FLUID for FLTK 1.5.0 User Manual

By F. Costantini, M. Melcher,
A. Schlosser, B. Spitzak, and M. Sweet.

Generated by Doxygen 1.12.0

March 10, 2025

Git revision f99fdcdc71a7c111a370818ca7d06761a2e0e5b5

Chapter 1

Introduction

Figure 1.1 FLUID

FLUID, short for Fast Light User Interface Designer, is a graphical editor capable of generating C++ source code and header files ready for compilation. These files ultimately create an FLTK based graphical user interface for an application.

The FLTK programming manual is available at `https://www.fltk.org/documentation.php`.

This manual provides instructions on launching FLUID as a command line tool and integrating `.fl` project files into the application build process. FLTK utilizes *CMake*, but other build systems and IDEs capable of running external tools can also build applications based on FLUID.

The majority of the manual focuses on using FLUID as an interactive GUI design tool. It covers an overview of windows, menu items, and dialog boxes, detailing how to create visually appealing and consistent user experiences through drag and drop functionality, a "what you see is what you get" editor, and alignment tools. The Settings Dialog will detail the process of initiating a new project, creating an alignment template, and incorporating internationalization.

Several tutorials will explain how to generate small apps in FLUID alone, and how to create more complex user interfaces, followed by some advanced subjects like creating integrated reusable widget classes.

The appendices contain additional technical information for reference.

1.1 Workflow

FLUID stores user interface designs within `.fl` project files. These files are transformed into a binary application through a multi-step process. Initially, FLUID converts `.fl` files into C++ source and header files. Subsequently, these files are compiled into object files, which are then linked with other object files to form an executable binary. FLUID-generated header files give access to UI elements from other C++ modules within the project. FLUID can

also generate forward declarations to variables and callback functions that are defined and implemented in other C++ modules.
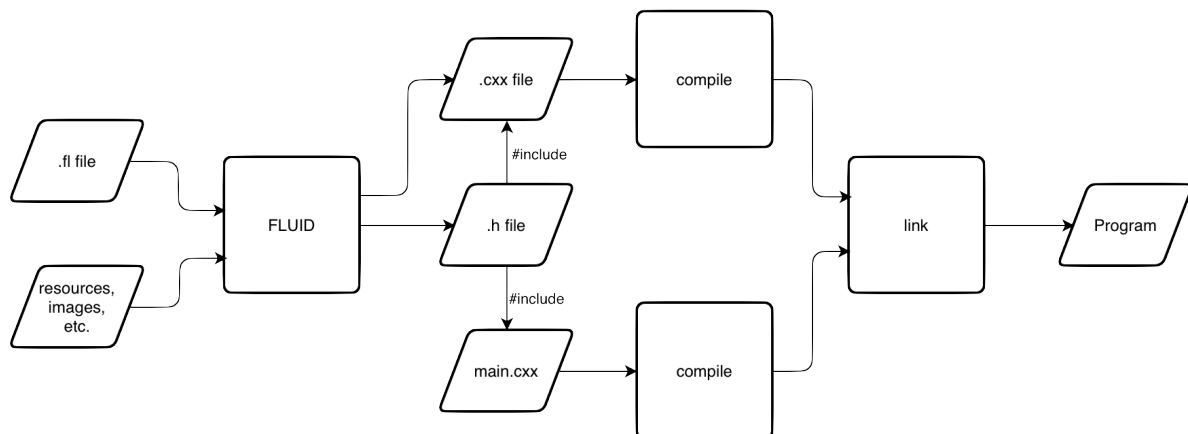
Figure 1.2 FLUID Workflow

Small applications can be fully designed and developed with FLUID alone. Users have the option to include shell scripts in FLUID projects, enabling them to directly call compilers and linkers to produce the binaries.

For medium-sized projects, a build system such as *CMake* or an IDE with integrated build setup is recommended. FLUID in interactive mode can pre-generate C++ code files for direct compilation by the IDE.

In larger projects, FLUID projects frequently reference external resources such as graphics, binary data, and internationalized text. In such scenarios, it is very useful to distribute the `.fl` project files instead of prebuilt source files. FLUID in command-line mode can then be called as an external tool, dynamically generating C++ source code from all external resources at build time.

Chapter 2

Command Line

FLUID can be used in interactive and in command line mode. If launched with `-c`, followed by a project filename, FLUID will convert the project file into C++ source files without ever opening a window (or opening an X11 server connection under Linux/X11). This makes FLUID a great command line tool for build processes with complex project files that reference external resources. For example, an image referenced by a `.fl` file can be modified and recompiled into the application binary without the need to reload it in an interactive FLUID session.

2.1 Command Line Options

To launch FLUID in interactive mode from the command line, you can give it an optional name of a project file. If no name is given, it will launch with an empty project, or with the last open project, if so selected in the application setting dialog.

The ampersand `&` is optional on Linux machines and lets FLUID run in its own new process, giving the shell back to the caller.

```
fluid filename.fl &
```

If the file does not exist you will get an error pop-up, but if you dismiss it you will be editing a blank file of that name.

FLUID understands all of the standard FLTK switches before the filename:

```
-display host:n.n
-geometry WxH+X+Y
-title windowtitle
-name classname
-iconic
-fg color
-bg color
-bg2 color
-scheme schemename
```

2.2 Compile Tool Options

FLUID can also be called as a command-line only tool to create the `.cxx` and `.h` file from a `.fl` file directly. To do this type:

```
fluid -c filename.fl
```

This is the same as the menu **File** > **Write Code...** . It will read the `filename.fl` file and write `filename.↩cxx` and `filename.h`. Any leading directory on `filename.fl` will be stripped, so they are always written to the current directory. If there are any errors reading or writing the files, FLUID will print the error and exit with a

non-zero code. You can use the following lines in a Makefile to automate the creation of the source and header files:

```
my_panels.h my_panels.cxx: my_panels.fl
        fluid -c my_panels.fl
```

Most versions of "make" support rules that cause `.fl` files to be compiled:

```
.SUFFIXES: .fl .cxx .h
.fl.h .fl.cxx:
        fluid -c $<
```

Check `README.CMake.txt` for examples on how to integrate FLUID into the `CMake` build process.

If you use

```
fluid -cs filename.fl
```

FLUID will also write the "strings" for internationalization into the file 'filename.txt', 'filename.po', or 'filename.msg', depending on the chosen type of i18n (menu: 'File/Write Strings...').

Finally there is another option which is useful for program developers who have many `.fl` files and want to upgrade them to the current FLUID version. FLUID will read the `filename.fl` file, save it, and exit immediately. This writes the file with current syntax and options and the current FLTK version in the header of the file. Use

```
fluid -u filename.fl
```

to 'upgrade' `filename.fl`. You may combine this with `-c` or `-cs`.

Note

> All these commands overwrite existing files w/o warning. You should particularly take care when running `fluid -u` since this overwrites the original `.fl` project file.

2.3 Windows Specifics

FLTK uses Linux-style forward slashes to separate path segments in file names. When running on Windows, FLUID will understand Microsoft drive names and backward slashes as path separators and convert them internally into forward slashes.

Under Windows, binaries can only be defined either as command line tools, or as interactive apps. FLTK generates two almost identical binaries under Windows. `fluid.exe` is meant to be used in interactive mode, and `fluid-cmd.exe` is generated for the command line. Both tools do exactly the same thing, except `fluid-cmd.exe` can use stdio to output error messages.

Chapter 3

Interactive Mode

In interactive mode, FLUID allows users to construct and modify their GUI design by organizing widgets hierarchically through drag-and-drop actions. The project windows provide a live preview of the final UI layout. Most widget attributes can be adjusted in detail using the Widget Properties Panel.

Users can also incorporate C++ coding elements such as functions, code blocks, and classes. FLUID supports the integration of external sources, for example images, text, or binary data, by embedding them directly into the generated source code.
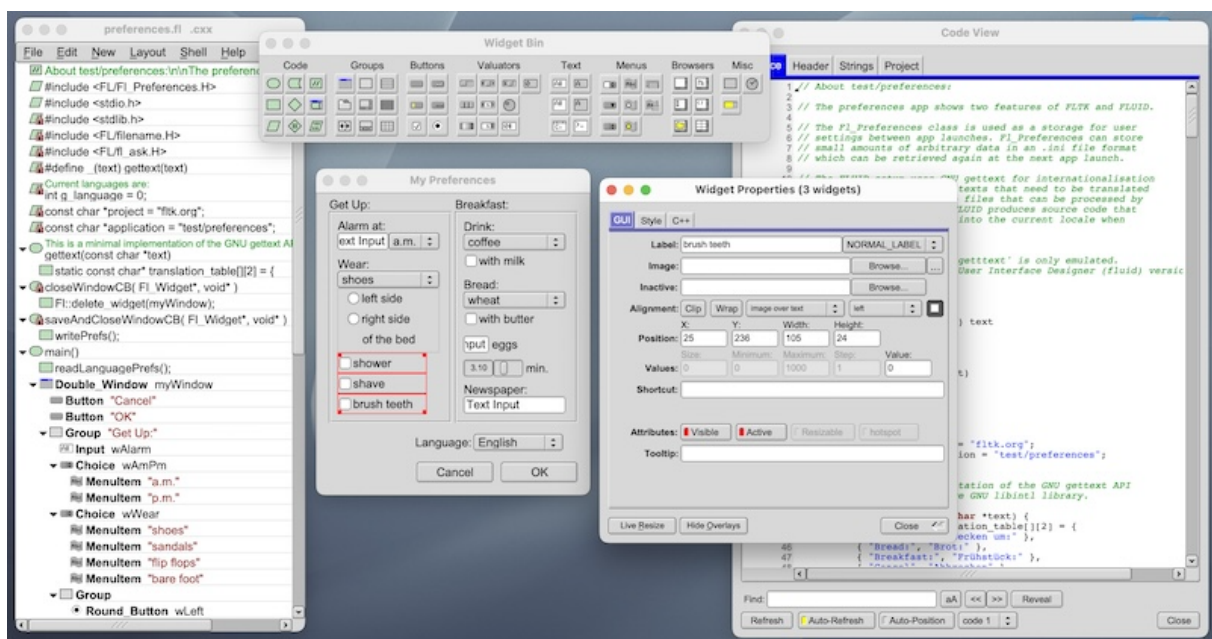
Figure 3.1 FLUID Overview

A typical FLUID session manages the widget hierarchy in the main application window on the left. The project file name is shown in the title bar. In the example above, we edit the FLTK *Preferences* example file `test/preferences.fl`.

See also

> Main Application Window

The layout editor window left of center, titled "My Preferences" shows the GUI design as it will be generated by the C++ source file. The widgets "shower", "shave", and "brush teeth" are shown in their selected state and can now be resized or moved by grabbing any of the red boxes around the selection frame.

See also

> Layout Editor Window

To the right, we have the "Widget Properties" panel. We can use this to edit many more parameters of the selected widget. If multiple widgets are selected, changes are applied to all of them where appropriate.

See also

> Widget Properties Panel

The Widget Bin at the top is an optional tool bar (**Edit** > **Show Widget Bin**, Alt-B). It gives quick access to all widget and code types. Widgets can be added by clicking onto the tool, or by dragging them out of the tool box into the layout editor.

