builds first
3D-point builds second

1 2   first  set  first  show  cr
3 4 5 second set  second show  cr

X = 1 Y = 2
X = 3 Y = 4 Z = 5
```

Pointers to objects

Some OO packages allow the user to define dynamic objects. A dynamic instance in built in dynamic memory and the user recieves a single pointer, without any type information. Even without dynamic objects, a function will sometimes receive the pointer to the object, so the type information will be lost.

It should be possible to send a message to an anonymous instance without specifying the class to which it belongs. The message dispatcher should ensure that the message arrives to the correct methos (late binding).

The following example creates a dynamic instance of class 3D-point, sends a show message to it and finally destroys the instance when it is no longer needed.

```
3D-point new
( point ) dup -> show
( point ) destroy

X = 0 Y = 0 Z = 0
```

THE WINDOW HIERARCHY

The available types of window are arranged in a class hierarchy, like this:

- `c-window`
  - `c-container`
    - **`c-frame`**
    - **`c-dialog`**
    - **`c-panel`**
    - **`c-menu`**
  - `c-control`
    - `c-text-control`
      - **`c-button`**
      - **`c-label`**
      - **`c-editbox`**
      - **`c-boolean`**
      - **`c-menuitem`**
      - **`c-checkbox`**
      - **`c-radio`**
    - `c-multiline`
      - **`c-listbox`**
      - **`c-combobox`**
      - **`c-memo`**
    - **`c-canvas`**

Those classes in **bold** are intended to be used directly. The rest are only used for building the hierarchy.

OTHER RESOURCES

Fonts

As the font used in the windows can be set be the user, a couple of functions are given in order to create and delete fonts.

Message boxes

The package allows the user to show information with several types of message box without the need of using an instance of a class.

Common dialogs

Apart from normal windows, there are classes to use the following predefined windows resources:

- `c-common-dialog`
  - **`c-colour-dialog`**
  - **`c-font-dialog`**
  - `c-file-dialog`
    - **`c-open-dialog`**
    - **`c-save-dialog`**

As before, only those classes in **bold** are intended to be used directly. The rest are only used for building the hierarchy.

FUTURE LINES OF WORK

The first objective is to complete the documentation of the package and to develop a set of commented examples. Ideally, these examples should cover all the funcionality of the code examples given in Programming Windows [5]. This is a widely used book, so that references to it can be made without having to delve a lot into the background.

At this moment, the package will be made available to SwiftForth users.

Later, the author intents to rewrite the package so that is can be used with other Forth systems.

REFERENCES

[1] John English. *John English's Window Library*. http://www.it.bton.ac.uk/staff/je/jewl/. 2000.

[2] *SwiftForth Reference Manual*. Forth Inc, 1999.

[3] Neal Crook et al. *gForth Manual*. 2003.

[4] Rick VanNorman. *SWOOP: Object-Oriented Programming in SwiftForth*. Forth Dimensions. Vol XX - 5.

[5] Charles Petzold. *Programming Windows*. Microsoft Press, 1999.

A Draft to the OCTAVO User Manual

1. RELATED UTILITIES

1.1. Message boxes

A message box is a dialog which can be used to notify the user of an error or other information, or to ask a yes/no question. There are five operations which display message boxes:

`message-box ( c-addr u )`

Shows a simple message.

`info-box ( c-addr u )`

Shows a message with an information icon.

`warning-box ( c-addr u )`

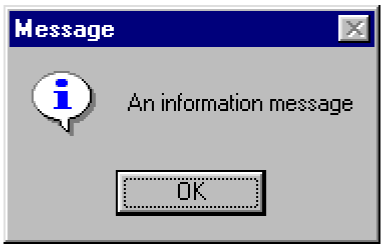
Shows a message with a warning icon.

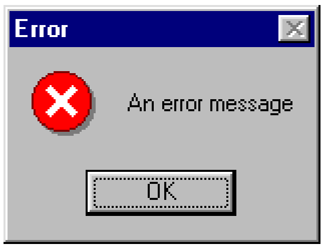
`error-box (c-addr u )`

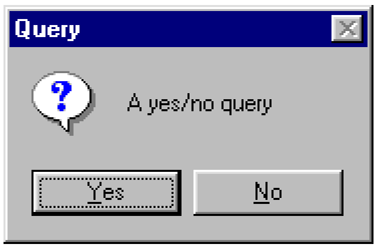
Shows a message with an error icon.