See also

> Widget Bin Panel

The optional panel on the right shows a live source code preview. This is very helpful for verifying how changes in the GUI will be reflected in the generated C++ code (**Edit** > **Show Code View**, Alt-C).

See also

> Code View Panel
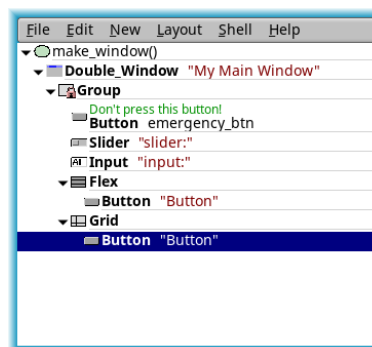
Chapter 4

Main Application Window

Figure 4.1 Main Application Window

A FLUID project is a hierarchy of nodes. Each node holds information to generate C++ source code which in turn generates the user interface that is created in the layout editor windows. Projects usually define one or more functions. These functions can generate one or more FLTK windows and all the widgets that go inside those windows.

The FLUID Main Window is split into three parts. The title bar shows the status of the source and project files. The menu bar provides a wealth of menu items for all major actions in FLUID. The biggest part of the app window is the widget browser, a tree structure that lists every code node and widget in the project.

4.1 Title Bar

Figure 4.2 Title Bar

The title bar shows the status of the project file, *function_panel.fl* in this case, followed by an asterisk if the project was changed after it was saved. If the asterisk shows, FLUID will ask the user to save changes before closing the project, loading another project, or starting a new one. Pressing `Ctrl-S` will save the project and make the asterisk disappear.

The *.cxx* in the title bar reflects the status of header and source files in relation to the project. A trailing asterisk indicates that the project and code files differ. Pressing `Ctrl-Shift-C` to write the code files will make this asterisk go away.

Note

   FLUID currently supports only one open project at a time.
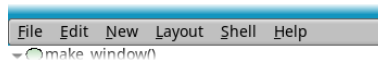
4.2   Application Menu Bar

Figure 4.3 Main Menu

The menu bar is the true control center of FLUID. All actions start here.

The *File* menu offers the common file operation for FLUID projects. Projects can be loaded, merged, and saved. *Print* will print a snapshot of all open project windows. The *New From Template* item opens a dialog that provides access to a small number of sample projects. More projects can be added using *Save as Template*.

Use *Write Code* to write the header and source code files, and *Write Strings* to write the translation file if one of the internationalization options is active.

The *Edit* menu is mainly used to manipulate widgets within the widget tree. The bottom entries toggle various dialogs and pop up the settings panel.

The *New* menu holds a list of all widgets that can be used in FLUID. They are grouped by functionality, very similarly to the widget bin. New widgets are added inside or right after the selected widget. If the parent widget is not compatible, FLUID tries to find another location for the widget. If that also fails, FLUID will pop up a dialog, describing the required parent type.

The *Layout* menu is used to adjust the position and size of widgets in relation to each other.

The *Shell* menu gives quick access to user definable shell scripts. Note that scripts can be stored inside `.fl` project files.

See also

   [The Main Menu](#)

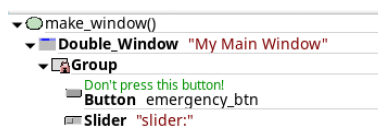## 4.3   Widget Tree View

Figure 4.4 Widget Browser

Widgets are stored in a hierarchy. You can open and close a level by clicking the "triangle" at the left of a widget. The leftmost widgets are the *parents*, and all the widgets listed below them are their *children*. Parents don't have to have any children.

The top level of the hierarchy is composed of *functions* and *classes*. Each of these will produce a single C++ public function or class in the output `.cxx` file. Calling the function or instantiating the class will create all of the child widgets.

The second level of the hierarchy contains the *windows*. Each of these produces an instance of class Fl_Window.

Below that are either *widgets* (subclasses of Fl_Widget) or *groups* of widgets (including other groups). Plain groups are for layout, navigation, and resize purposes. *Tab groups* provide the well-known file-card tab interface.

Widgets are shown in the browser by either their *name* (such as "Button emergency_btn" in the example), or by their *type* and *label* (such as "Double_Window "My Main Window"").

You *select* widgets by clicking on their names, which highlights them (you can also select widgets from any displayed window). You can select many widgets by dragging the mouse across them, or by using Shift+Click to toggle them on and off. To select no widgets, click in the blank area under the last widget. Note that hidden children may be selected even when there is no visual indication of this.

You *open* widgets by double-clicking on them, or (to open several widgets you have picked) by typing the F1 key. A control panel will appear so you can change the widget(s).

Nodes are moved within their group using `F2` and `F3`. They can be grouped and ungrouped with `F7` and `F8`, and relocated by selecting them and using cut, copy, and paste.

Every line in the browser has the same basic format. The level of indentation reflects the depth of a node within the tree.

The triangle appears only in front of nodes that can have children. If it is white, the group has no children. If it is black, there is at least one child. If the triangle points to the right, the children are hidden in the tree view. Click the triangle to reveal all children.

The icon to the right is a small representation of the base type of the node. Widgets are gray, windows have a blue title bar, and functional nodes are green. If the widget is static or private, a padlock icon will appear in the bottom right corner of the type icon.

The content of text fields depends on the node type. If a comment is set, it appears in green over the text. Widgets combine their type (bold black) and label text (red), or their C++ name in black (not bold).

All colors and font styles can be customized in the User tab of the Settings panel.

4.4 The Main Menu

The "New" menu of the main menu bar is duplicated as a pop-up menu on any layout editor window. The shortcuts for all the menu items work in any window. The menu items are:

File > **New (Ctrl+n)**: Close the current project and start a new, empty project.

File > **Open... (Ctrl+o)**: Discard the current editing session and read in a different `.fl` project file. You are asked for confirmation if you have changed the current file.

FLUID can also read `.fd` files produced by the Forms and XForms "fdesign" programs. It is best to *File* > *Merge* them instead of opening them. FLUID does not understand everything in a `.fd` file, and will print a warning message on the controlling terminal for all data it does not understand. You will probably need to edit the resulting setup to fix these errors. Be careful not to save the file without changing the name, as FLUID will write over the `.fd` file with its own format, which fdesign cannot read!

File > **Insert... (Ctrl+i)**: Insert the contents of another `.fl` file without changing the name of the current `.fl` file. All the functions (even if they have the same names as the current ones) are added, and you will have to use cut/paste to put the widgets where you want.

File > **Save (Ctrl+s)**: Write the current data to the `.fl` file. If the file is unnamed then FLUID will ask for a filename.

File > **Save As... (Ctrl+Shift+S)**: Ask for a new filename and save the file.

File > **Save A Copy...**: Save a copy of the `.fl` data to a different file.

File > **Revert...**: Revert the `.fl` data to the previously saved state.

File > **New From Template...**: Create a new user interface design from a previously saved template. This can be useful for including a predefined enterprise copyright message for projects, or for managing boilerplate code for repeating project code.

File > **Save As Template...**: Save the current project as a starting point for future projects.

File > **Print... (Ctrl-P)**: Generate a printout containing all currently open windows within your project.

File > **Write Code (Ctrl+Shift+C)**: Write the GUI layout as a `.cxx` and `.h` file. These are exactly the same as the files you get when you run FLUID with the `-c` switch.

The output file names are the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".h" or ".cxx" appended.

File > **Write Strings (Ctrl+Shift+W)**: Write a message file for all of the text labels and tooltips defined in the current file.

The output file name is the same as the `.fl` file, with the leading directory and trailing ".fl" stripped, and ".txt", ".po", or ".msg" appended depending on the Internationalization Mode.

File > **Quit (Ctrl+q)**: Exit FLUID. You are asked for confirmation if you have changed the current file.

Edit > **Undo (Ctrl+z)** and **Redo (Shift+Ctrl+z)**: FLUID saves the project state for undo and redo operations after every major change.

Edit > **Cut (Ctrl+x)**: Delete the selected widgets and all of their children. These are saved to a "clipboard" file and can be pasted back into any FLUID window.

Edit > **Copy (Ctrl+c)**: Copy the selected widgets and all of their children to the "clipboard" file.

Edit > **Paste (Ctrl+v)**: Paste the widgets from the clipboard file.

If the widget is a window, it is added to whatever function is selected, or contained in the current selection.

If the widget is a normal widget, it is added to whatever window or group is selected. If none is, it is added to the window or group that is the parent of the current selection.

To avoid confusion, it is best to select exactly one widget before doing a paste.

Cut/paste is the only way to change the parent of a widget.

Edit > **Duplicate (Ctrl-u)**: Duplicate all currently selected widgets and insert the duplicates after the last selected widget.

Edit > **Delete**: Delete all selected widgets.

Edit > **Select All (Ctrl+a)**: Select all widgets in the same group as the current selection.

If they are all selected already then this selects all widgets in that group's parent. Repeatedly typing `Ctrl+a` will select larger and larger groups of widgets until everything is selected.

Edit > **Properties... (F1 or double click)**: Display the current widget in the widgets panel. If the widget is a window and it is not visible then the window is shown instead.

Edit > **Sort**: Sort the selected widgets into left to right, top to bottom order. You need to do this to make navigation keys in FLTK work correctly. You may then fine-tune the sorting with "Earlier" and "Later". This does not affect the positions of windows or functions.

Edit > **Earlier (F2)**: Move all of the selected widgets one earlier in order among the children of their parent (if possible). This will affect navigation order, and if the widgets overlap it will affect how they draw, as the later widget is drawn on top of the earlier one. You can also use this to reorder functions, classes, and windows within functions.

Edit > **Later (F3)**: Move all of the selected widgets one later in order among the children of their parent (if possible).

Edit > **Group (F7)**: Create a new Fl_Group and make all the currently selected widgets children of that group.

Edit > **Ungroup (F8)**: Move the selected children of a group out of the group and up one level in the hierarchy. If all children of a group are selected and moved, the remaining empty group is deleted.

Edit > **Show or Hide Overlays (Ctrl+Shift+O)**: Toggle the display of the red overlays off, without changing the selection. This makes it easier to see box borders and how the layout looks. The overlays will be forced back on if you change the selection.

Edit > **Show or Hide Guides (Ctrl+Shift+G)**: Guides can be used to arrange a widget layout easily and consistently. They indicate preferred widget positions and sizes with user definable margins, grids, and gap sizes. See the "Layout" tab in the "Settings" dialog, Layout Preferences.

This menu item enables and disables guides and the snapping action when dragging widgets and their borders.

Edit > **Show or Hide Restricted (Ctrl+Shift+R)**: The behavior of overlapping widgets in FLTK is undefined. By activating this button, a hatch pattern is shown, highlighting areas where restricted or undefined behavior may occur.

Edit > **Show or Hide Widget Bin (Alt+B)**: The widget bin provides quick access to all widget types supported by FLUID. Layouts can be created by clicking on elements in the widget bin, or by dragging them from the bin to their position within the layout. This button shows or hides the widget bin.

Edit > **Show or Hide Code View (Alt+C)**: Shows or hide the source code preview window. Any changes to the layout or code in the layout editor can be previewed and verified immediately in the Code View window.

Edit > **Settings... (Alt+p)**: Open the application and project settings dialog: Settings Dialog

New > **Code** > **Function**: Create a new C function. You will be asked for a name for the function. This name should be a legal C++ function template, without the return type. You can pass arguments which can be referred to by code you type into the individual widgets.

If the function contains any unnamed windows, it will be declared as returning an Fl_Window pointer. The unnamed window will be returned from it (more than one unnamed window is useless). If the function contains only named windows, it will be declared as returning nothing (`void`).

It is possible to make the `.cxx` output be a self-contained program that can be compiled and executed. This is done by deleting the function name so `main(argc,argv)` is used. The function will call `show()` on all the windows it creates and then call `Fl::run()`. This can also be used to test resize behavior or other parts of the user interface.

You can change the function name by double-clicking on the function.

See also

> [Function and Method Panel](#)

New > **Group** > **Window**: Create a new Fl_Window widget. The window is added to the currently selected function, or to the function containing the currently selected item. The window will appear, sized to 480x320. You can resize it to whatever size you require.

The widget panel will also appear and is described later in this chapter.

New > **...**: All other items on the New menu are subclasses of `Fl_Widget`. Creating them will add them to the currently selected group or window, or the group or window containing the currently selected widget. The initial dimensions and position are chosen by copying the current widget, if possible.

When you create the widget you will get the widget's control panel, which is described later in this chapter.

Layout > **Align** > **...**: Align all selected widgets to the first widget in the selection.

Layout > **Space Evenly** > **...**: Space all selected widgets evenly inside the selected space. Widgets will be sorted from first to last.

Layout > **Make Same Size** > **...**: Make all selected widgets the same size as the first selected widget.

Layout > **Center in Group** > **...**: Center all selected widgets relative to their parent widget

Layout > **Synchronized Resize**: If unchecked, groups and windows can be resized without resizing their children. If set, the layout of the children is changed according to their `resize()` settings. Try **Live Resize** to verify the effects before permanently modifying the layout.

Layout > **Grid and Size Settings... (Ctrl+g)**: Display the grid settings panel. See [Layout Preferences](#) .

This panel controls dimensions that all widgets snap to when you move and resize them, and for the "snap" which is how far a widget has to be dragged from its original position to actually change.

Layout preferences are defined using margins to parent groups and windows, gaps between widget, and/or by overlaying a grid over a group or window. A layout comes as a suite of three presets, one for the main application window, one for dialog boxes, and one for toolboxes.

FLUID comes with two included layout suites. `FLTK` was used to design FLUID and other included apps, and `Grid` is a more rigid grid layout. Users can add more suites, import and export them, and include them into their `.fl` project files.

Shell > **Customize... (Alt+x)**: Displays the shell command settings panel. Shell commands are commonly used to run a 'make' script to compile the FLUID output. See [Shell Commands](#) .

Help > **About FLUID**: Pops up a panel showing the version of FLUID.

Chapter 5

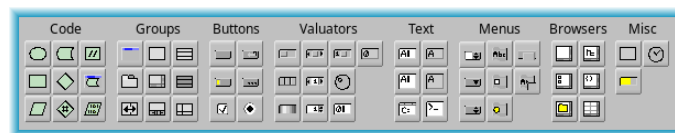# Widget Bin Panel

The Widget Bin Panel

Figure 5.1 Widget Bin

The Widget Bin can be activated via the main menu: *Edit > Show Widget Bin* . FLUID will remember its state and dimensions.

The Widget Bin is a great way to quickly create a GUI project. Clicking an icon in the bin will create the corresponding code or widget node inside or right after the selected widget. If the parent widget is not supported for this widget type, FLUID tries to find a better position. If that fails, a dialog box will pop up, explaining what type of parent node is required.

The Window and Widget Class icons can be dragged onto the desktop, creating a new window or widget at the drop position.
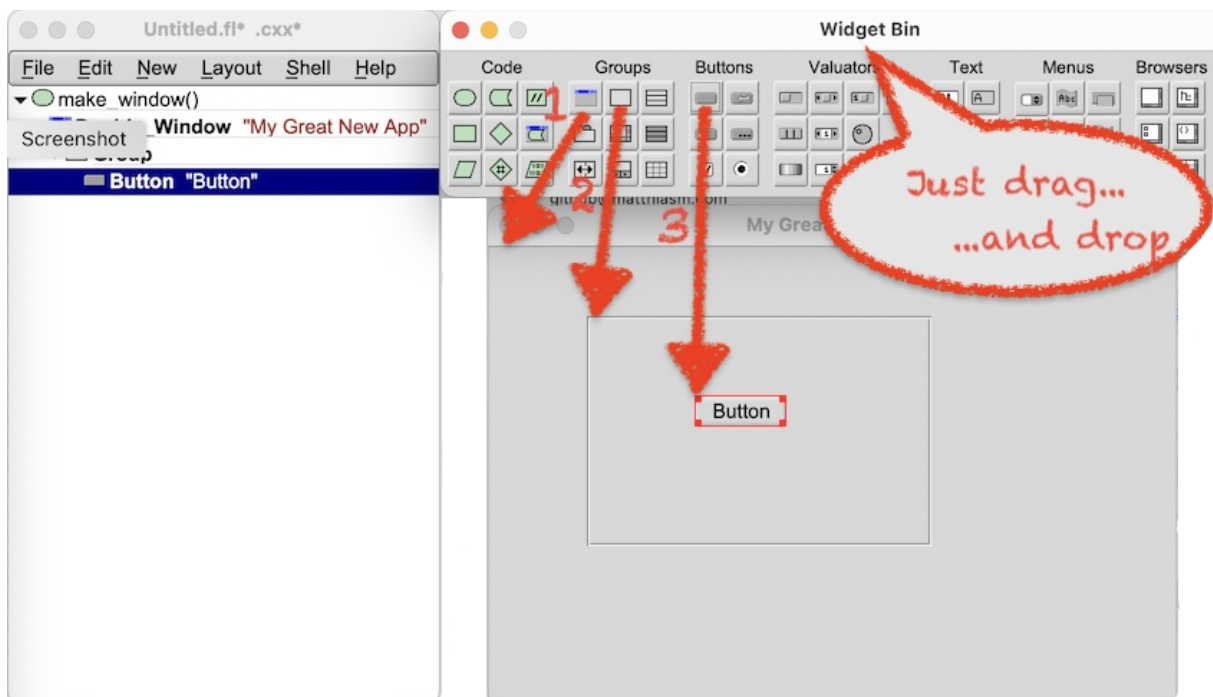
All other widget types can be dragged from the bin into a window, or a group inside a window. When dropped, they will be positioned close to the drop point and inserted into the widget tree as the last child of the chosen group. The order of widgets within their group can be changed with the `F2` and `F3` keys.

Chapter 6

Layout Editor Window

Figure 6.1 Layout Editor Window

The Layout Editor window is used to interactively add groups and widgets, and resize and align them. The editor window already looks very much like the final product that will be built by the FLUID generated C++ source code.

To create a user interface, first add a function to the project tree by either clicking the Function icon in the widget bin, or by selecting **New**∗ > **Code** > **Function**/**Method** from the main menu.

Now just drag the Window icon from the widget bin onto the desktop. FLUID will generate code that instantiates this window when the function is called. The return value of the function is a pointer to that window, unless changed in the Function Panel. Widgets can be added to the window by dragging them from the widget bin. If a widget is dropped on a group, it will automatically become a child of that group.

6.1 Selecting and Moving Widgets

To move or resize a widget, it must be selected first by clicking on it. Multiple widgets can be selected by holding down the Shift key when clicking on them, or by dragging a selection box around widgets. Widgets can also be

selected in the widget browser the main window. Shift-click will select a range of widgets, Ctrl-click will add widgets to the selection.

Figure 6.2 Select Multiple Widgets

Menu items are selected by clicking on the menu button and selecting it from the popup menu. Multiple menu items can only be selected in the widget browser in the main application window.

Once selected, widgets can be moved by clicking and dragging the center of the selection box. The outer edges allow resizing in one direction, and dragging the corners resizes widgets horizontally and vertically.

Widgets can also be repositioned with the arrow keys. Without a shift key, the selection moves by a single pixel. With the Meta key held down, they move by the amount indicated in the *Gap* field in the *Widget* section of the *Layout* setting panel.

Holding down the Shift key resizes a selected widget by moving the bottom right corner of the widget. Holding Shift and Meta while pressing arrow keys resizes by the amount in the *Widget Gap* layout setting.

Children of groups that reposition their contained widgets may behave differently. Pressing the arrow keys on children of `Fl_Grid` will move the widget from grid cell to grid cell instead. Resizing a child of `Fl_Flex` will also mark the child size as `fixed`.

The tab and shift+tab keys "navigate" the selection. Tab or shift+tab move to the next or previous widgets in the hierarchy. If the navigation does not seem to work you probably need to "Sort" the widgets. This is important if you have input fields, as FLTK uses the same rules when using tab keys to move between input fields.
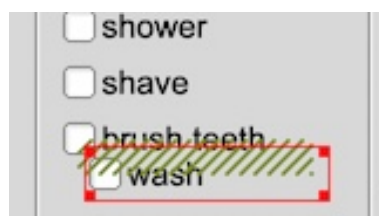
6.2 Layout Helpers

Figure 6.3 Overlapping Widgets

In FLTK, the behavior of overlapping children of a group is undefined. If enabled in the settings, FLUID will show overlapping widgets in a group with a green hash pattern.

Figure 6.4 Out Of Bounds

The behavior of widgets that reach outside the bounds of their parent group is also undefined. They may be visible, but confuse the user when they don't react to mouse clicks or don't redraw as expected. Outside widgets are marked with a red hash pattern.

Note that `Fl_Tile` requires that all children exactly fill the area of the tile group to function properly. The hash patterns are great helpers to align children correctly.

6.3 Layout Alignment

FLUID layouts are a handful of rules that help creating a clean and consistent user interface. When repositioning widgets, the mouse pointer snaps to the closest position based on those rules. A guide line is drawn for the rule that was applied. Guides and snaps can be disabled with `Ctrl-Shift-G` or via the Edit∗ > *Hide Guides* menu.
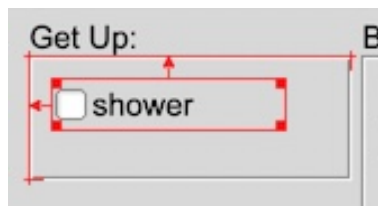

Figure 6.5 Snap To Group

If a horizontal or vertical outline snaps to the group, the border of that group will highlight. If the outline snaps to the margin of the parent window or group, an additional arrow is drawn.

Children of `Fl_Tabs` use the top and bottom margin from the *Tabs* section. If all children use this rule, the margin height will also be the height of all tabs.
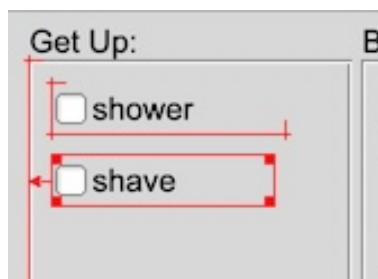

Figure 6.6 Snap To Sibling

The selection can also snap to the outline of other widgets in the same group, or to the outline plus the Widget Gap. The outline that triggers the snap action is highlighted.