`question-box (c-addr u - flag )`

Shows a meesage with a query icon and the user is allowed to choose between "Yes" or "No". The function returns TRUE if the user chooses "Yes" and FALSE otherwise.







Something related to a message box is a sound.

`play-sound ( c-addr u )`

Plays the sound contained in the file of the given name.

2.1 Fonts

The user is allowed to create his or her own fonts and to use them in windows. Fonts should be destroyed when they are no longer needed.

The following operations apply to fonts:

`/font ( c-addr u size style - font )`

Creates a a font with the specified typeface name and point size. The other parameter can be NORMAL, BOLD, ITALIC or a combination of the last two.

`font/ ( font )`

Releases the resources asociated with the font.

2. WINDOWS

The operations defined for `c-window` can be applied to any window at all. These are as follows:

`show`

Makes the window visible.

`hide`

Makes the window invisible.

`focus`

Selects the window as having the input focus.

`visible? ( - flag )`

Returns TRUE if the window is visible, FALSE otherwise.

`@font ( - font )`

Returns the current font for the window.

`!font ( font )`

Sets the font used for displaying text in the window to the specified font.

`@origin ( - x y )`

Gets the position of the top left corner of the window.

`!origin ( x y )`

Sets the position of the top left corner of the window to the specified point.

`@size ( - x y )`

Gets the width and height of the window.

`!size ( x y )`

Sets the width and height of the window.

`@length ( - n )`

Returns the length of the text asociated with the window

`@text ( c-addr u - n )`

Copies the text asociated with the window to the given string. n is the length of the string.

`!text ( c-addr u )`

Sets the text asociated with the window to the given string.

`msg-create ( - u )`

This method is called when the window is being created. The application should return 0 if the creation process should continue.

`msg-close ( - u )`

This method is called when the window should be closed. The application should return 0 when it processes this message.

`msg-lbutton ( - u )`

This method is called when the user presses the left button of the mouse when it is over the client part of the window. The application should return 0 when it processes this message.

`msg-end ( - u )`

This is the last method called when the window is being destroyed. The application should return 0 when it processes this message.

`msg-command ( - u )`

This method is called when the window receives a command (probably because a child button has been pressed. The application should return 0 when it processes this message.

`msg-paint ( - u )`

This method is called when the window should be repainted. The application should return 0 when it processes this message.

`msg-rbutton ( - u )`

This method is called when the user presses the right button of the mouse when it is over the client part of the window. The application should return 0 when it processes this message.

`msg-focus ( - u )`

This method is called when the window has gained the keyboard focus. The application should return 0 when it processes this message.

`msg-size ( - u )`

This method is called after the window size has changed. The application should return 0 when it processes this message.

`free`

If the window is dynamic (i.e. it was created with a `new`) all its memory is deallocated.

`destruct`

The window is destroyed.

`construct`

The window is constructed with default origin and size. After this point, the window exists but is not visible.

The basic categories of window are as follows:

Containers: windows which can contain other windows.

Text controls: control windows which store a single line of text.

Boolean controls: text controls which can be in one of two states, checked or unchecked .

Multiline controls: control windows which store multiple lines of text.

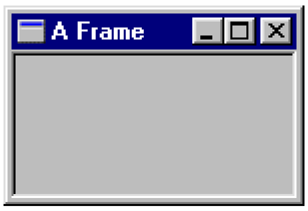
Canvases: these are controls which can be used as general-purpose drawing surfaces.

These are each described in more detail below.

3. CONTAINERS

Containers are windows which can contain other windows. There are four types of containers: frames, dialogs, panels and menus.

3.1 Frames



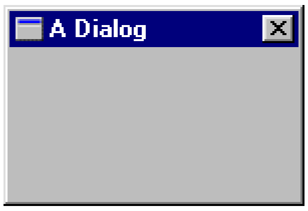
A frame (type `c-frame`) is a top-level window intended for use as the main window of an application. A frame is a container that other windows (controls etc.) can be attached to. Subwindows of a frame are inserted into the client area of the frame (the sunken area that it encloses) and all subwindow measurements are taken relative to the boundaries of the client area.