Note that only the first snap guide found is drawn for horizontal and vertical movement. Multiple rules may apply, but are not highlighted.
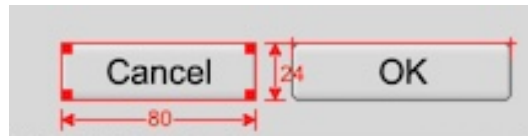

Figure 6.7 Snap To Size

Widget size rules define a minimum size and an increment value that may be applied multiple times to the size. For example, with a minimum width of 25 and an increment of 10, the widget will snap horizontally to 25, 35, 45, 55, etc. .

Figure 6.8 Snap To Grid

The grid rule is the easiest to explain. All corners of a selection snap to a fixed grid. If the selected widgets are children of a window, they will snap to the window grid. If they are in a group, they snap to the group grid.
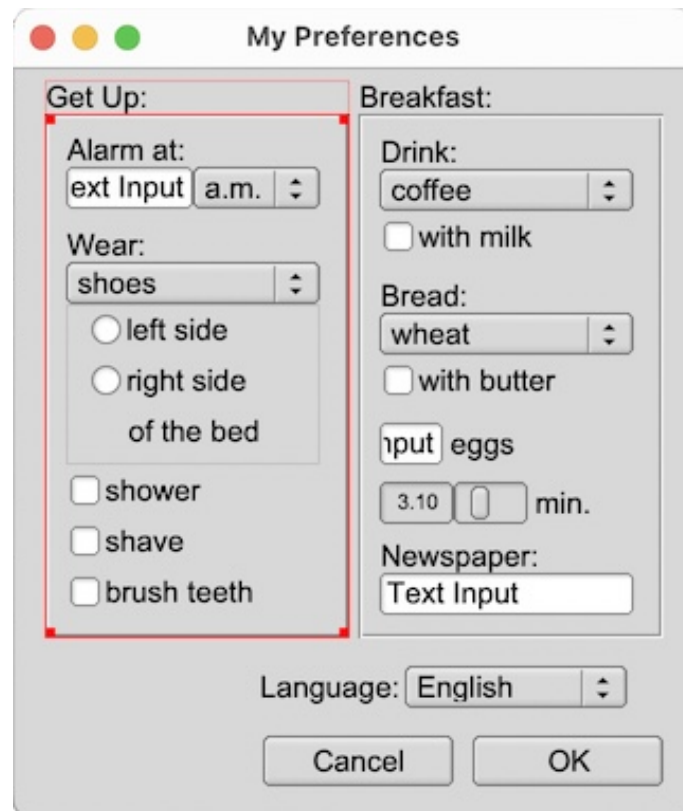
6.4   Live Resize

Figure 6.9 Selected Group

The Resizable system within FLTK is smart, but not always obvious. When constructing a sophisticated GUI, it is helpful to organize widgets into multiple levels of nested groups. Sometimes, incorporating an invisible resizable box can improve the behavior of a group. FLUID offers a Live Resize feature, allowing designers to experiment with resizing at each level within the hierarchy independently.

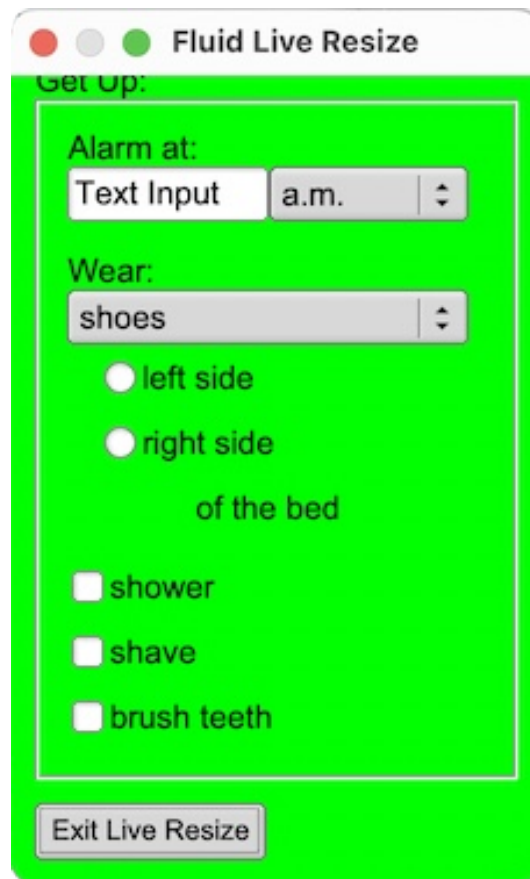To test the resizing behavior of a group, begin by selecting it:

Figure 6.10 Live Resize

Click on *Live Resize* in the widget panel. FLUID will generate a new window with all the resizing attributes inherited from the original design. This enables the designer to thoroughly test the behavior and limitations, making adjustments until they are satisfied. This streamlined process makes it significantly easier to address resizing behavior at a higher level, particularly once the lower levels are behaving as intended.

In the example above, the radio buttons are not fixed to the left side of the group and the text snippet "of the bed" does not stay aligned to "right side". To fix this, a thin hidden box could be added to the right edge of the group that holds the radio button which is then marked `resizable`.

6.5 Limitations

Almost all FLTK widgets can be edited with FLUID. Notable exceptions include

- FLUID does not support an `Fl_Window` inside another `Fl_Window`

- widgets inside `Fl_Scroll` can not be created in the hidden areas of the scrollable rectangle. It is recommended to organize the children in a separate Widget Class that is derived from `Fl_Scroll` and then inserted as a single custom widget.

- children of `Fl_Pack` are not automatically reorganized to fit the packing group. Again, a Widget Class is recommended here.

- if children of `Fl_Grid` are again some kind of group, their internal layout may not follow changes in the grid widgets. It's best to complete the grid first, then add children to the grid cells, size them correctly, and then finally lay out the grid cell children.

Chapter 7

Functional Node Panels

7.1 Function and Method Panel

◯ Functions and Methods

Fluid can generate C functions, C++ functions, and methods in classes. Functions can contain widgets to build windows and dialogs. *Code* nodes can be used to add more source code to a function.

Parents

To generate a function, the function node must be created at the top level or inside a declaration block. If added inside a class node, this node generates a method inside that class.

Children

Function nodes can contain code nodes and windows that in turn contain widgets. If the function node has no children, only a forward declaration will be created in the header, but no source code will be generated.
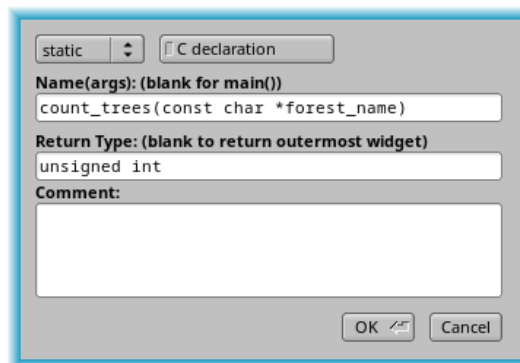
Figure 7.1 Function/Method Properties

Declaring a Function

A function node at the top level or inside a declaration block generates a C or C++ function.

The *Name* field contains the function name and all arguments. If the *Name* field is left empty, Fluid will generate a typical 'main()' function.

```
// .cxx
int main(int argc, char **argv) {
  // code generated by children
  w->show(argc, argv); // <-- code generated if function has a child widget
  Fl::run();
}
```

If a function node has a name but no children, a forward declaration is generated in the header, but the implementation in the source file is omitted. This is used to reference functions in other modules.

```
// .h
void make_window();
```

If the function contains one or more Code nodes, the implementation code will be generated. The default return type is `void`. Text in the *Return Type* field overrides the default type.

```
// .cxx
void make_window() {
  // code generated by children
}
```

If the function contains one or more windows, a pointer to the first window will be returned. The default return type will match the window class.

```
// .h
Fl_Window* make_window();
// .cxx
Fl_Window* make_window() {
  Fl_Window* w;
  // code generated by children:
  // w = new Fl_Window(...)
  return w;
}
```

Options for Functions

Choosing *static* in the pulldown menu will generate the function `static` in the source file. No forward declaration will be generated in the header file.

```
// .cxx
static Fl_Window* make_window() { ... }
```

Choosing *global* will generate a forward declaration of the function in the header file and no `static` attribute in the source file.

```
// .h
void make_window();
// .cxx
Fl_Window* make_window() { ... }
```

Additionally, if the *C* option is checked, the function will be declared as a plain C function in the header file.

```
// .h
extern "C" { void my_plain_c_function(); }
// .cxx
void my_plain_c_function() { ... }
```

The *local* option will generate a function in the source file with no `static` attribute. No forward declaration will be generated in the header file.

```
// .cxx
Fl_Window* make_window() { ... }
```

Declaring a Method

A function node inside a class node generates a C++ method. If a method node has no children, the declaration is generated in the header, but no implementation in the source file.

```
// .h
class UserInterface {
public:
  void make_window();
};
```

If the method contains one or more Code nodes, an implementation will also be generated.

```
// .cxx
void UserInterface::make_window() {
  printf("Hello, World!\n");
}
```

If the method contains at least one widget, a pointer to the topmost widget will be returned and the return type will be generated accordingly.

```
// .h
class UserInterface {
public:
  Fl_Double_Window* make_window();
};
// .cxx
Fl_Double_Window* UserInterface::make_window() {
  Fl_Double_Window* w;
  // code generated by children
  return w;
}
```

Options for Methods

Class access can be defined with the pulldown menu. It provides a choice of `private`, `protected`, and `public`.

Fluid recognizes the keyword `static` or `virtual` at the beginning of the return type* and will generate the declaration including the keyword, but will omit it in the implementation. The return type defaults still apply if there is no text after the keyword.

Further Options

Users can define a comment text in the *comment* field. The first line of the comment will be shown in the widget browser. The comment text will be generated in the source file before the function.

```
// .cxx
//
// My multilen comment will be here... .
// Fluid may actually use C style comment markers.
//
Fl_Window* make_window() {
```

FLUID recognizes default values in the argument list and generates them in the declaration, but omits them in the implementation.

A short function body can be appended in the *Name* field. With no child, this creates an inlined function in the header file.

7.2 C Source Code

 Code

Code nodes hold arbitrary C++ code that is copied verbatim into the source code file. They are commonly used inside Function nodes.

Parents

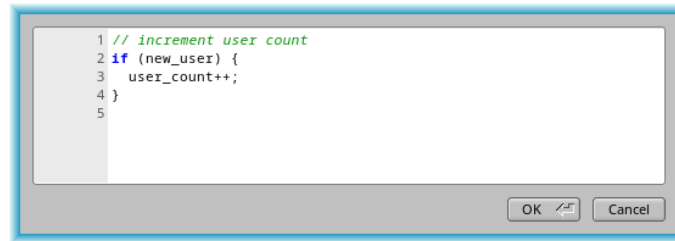Code nodes can be added inside Functions, Code Blocks, and Widget Classes.

Figure 7.2 Code Properties

The Code Properties panel features a syntax-highlighting C++ code editor. Some basic bracket and braces match checking is done when closing the dialog.

When inside a Function or Code Block, the C++ code is inserted directly. Inside a Widget Class, the code will be added to the constructor of the widget class.

7.3   Code Block

◇ Code Block

Code Blocks are used when a single function generates different GUI elements conditionally.

Parents

Code Blocks are used inside functions and methods.

Children

Code Blocks can contain widgets, code, or more code blocks.
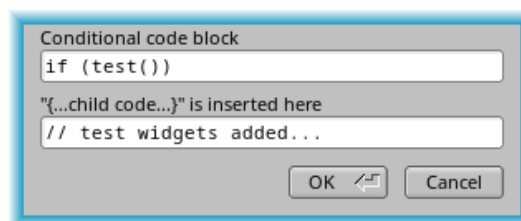

Figure 7.3 Code Block Properties

The two fields expect the code before and after the `{  ...  }` statements. The second field can be empty.

Two consecutive Code Blocks can be used to generate `else/else if` statements by leaving the second field of the first node empty.

7.4 Declaration

 Declaration

Parents

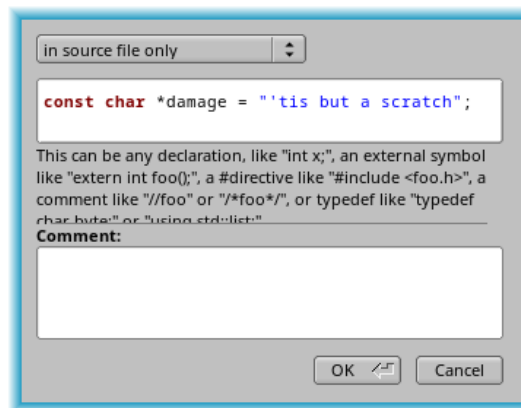Declarations can be added at the top level or inside classes and widget classes.

Figure 7.4 Declaration Properties

Declaration nodes are quite flexible and can be a simple variable declaration such as `int i;`. But include state-ments are also allowed, as are type declarations, and comments. FLUID does its best to understand user intention, but the generated code should be verified by the user.

Declarations nodes at the top level can selectively generate code in the header and /or in the source file. If a declaration is inside a class, the user can select if the class member is *private*, *protected*, or *public* instead.

7.5 Declaration Block

 Declaration Block

Declaration Blocks are a way to selectively compile child nodes via preprocessor commands, typically `#ifdef TEST` and `#endif`.

Parents

Declaration Blocks can be created at the top level or inside classes.

Children

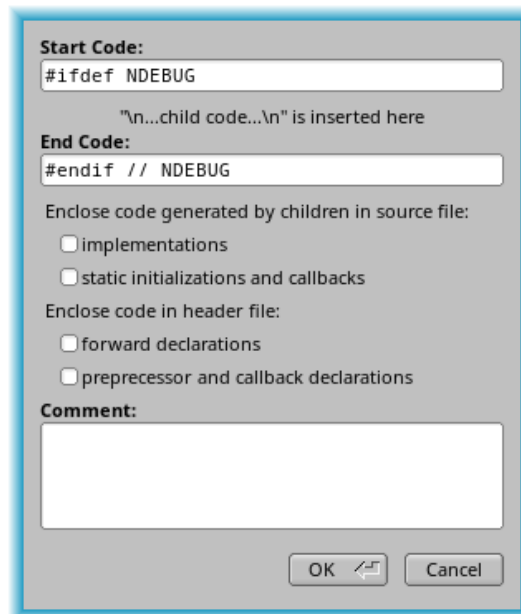Declaration Blocks can contain classes, functions, methods, declarations, and comments.

Figure 7.5 Declaration Block Properties

The C++ code in the "Start" field is output before the code of all children of this node is written to the source file. The text in the "End" field is written after code for all children was generated.

The following check boxes enable code generation for different locations in header and source code files. The first two boxes modify the C++ source code file. If the first check box, "implementations", is ticked, all C++ code that implements the children of this declaration block will be enclosed with the code from the "Start" and "End" fields. This check box is marked by default.

The second check box, "static initializations and callbacks", will enclose callbacks that may be created by child widgets, menu item arrays, and code as well as data for images. This box should be ticked in most cases, but may be harmful if one image is used more than once and outside of this declaration block.

The next two boxes modify the C++ code in the header file. Ticking "forward declarations" will wrap the code that declares functions, methods, and menus. The last box ensure that code declaring widgets is wrapped with yet another copy of from the "Start" and "End" fields. This will also wrap `#include` statements and declarations from widget "Code" fields.

FLUID optimizes header files by removing duplicate include statements and certain declarations. Declaration blocks are commonly used for conditional compilation and may effectively "optimize away" include statements that are still needed elsewhere. This can be litigated by explicitly creating Declaration nodes outside of the declaration block.

The "Start" and "End" code may appear multiple times per file if more than one of the check boxes above is ticked. The code should not have any side effects or cause conflicts when compiled more than once. It's not safe to rely on a specific order of the generated blocks.

7.6 Classes

⬜ Class

FLUID can generate code to implement C++ classes. Classes can be used to keep dialogs and groups of UI elements organized. See Widget Class nodes as an alternative to implement compound widgets.

Parents

Class nodes can be created at top level or inside a Class or Widget Class node.

Children

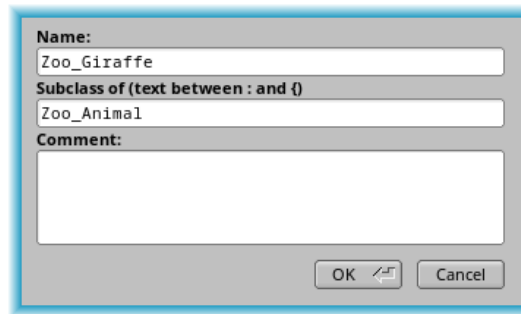Class nodes can contain Functions, Declarations, Widgets, Data, and other classes.

Figure 7.6 Class Properties

The *Name:* and *Subclass of:* fields should be set to standard C++ class names.

Function nodes inside classes are implemented as methods. Constructors and destructors are recognized and implemented as such. Inlined data is declared as a static class member.

Note that methods without a code or widget node are only declared in the header file, but no code is generated for them in the source file.

7.7 Widget Class

Widget Class

The Widget Class node creates a new widget type by deriving a class from another widget class. These are often compound widgets derived from `Fl_Group`. A less automated but more flexible way to implement compound widgets is the Class node.

Parents

Widget Class nodes can be created at top level or inside a Class or Widget Class node.

Children

Widget Class nodes can contain Functions, Declarations, Widgets, Data, and other classes.

Properties

Widget Class nodes use the Widget panel to edit their properties. The super class can be set in the *C++* tab in the *Class* field. If that field is empty, FLUID derives from `Fl_Group`.

The Widget Class always creates a constructor with the common widget parameters:
```
MyWidget::MyWidget(int X, int Y, int W, int H, const char *L)
: Fl_Group(X, Y, W, H, L) { ... }
```

If the super class name contains the text `Window`, two more constructors and a common initializer method are created:
```
MyWidget::MyWidget(int W, int H, const char *L) :
  Fl_Window(0, 0, W, H, L) { ... }

MyWidget::MyWidget() :
  Fl_Window(0, 0, 480, 320, 0) { ... }

void MyWidget::_MyWidget() { ... }
```

Code and Widget nodes are then added to the constructor. Function nodes are added as methods to the class. Declarations are added as class members. Data nodes generate static class members.

It may be useful to design compound widgets with a variable size. The Widget Panel provides a choice menu in the *GUI* tab's *Position* row under Children∗. The options *resize* and *reposition* generate code to fix up the coordinates of the widget after instantiation.

Note that methods without a code or widget node are only declared in the header file, but no code is generated for them in the source file.

7.8 Comments

▨ Comment

This node adds a comment block to the generated source code.

Parents

Comment nodes can be added inside Functions, Code Blocks, and Widget Classes. If a Comment node is the top node in a tree, it will appear in the source files even before the `// generated by FLUID ...` line.
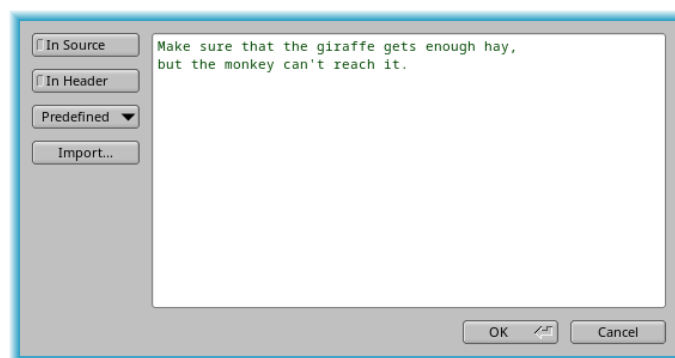

Figure 7.7 Comment Properties

Comment blocks are generated by adding `//` to the start of each line unless the first line of a comment starts with `/*`. In that case, FLUID assumes a correct block comment and will copy the text verbatim.

Comments can be generated in the header file, the source file, or both.

FLUID keeps a small database of predefined comments. Users can add reoccurring comment blocks, license information for example, to this database via the pulldown menu.

Comments can also be imported from an external file.

7.9 Inlined Data

Inlined Data

The Data node makes it easy to inline data from an external file into the source code.

Parents

Data nodes can be added at the top level or inside Widget Classes, Classes, and Declaration Blocks.
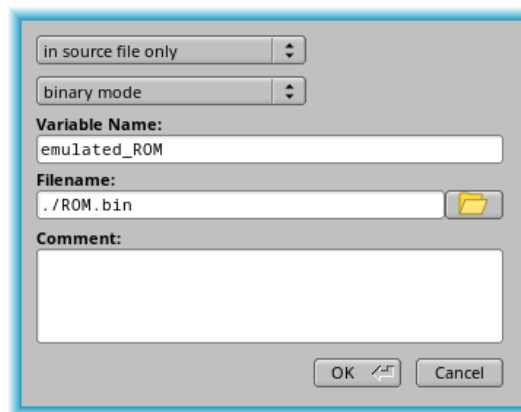

Figure 7.8 Data Properties

At top level, or inside a Declaration Block, Data can be declared *in source file only*, *static in source file*, or *in source and extern in header*.

If Data is inside a Class node, it is always declared `static`. The user can select *private*, *protected*, or *public*.

Data in binary mode will be stored in an `unsigned char` array. The data size can be queried with `sizeof()`. In Text mode, it will be stored as `const char*` and terminated with a `NUL` character.

In compressed mode, data will be compressed with zlib `compress()` and stored in an `unsigned char` array. A second variable, holding the original data size, is declared `int` by appending `_size` to the variable name.
```
// .cxx
int myInlineData_size = 12034;
unsigned char myInlineData[380] = { 65, 128, ... };
```

The Variable Name should be a regular C++ name. The Filename field expects the path and name of a file, relative to the location of the .fl file.