This class has the following constructor:

`construct`

Constructs the frame.

3.2 Dialogs



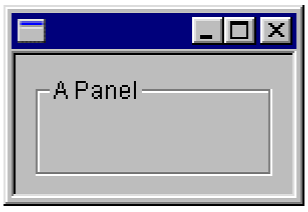
A dialog (type `c-dialog`) is a top-level window intended for use in modal interactions with the user. Dialogs are not normally visible except when they are being executed. Executing a dialog makes the dialog window visible in the centre of the screen and disables all other windows belonging to the application until a command is generated, either by closing the dialog window or by generating a command from a control enclosed by the dialog. Dialogs can be moved but unlike frames they cannot be resized, minimised or maximised.

This class has the following constructor:

`construct`

Constructs the dialog.

3.3 Panels



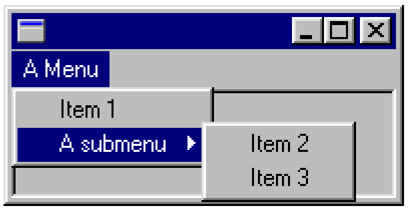
A panel (type `c-panel`) is a container intended for grouping controls together. A panel is not a top-level window and therefore must be enclosed by another container.

This class has the following constructor:

`construct ( parent )`

Constructs the control as a child of the parent window.

3.4 Menus



Menus (type `c-menu`) are containers for menu items. Menus can only be attached to frames or to other menus. A menu which is attached to a frame will appear on a menu bar above the parent's client area, and a menu attached to a menu will appear as a menu item with an arrowhead next to it, which will display a submenu when it is selected. The font used for displaying menus cannot be changed. Any attempt to change the font will be ignored, and the font will be reported as being the same as the parent frame's font. Similarly, attempts to hide a menu will be ignored.

This class has the following constructor:

`construct ( parent )`

Constructs the control as a child of the parent window.

4. CONTROLS

Controls are windows which provide for user interaction. They can be regarded as a visual representation of an internal variable whose value can be accessed and altered by the program. In most cases, the user of the program can also interact with controls to change their values. Some controls (buttons, menu items and canvases) can also generate command codes which will be returned as the result of the Next_Command function (described in the section on frames, above).

Several types of controls are provided: Text controls, boolean controls, multiline contols and canvases.

The following methods are available for controls:

`enable`

Enables the control.

`disable`

Disables de control. Disabled controls will not interact with the user, and are usually "grayed out" to indicate when they are disabled.

`enabled? ( - flag )`

Checks whether the control is enabled. All controls are enabled by default.

4.1 Text controls

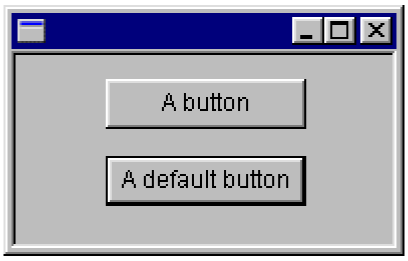
Text controls contain a single string of text which can be inspected and altered by the program. The types of text control available are as follows:

Buttons: a button is a rectangular control with a text label. When a button is pressed it generates a command code.

Labels: a label is a non-interactive text item which can be used to label other controls.

Editboxes: an editbox is a control which contains a single line of text which can be edited by the user.

4.1.1 Buttons



A button (type `c-button`) is a control that generates a command when it is pressed.

This class has the following constructor:

`construct ( command parent )`

> Constructs the control as a child of the parent window. When the menu item is selected, the parent will recieve the command code.

4.1.2 Labels

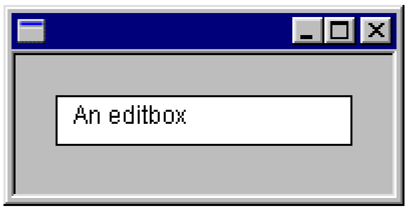
A label (type `c-label`) is a static control that is not interactive. Labels cannot be enabled or disabled, and any attempt to enable or disable a label is ignored.

This class has the following constructor:

`construct (parent )`

> Constructs the control as a child of the parent window.

4.1.3 Editboxes



An editbox (type `c-editbox`) is a control that allows you to interactively edit a single line of text. If the length of the line exceeds the width of the visible box, the text will scroll automatically as you move the cursor along it.

This class has the following constructor:

`construct (parent )`

> Constructs the control as a child of the parent window.

This class has the following new operation:

`modified ( - flag )`

> Returns TRUE if the user has modified the text in the edit box since the last time the function was called.

4.2 Boolean controls

Boolean controls are a subclass of text controls which also contain a Boolean state which can be inspected and altered by the program. The types of Boolean controls available are as follows:

> **Menu items**: these are controls which can only be attached to menus (see the section on menus, above). When selected they generate a command code. They can be set to be checked or unchecked in a similar way to checkboxes, as described below.
>
> **Checkboxes**: a checkbox is a labelled control with a checkable box to the left of the label. The box will change between the checked or unchecked states when it is selected.

Radiobuttons: a radiobutton is like a checkbox except that only one radiobutton in a group enclosed by a container can be checked at any one time. Selecting a radiobutton will change it to the checked state if it is unchecked, and at the same time any other radiobuttons in the same group will be unchecked.

Boolean controls support two extra operations:

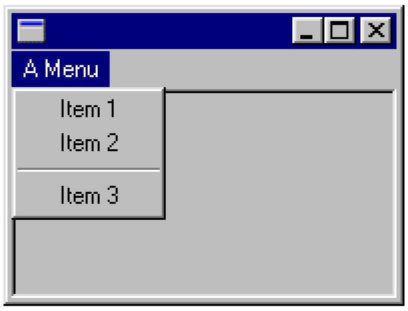
`@state ( - flag )`

Returns the state of the control (TRUE if checked, FALSE if unchecked).

`!state ( flag )`

Set the control to the specified state (checked if the flag is TRUE, false otherwise).

4.2.1 Menu items



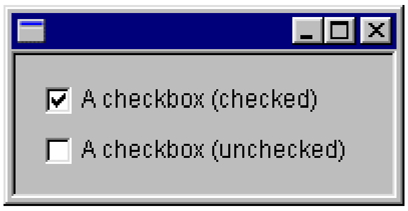
A menu item (type `c-menuitem`) is a control which can only be attached to a menu. As with menus, you cannot hide or change the font of a menu item, and the font will always be reported as being the same as the parent frame's font. Menu items can be checked or unchecked; when they are checked, a checkmark is displayed to the left of the text.

This class has the following constructor:

`construct ( command parent )`

Constructs the control as a child of the parent window. When the menu item is selected, the parent will recieve the command code.

4.2.2 Checkboxes



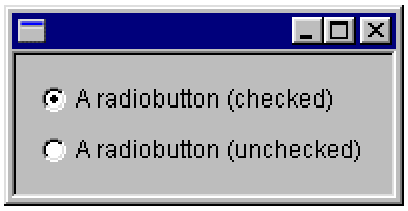
A checkbox (type `c-checkbox`) is a Boolean text control comprising a text label to the right of a box which can be checked or unchecked interactively.

This class has the following constructor:

`construct (parent )`

Constructs the control as a child of the parent window.

4.2.3 Radiobuttons



A radiobutton (type `c-radiobutton`) is a Boolean text control comprising a text label to the right of a box which can be checked or unchecked interactively. Unlike a checkbox, a radiobutton's state is related to the state of any other radiobuttons belonging to the same group. A group of radiobuttons is created by adding consecutive radiobuttons to the same container window. They must be added consecutively, because adding any other type of control to a window will end the group.

Selecting an unchecked radiobutton will set it to the checked state, and at the same time will uncheck any checked radiobutton enclosed by the same container. Selecting a checked radiobutton has no effect; the only way to uncheck a radiobutton is by checking another one belonging to the same group (or by calling `!State`).

This class has the following constructor:

`construct (parent )`

Constructs the control as a child of the parent window.

4.3 Multiline controls

A multiline control is similar to a text control except that it can contain multiple lines of text, one of which can be selected as the *current line*. Lines can be referenced using a line number, where line 1 is the first line. For convenience, you can also access the current line as line 0, assuming that a line has been selected as the current line.