Chapter 8

Widget Properties Panel

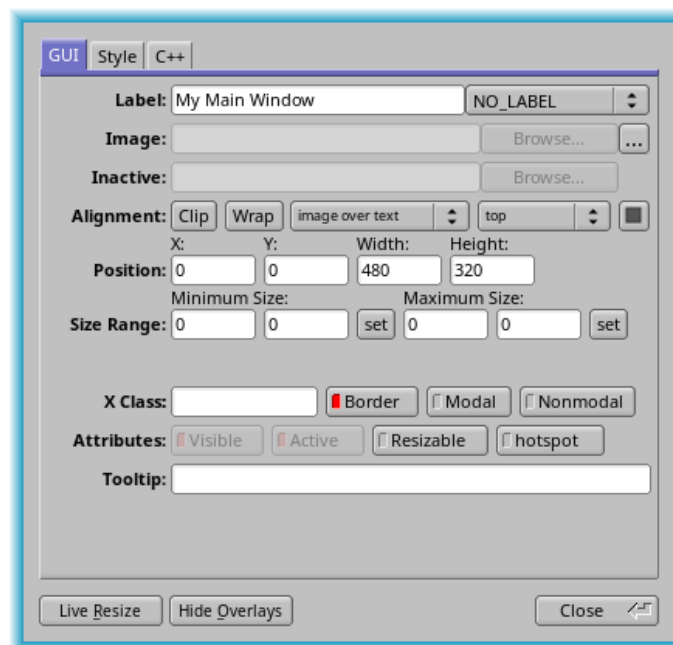## The Widget Properties Panel

Figure 8.1 Widget Properties

This panel is used to edit the properties of the currently selected widgets. It can be opened by double-clicking on a widget or by pressing `F1`.

When you change attributes using this panel, the changes are reflected immediately in the window. It is useful to hit the "Hide Overlays" button (or type Ctrl+Shift+O) to hide the red overlay so you can see the widgets more accurately, especially when setting the box type.

One or more widgets can be selected at the same time, and most attribute changes will be applied to the entire selection. Depending on the selected widget types, some properties may be grayed out or may not be visible.

All changes in the widget panel are immediately applied to all selected widgets and their effect can be seen in the project window. It can be very useful to keep the *Code View* window open at all times. All code changes appear instantly for all generated files if *Auto-Refresh* is active.

FLUID generates only code for properties that differ from their default setting. If a widget class is derived from another class, FLUID can't know the defaults and will generate code for all attributes instead.

Correctly resizing a window can be a complex task. It is easier to check resizing behavior on a more local level first. To use *Live Resize*, select any group or window in your project. FLUID creates a resizable clone of that part of your design to try out resizing behavior of that group only.
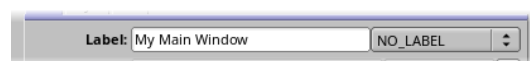
The widget panel itself is resizable to make more room for entering code and long texts.

8.1 The GUI Tab
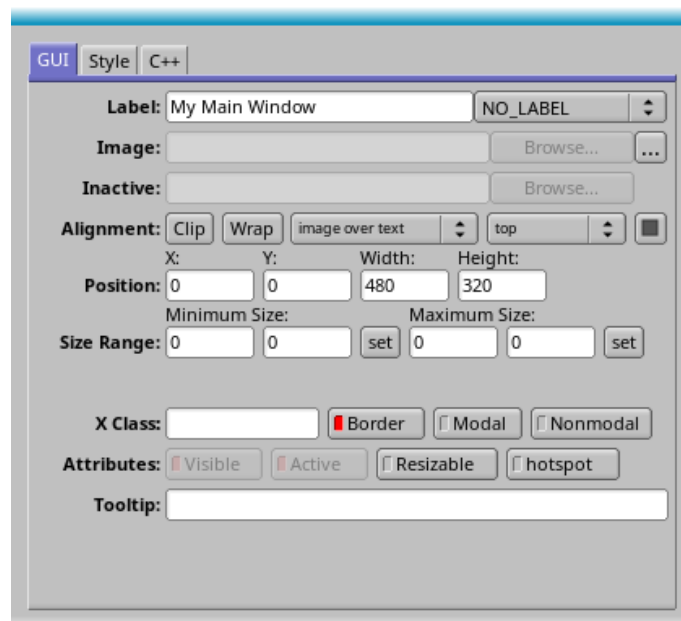
The widget panel has three standard tabs that apply to all widgets. Some widget types, `Fl_Grid` and children of `Fl_Grid`, will create additional tabs with more options.

The GUI tab controls basic GUI settings, including label text and widget size.

The *Label* field can be any Unicode string and is stored as a static text. If internationalization is enabled, the corresponding modifiers are added. Labels can span multiple lines by pressing `Ctrl-J` to insert a newline (NL) character.

The `@` character adds a symbol to the label. See *Labels and Label Types* in the FLTK documentation.

The pulldown menu offers some additional rendering styles for the label.

Add an image to the widget label here. The second row takes an optional image for rendering a deactivated widget.

The image path must be relative to the location of the `.fl` file and not necessarily the current directory. It is helpful to keep images in the same directory as the `.fl` file.

The image data is inlined into the source code. If many widgets share the same image then only one copy is written. Since the image data is embedded in the generated source code, you need only distribute the C++ code and not the image files themselves. The `.fl` project files however store only the image file name, so you will need the image files as well to read a project file.

FLUID can read XBM bitmap files, XPM pixmaps with a transparency channel, and all other types supported by the FLTK image extension. Images can be stored in their original file format, or converted into their uncompressed rgb or grayscale pixel data, with or without alpha channel. By default, image files ending in .jpg, .png, .svg, and .svgz are stored as they are. All other formats are converted to pixel data. The storage format can be manually selected in the "convert to raw pixel data" checkbox in the image properties dialog.

Images stored in their original format are usually compressed well and take a lot less space, but they also require that the fltk-image library and all its dependencies are linked to the application. Storing uncompressed pixel data increases the size of the application, but has less dependencies and saves time when launching because images don;t need to be decompressed. As a good rule of thumb, keeping the original format is good for images larger than 24x24 pixels and when the application links to the fltk-image anyway. An app that has only a hand full of small icons may be better off storing raw pixel data and not link with fltk-image.

The image properties dialog provides *Scale* settings to scale the image before rendering to screen. To make full use of high-dpi screen support, images should be stored at double resolution and then scaled to FLTK coordinates. This gives FLTK the chance to fall back to the full size image for high-dpi screens.

Control alignment of the label in relation to the widget position and size as well as the relation between the image and the label. The box on the right toggles between inside and outside label alignment.
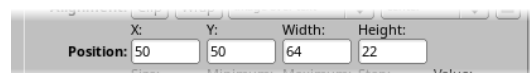
Control the size and position of a widget here. The input fields react to vertical scroll wheel input for interactive positioning.

All fields understand basic math. They are evaluated after the formula is entered and the result is stored in the respective properties. Formulas can also contain a number of variables. The *x* input can handle the variables `x` for its own position, `px` for the parent position, `sx` for the previous sibling, `cx` for the leftmost x position of all children, and `i`, which is a counter through all selected widgets.

The formula `x+10` in the *x* field moves all selected widgets 10 pixels to the right. `100+25*i` in the *y* field arranges all widgets vertically starting at 100 with 25 pixels distance.

Name	Value
`i`	zero based counter of selected widgets
`x, y, w, h`	position and size of the current widget
`px, py, pw, ph`	dimensions of the parent widget
`sx, sy, sw, sh`	dimensions of the previous sibling
`cx, cy, cw, ch`	bounding box of all children

Activate for widgets that can take numerical values, these input fields take floating point numbers. They generate lines like `o->minimum(2);` only if the corresponding value differs from the default value for this property.

This row is only visible for children of `Fl_Flex` widgets. It sets the width or height of a widget in a horizontal or vertical Flex widget. If *fixed* is unchecked, this value is instead calculated by the Flex.

This row is only visible for `Fl_Flex` widgets. It sets the various margins and the gap value for this widget.

This row is only visible for top level windows. The fields set the minimum and maximum size range for windows. Use the *set* button to copy the current size. Set width and height to `0` to disable that aspect of the size range.

This option is only visible for buttons and other widgets that can react to a shortcut key combination. FLUID does not check if a shortcut was already used elsewhere.

If *shortcut use FL_COMMAND* is set in the project settings, modifiers are created in a more compatible way across platforms.

This row is only visible for top level windows. Note that selecting *modal* and *non modal* together is undefined.

The string typed into the *X Class* field is passed to the X window manager as the class. This can change the icon or window decorations. On most window managers you will have to close the window and reopen it at runtime to see the effect.

Some additional attributes for all widget types.

The *Visible* button controls whether the widget is visible (on) or hidden (off) initially. Don't change this for windows or for the immediate children of a Tabs group.

The *Active* button controls whether the widget is activated (on) or deactivated (off) initially. Most widgets appear grayed out when deactivated.

The *Resizable* button controls whether the window or widget is resizable. In addition all the size changes of a window or group will go "into" the resizable child. If you have a large data display surrounded by buttons, you probably want that data area to be resizable. You can get more complex behavior by making invisible boxes the resizable widget, or by using hierarchies of groups. Resizing of a window or group can be tested using the *live resize* button.

Note that the *Resizable* indicator is ambiguous when a window is selected. Making a window resizable will resize all children proportionally. Setting a child of a window will make that child the center of the resize operation. In both cases, the *Resizable* indicator of the window will be set.

The *Hotspot* button causes the parent window to be positioned with that widget centered on the mouse. This position is determined when the FLUID function is called, so you should call it immediately before showing the window. If you want the window to hide and then reappear at a new position, you should have your program set the hotspot itself just before `show()`.

The *Tooltip* field can be any Unicode string and is stored as a static text. If internationalization is enabled, the corresponding modifiers are added.
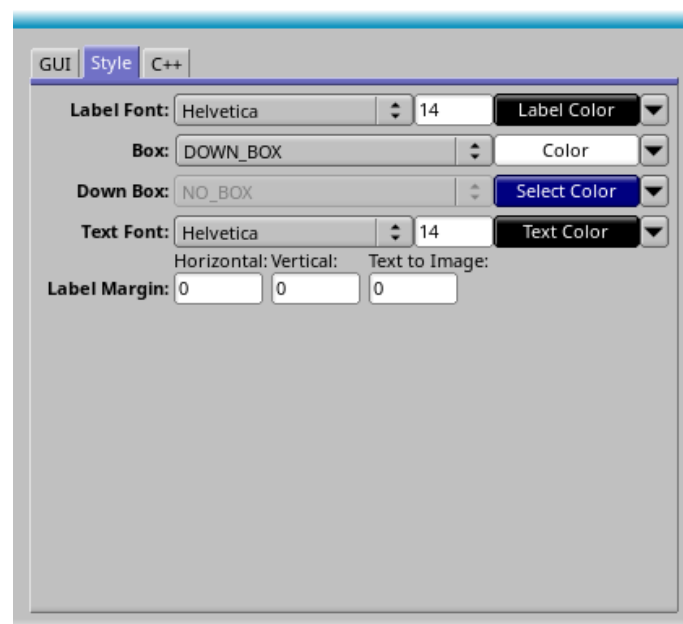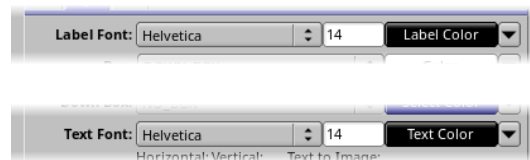
8.2 The Style Tab

Figure 8.2 Style Tab

The Style tab is used to edit font styles and sizes, and the color of elements of the widget.

The font pulldown menu provides a list of standard fonts. To enter the index of a user loaded font, an *extra code* field must be used. The *label color* and *text color* fields opens a color palette selector. The arrow pulldown contains a list of the most commonly used colors. Again, user specific colors can be defined using the *extra code* field.