The types of multiline control available are as follows:

Listboxes: a listbox contains a list of lines of text, and an individual line can be selected as the current line.

Comboboxes: a combobox is a combination of a listbox and an editbox. The listbox part is a drop-down list from which you can select an item to be copied into the editbox part of the control, or you can type directly into the editbox part of the control.

Memos: a memo is a multiline text editor suitable for general-purpose text editing.

The following extra functions are available for multiline controls.

`lines ( - #lines )`

Returns the number of linesof text in the control.

`select ( line )`

Selecst the specified line as the current line. If the line number is specified as zero, no line will be selected (i.e. there will no longer be a current line). The selected line is indicated visually in different ways by different controls.

`selected ( - line )`

Returns the number of the currently selected line (or zero if no line is selected).

`@length ( line - n )`

Returns the length of the text of the specified line. Line can be 0 in order to get the length of the current line.

`@text ( c-addr u line - n )`

Copies at most u charecters from the specified line to c-addr. The function returns the actual number of characters copied. Line can be 0 in order to use the current line.

`!text ( c-addr u line )`

Sets the text of the specified line. Line can be 0 in order to set the text of the current line.

`append ( c-addr u )`

Appends a new line of text to the end of the control.

`insert ( c-addr u line )`

Inserts a new line of text above the specified line number. Line can be 0 in order to insert above the currently selected line (or to the end of the control if there is no current line).

`delete ( line )`

Deletes the specified line. Line can be 0 in order to delete the currently selected line.

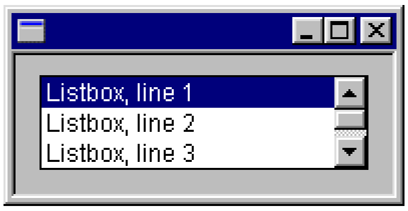
`clear`

Deletes all the lines from the control.

Individual types of multiline control interpret these operations in slightly different ways, as described below.

4.3.1 Listboxes

A listbox (type `c-listbox`) is a list of text lines which will display a vertical scroll bar if the number of lines exceeds the space available. You can select a line by clicking on it with the mouse or by calling select, in which case it becomes the control's current line and is highlighted on the display.



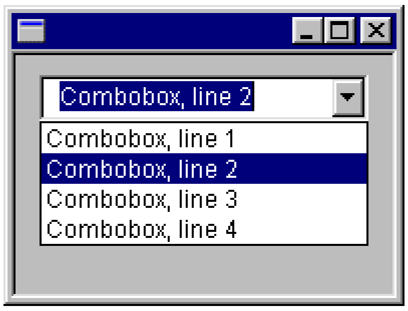
This class has the following constructor:

`construct ( parent )`

Constructs the control as a child of the parent window.

4.3.2 Comboboxes

A combobox (type `c-combobox`) is a combination of an editbox and a listbox. The listbox is not normally visible; it can be pulled down in a similar way to a menu by clicking on the button at the right of the editbox. A line can be selected as the current line by pulling down the listbox and selecting a line, or by calling select. The editbox always contains the text of the current line for the control, and its contents can always be accessed using 0 as the line number. If text is entered directly into the editbox which does not match any of the values in the listbox, it is treated as if there is no current line (`@Line` will return zero).



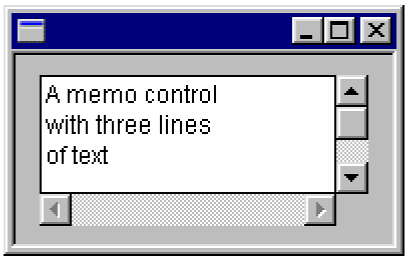
This class has the following constructor:

`construct (parent )`

Constructs the control as a child of the parent window.

4.3.3 Memos

A memo (type c-memo) is a general-purpose text editor. Unlike other multiline controls, the cursor can be positioned at an individual character rather than selecting an entire line, and a block of text can be selected which spans multiple lines (in which case the current position is taken to be the beginning of the selection).



This class has the following constructor:

`construct (parent )`

Constructs the control as a child of the parent window.

The operations specific to memos are as follows:

`cut`

Cuts the currently selected text in the memo (if any) to the system clipboard. This deletes the selected text if there is any.

`copy`

Copies the currently selected text in the memo (if any) to the system clipboard.

`paste`

Pastes the contents of the system clipboard to the memo, replacing the currently selected text. If no text is selection, the clipboard contents are inserted at the current position

`undo`

Undoes the user's last change to the contents of the memo. Note that the undo operation itself counts as a text change, so you can undo an undo operation.

`modified? ( - flag )`

Checks whether the user has modified the text in the memo since the last time this function was called.

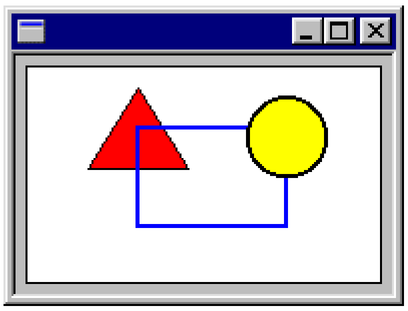
5. Canvases

A canvas (type `c-canvas`) is a general-purpose drawing surface with an asociated bitmap. Canvases are not described in further detail as the bitmap is modified with a package external to Octavo.

This class has the following constructor:

`construct ( parent )`

Constructs the control as a child of the parent window.



6. Common dialogs

The dialogs described here are provided as standard off-the-shelf dialog boxes for common interactions in much the same way as message boxes are provided for displaying simple messages. They are as follows:

Colour dialogs: dialogs to allow the user to select a colour.

Font dialogs: dialogs to allow the user to select a font.

File dialogs: dialogs to allow the user to select a file name, either for opening an input file or for creating an output file.

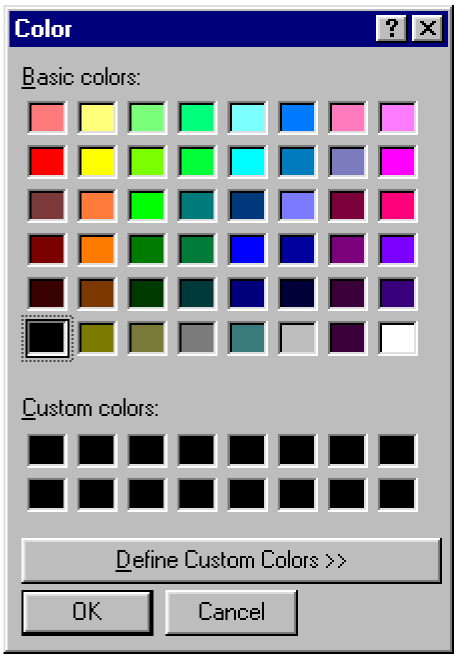
All common dialogs provide the following operation:

`execute ( - flag )`

Display the dialog and return TRUE if the user closes it by pressing the OK button, and FALSE otherwise.

6.1 Colour dialogs

A colour dialog (type `c-colour-dialog`) provides a palette of standard colours that you can choose from, or you can add your own custom colours to the palette. A colour dialog looks like this:



This class has the following constructor:

`construct`

Constructs the control.

The class has these additional methods:

`set ( colour )`

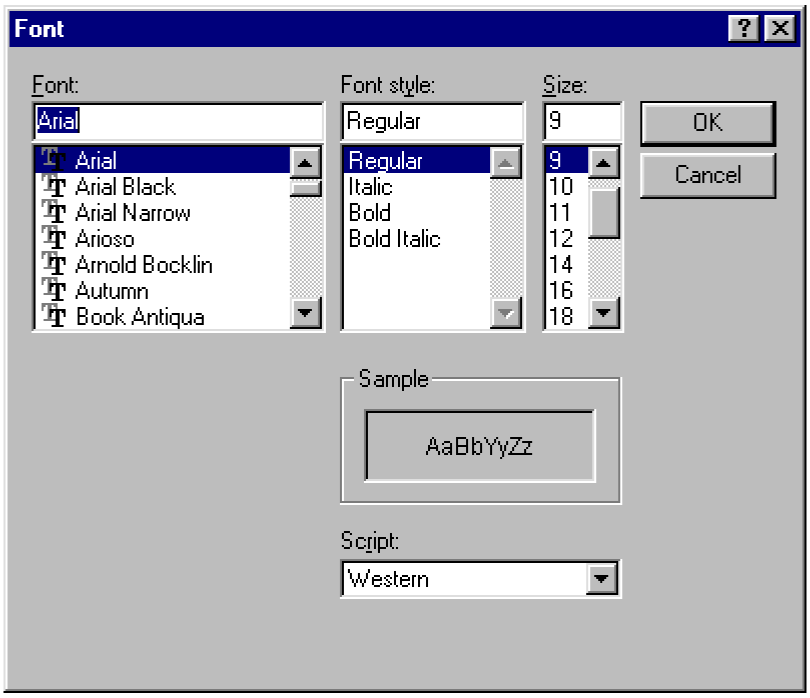
Store the specified colour in the dialog. This colour will be selected when the dialog is executed.