The *Text Font* row is only available for widgets that contain an additional text area.
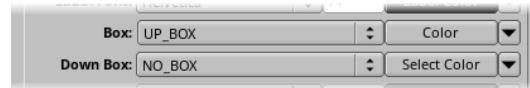
Select the up and down box for the given widget. The first six entries in the box and frame style list are influenced by the FLTK Scheme setting. Other box styles will always look the same, independently of the selected scheme.

Many widgets will work, and draw faster, with a "frame" instead of a "box". A frame does not draw the colored interior, leaving whatever was already there visible. Be careful, as FLUID may draw this ok but the real program may leave unwanted stuff inside the widget.

If a window is filled with child widgets, you can speed up redrawing by changing the window's box type to "NO_BOX". FLUID will display a checkerboard for any areas that are not colored in by boxes. Note that this checkerboard is not drawn by the resulting program. Instead random garbage will be displayed.

The *Down Box* row is only available for widgets that can be pressed down by the user.

Some widgets will use the *Select Color* for certain parts. FLUID does not always show the result of this: this is the color buttons draw in when pushed down, and the color of input fields when they have the focus.
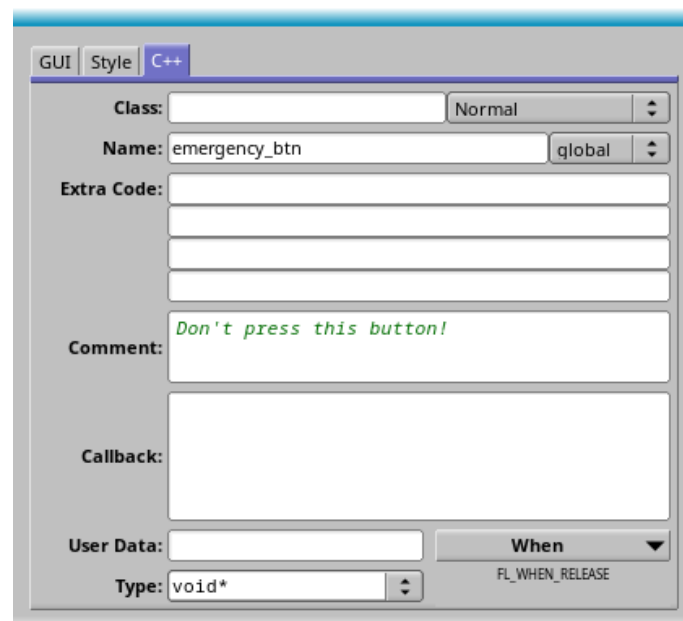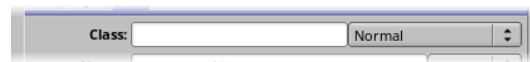
8.3 The C++ Tab

Figure 8.3 C++ Tab

The C++ tab has various input fields for adding C++ code at various places in the source and header file.

If the class property is set, FLUID assumes that the user wants to instantiate a widget that is derived from the selected widget. For a derived widget, the default values of attribute can not be known. FLUID will generate code to explicitly set every known attribute of the super class.

FLUID generates "include" statements for known classes. Custom classes should provide a `#include` line as one of the "Extra Code" lines of the widget.

If the selected widget is a Widget Class node, the Class property will instead set the super class of the widget.

The pulldown menu on the right side contains additional subtypes for some widget types. `Fl_Button` widgets, for instance, can be further refined to be an `Fl_Toggle_Button` or an `Fl_Radio_Button`.

The name field can be any valid C++ variable name. If the widget is inside a class, the pulldown menu lets the user select between *private*, *protected*, and *public*. If not in the group, the variable can be *global* or *static* within the source file.

Widgets created by FLUID are either "named", "complex named" or "unnamed". A named widget has a legal C++ variable identifier as its name (i.e. only alphanumeric and underscore). In this case FLUID defines a global variable or class member that will point at the widget after the function defining it is called. A complex named object has punctuation such as `'.'` or `'->'` or any other symbols in its name. In this case FLUID does not attempt to declare it. This can be used to get the widgets into structures. An unnamed widget has a blank name and no pointer is stored.

You can name several widgets with "name[0]", "name[1]", "name[2]", etc.. This will cause FLUID to declare an array of pointers. The array is big enough that the highest number found can be stored. All widgets in the array must be the same type.
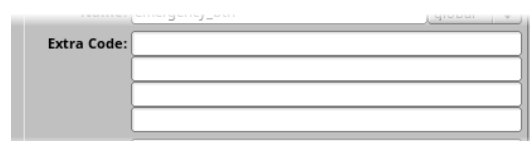

These four input fields can be used to add arbitrary code to different parts of the header and source file. A line can be divided into multiple lines of code by inserting a `Ctrl-J`.

All Extra Code fields are interpreted individually. If a field contains a # character, or the words `extern`, `typedef`, or `using`, FLUID assumes that the code is a declaration and writes it to the header file, and only if it does not duplicate previously written code. This is great for creating a `#include "MyWidgetType.H"` include statement in the header.

If the code is not recognized as a declaration, it will instead be put after the code that instantiates the widget and all its children. For menu items, the code is added after the container `Fl_Menu_` is created, but before the menu array is added to the container.

FLUID will check for matching parentheses, braces, and quotes, but does not do much other error checking. Be careful here, as it may be hard to figure out what widget is producing an error in the compiler. If you need more than four lines you probably should call a function in your own `.cxx` code.
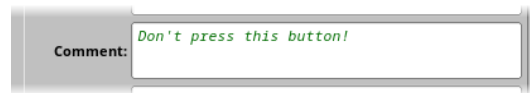
Comments are added to the source code before the widget constructor by adding `//` in front of every line of the comment. The first few characters of a comment are also visible in the widget browser in the main window.

The callback field can be interpreted in two ways. If the callback text is only a single word, FLUID assumes that this is the name of an external callback function and declares it in the header as `extern void my_button↩_action(Fl_Button*, void*);`.

Otherwise, FLUID assumes that the text is the body of a C++ callback function and instead creates a local static callback function. The name of the callback function is generated by FLUID and guaranteed to be unique within the file.

```
static void cb_input(Fl_Input *o, void *v) {
  ... // my text from the Callback field here
}
```

You can refer to the widget as `o` and the `user_data()` as `v`. FLUID will check for matching parentheses, braces, and quotes, but does not do much other error checking.

If the callback is blank then no callback is set.

The *User Data* field can contain any valid C++ code and is copied as the callback argument. If blank the default value of zero is used. Type∗ is currently limited to a pointer (a type name ending in ∗) or `long`.

The *When* pulldown gives access to the `Fl_When` flags, including some commonly used combinations.
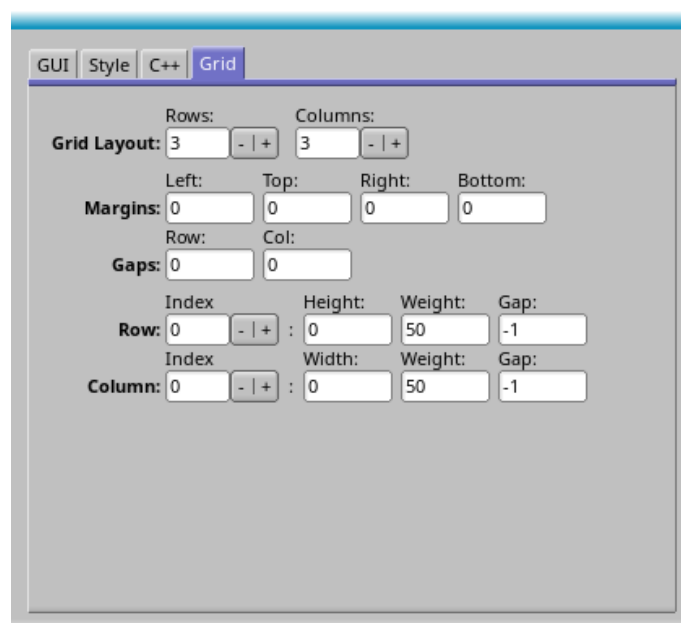
8.4 The Grid Tab

Figure 8.4 Grid Tab

This tab is only available if the selected widget is an `Fl_Grid`. When editing a Grid widget, no other widgets should be selected.

The *Grid Layout* fields adjust the number of rows and columns in the grid.

The *Margins* fields describe the size of the margins around all children of the grid.

The *Gaps* fields set the gaps between individual children in the grid.

The *Row* and *Column* groups can be used to set the size of individual rows and columns within the grid.
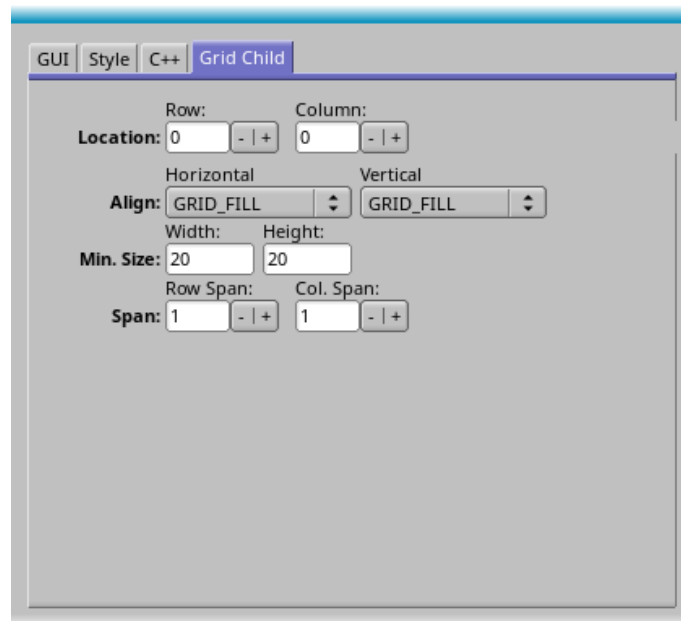
8.5 The Grid Child Tab

Figure 8.5 Grid Child Tab

This tab is only available if the selected widget is a child of an `Fl_Grid`. When editing a child of a Grid widget, no other widgets should be selected.

Use the *Location* group to move a child around within the grid. Note that every cell in a grid can only manage one single widget. When moving widgets over occupied cells, they become "transient". Just continue and move them into an available cell. If a layout is saved with a transient widget, all grid attributes for that widget are lost, and it will remain unassigned in the project file and in the source code.

The *Align* fields provide a way to align a widget within its cell.

The *Min. Size* fields define a minimum width and height for the widget in the cell.

The *Span* fields change the number of cells that a widget can span in x and y.

Note

> Most attributes in this tab will also change the size of the widget. If the child of the Grid is itself a group, the children of that group do not follow changes in position or size. It is recommended to either lay out the grid contents first and leave it unchanged, or to use widgets generated with Widget Class that automatically adjust themselves to the size constraints of the grid.