`get ( - colour )`

Return the colour stored in the dialog.

6.2 Font dialogs

A font dialog (type `c-font-dialog`) lets you select a font by selecting the desired name, size and style from a list of available fonts. A font dialog looks like this:



This class has the following constructor:

`construct`

Constructs the control.

The following operations apply to a `c-font-dialog`:

`set ( font )`

Stores the font in the dialog. The font will be selected when the dialog is executed.

`get ( - font )`

Returns the font stored in the dialog.

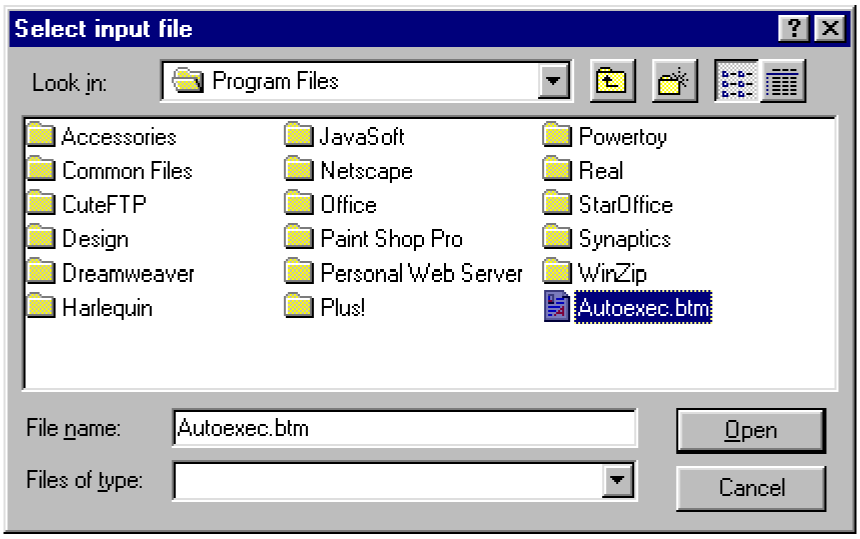
6.3 File dialogs

File dialogs allow you to select a filename to be opened for input or output. There are two types of file dialog available:

Open dialogs (type `c-open-dialog`): select a filename for an input file.

Save dialogs (type `c-save-dialog`): select a filename for an output file.

Both types of dialog look like this:



This is an Open dialog; the Save dialog looks exactly the same except that the button labelled "Open" is replaced by a button labelled "Save". They are functionally identical except for the types of file that can be selected. In an Open dialog, only existing files can be selected; in a Save dialog, new names can be typed in directly.

This classes have the following constructor:

`construct`

Constructs the control.

The following operations apply to these controls:

`set ( c-addr u )`

Store the specified filename in the dialog. This filename will be selected when the dialog is executed.

`get (c-addr u - n )`

Copies the filename stored in the dialog to the given string. `n` is the length of the string.

`add_filter ( c-addr1 u1 c-addr2 u2 )`

Add a filename filter to the dialog. Only files which match the selected filter will be displayed when the dialog is executed. More than one filter can be associated with a single dialog, and the user can choose between filters while the dialog is executing. The text `c-addr1 u1` is a description of the selected files, e.g. "Bitmap files (*.bmp)", while the text `c-addr3 u2` is a file specification (e.g. "*.bmp") or a list of file specifications separated by semicolons (e.g. "*.bmp;*.rle").

`!directory ( c-addr u )`

Set the initial directory for the dialog. If a directory with the specified name exists, it will be the initial directory displayed when the dialog is executed. If another directory is selected when the dialog is executed, it will be used as the initial directory the next time the dialog is executed unless this function is called again.