Chapter 9

Settings Dialog
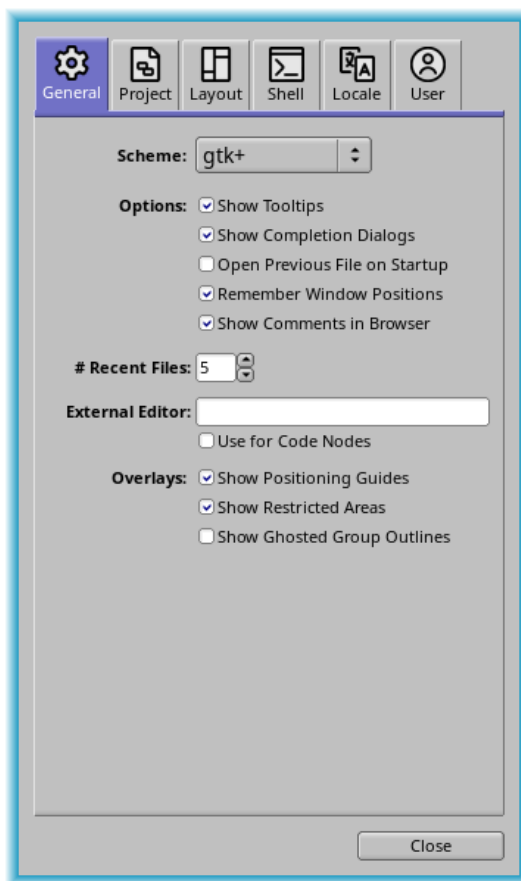
Figure 9.1 Settings Dialog

The *Settings* dialog combines application preferences and project settings in a compact set of six tabs.

The *General* tab contains a collection of application wide settings. They are stored as user preferences.

The *Project* tab holds settings for the current project. They are saved with the `.fl` file.

The *Layout* tab manages databases of preferred widget alignment. These preferences can be saved per user, or as part of the project, or exported for use in other projects.

The *Shell* tab manages a database of quick access shell commands and scripts. Shell commands can be saved as a user preference and also as part of the `.fl` project file.

The *Locale* tab sets the method of internationalizing texts in the project, commonly used for labels and tooltips.

The *User* tab manages customization of fonts and colors in the widget browser. These settings are stored as user preferences.

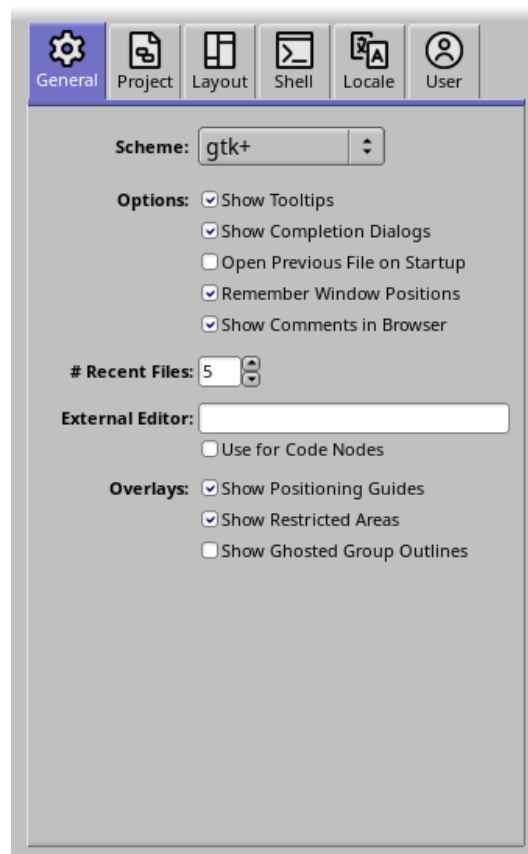## 9.1 Application Settings

Figure 9.2 General Settings Tab

Scheme:

Select one of the graphics schemes built into FLTK. It's helpful to verify the look of various schemes for an application design.

Options:

Various options to make life as a developer more convenient.

Recent Files:

FLUID keeps track of recently opened files.

External Editor:

Users that don't like the built-in FLUID code editor can enter a shell command here that opens the content of Code nodes in an external editor. FLUID does its best to pick up on changed content or when the editor is closed.

Overlays:

The *Position Guides* are little red arrows that indicate if snap points are found. See the *Layout* tab for details. *Restricted Areas* are areas where widgets from within the same group overlap. They are visible in the project window as a diagonally hashed pattern. *Ghosted Group Outlines* show faint frames around groups that would otherwise be invisible in the project window.

9.2 Project Settings
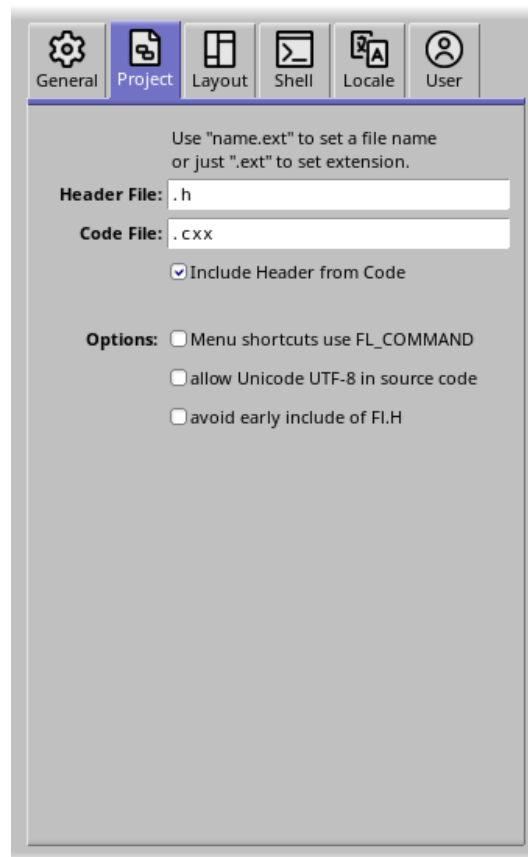
Figure 9.3 Project Settings Tab

Header File, **Code File**:

These fields are used to build the file path and name of the generated header and source file. If one field is empty the value defaults to `.h` and `.cxx` respectively. If a name starts with a `.`, FLUID assumes that the rest of the text is a file extension. The code file name is then generated by replacing the extension of the `.fl` project file name.

Todo Document the exact way the source and header file paths are calculated for interactive FLUID, and for FLUID launched from the command line.

Include Header from Code:

If checked, the statement to include the header file is automatically generated in one of the first lines of the source file.

Menu shortcuts use FL_COMMAND:

Setting this option will replace FL_CTRL and FL_META as a modifier for shortcuts with the platform aware modifiers FL_COMMAND and FL_CONTROL, making shortcuts more portable between macOS and Windows/Linux.

allow Unicode:

If unchecked, Unicode characters in strings are escaped. If checked, the Unicode character is stored in the source code in UTF-8 encoding.

avoid early include:

FLUID by default includes $<FL/Fl.H>$ early in the header file. If this option is checked, users can include other files before including the FL header. The user must then include $<FL/Fl.H>$ later using a Declaration node.
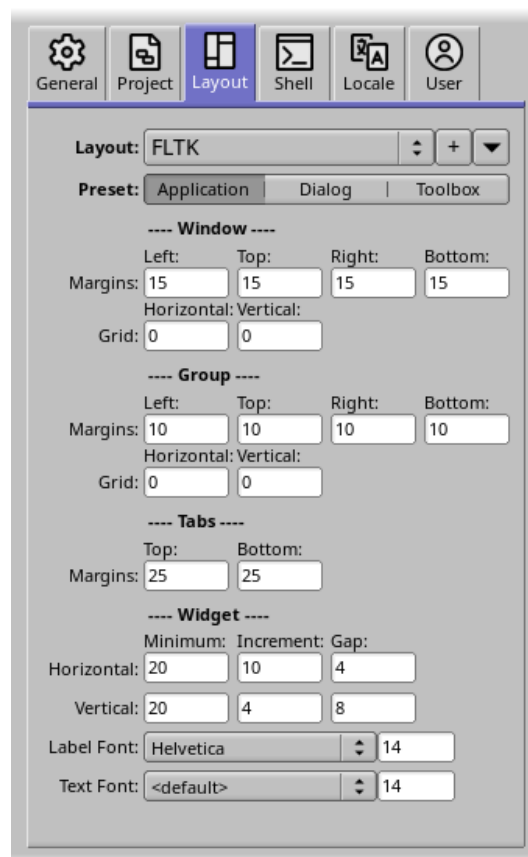
9.3 Layout Preferences

Figure 9.4 Layout Settings Tab

Layouts are a collection of hints that help when interactively positioning and resizing widgets in the project window. Layouts come in a set of three for the application window, for dialog boxes, and for toolboxes.

Layout:

The layout pulldown menu lets users choose from a list of existing layouts. The plus button creates a new set of layouts based on the currently selected layout. The pulldown menu has items to rename, load, and save layouts. It can also change the location where the layout is stored. The FLUID beaker is for layouts that are predefined in FLUID, the portrait icon stores as user preference, the document icon stores the layout in the `.fl` file, and the disk icon lets users store layout in external files.

Window Margin and Grid:

Snap widget position to that margin in relation to the window. The grid snaps widgets to fixed intervals.

Group Margin and Grid:

Snap widget position to that margin in relation to the group. The grid snaps widgets to fixed intervals relative to the top left of the group.

Tabs Margin:

Snap the tab inside `Fl_Tabs` to the tab border and the offset given in Margins.

Widget Minimum, Increment, and Gap:

Minimum sets the minimal width of a widget. *Increment* is the size multiplier added to the *Minimum* value. *Gap* is the preferred distance to other widgets in the same group.

Label Font, **Text Font**:

The preferred label and text font and size for new widgets.
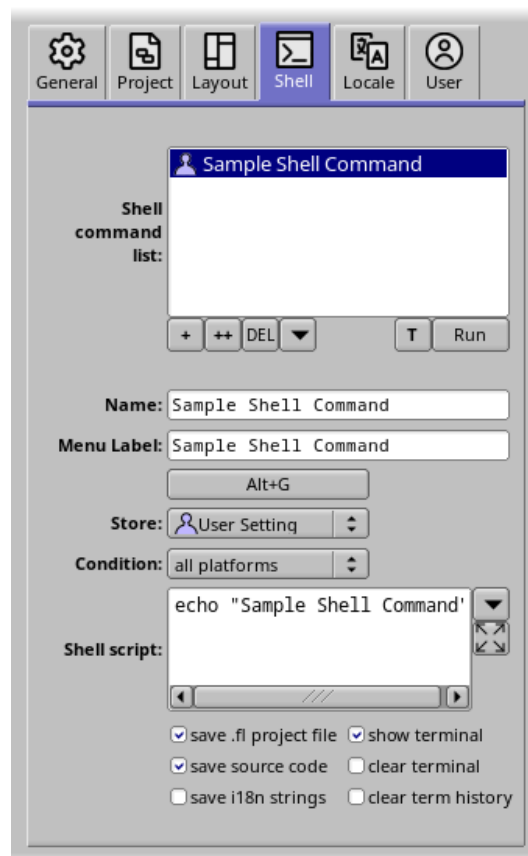
9.4   Shell Commands

Figure 9.5 Shell Settings Tab

Shell Command List:

A list of all currently available shell commands. The portrait symbol in front of the name indicates that the script is stored in the user preferences. The document symbol saves them within the `.fl` project file.

`[+]` adds a fresh new script to the list, `[++]` duplicates the currently selected script. `[DEL]` deletes it, and `[v]` offers import and export functionality. The `[T]` button shows the terminal window, and finally the `[Run]` button runs the selected shell script.

Selecting a shell script will fill in the bottom half of the dialog.

Name:

This is the name of the script as it appears in the Shell Command List.

Menu Label:

Shell scripts that match the *Condition* flag are also available for quick access in the *Shell* menu in the main window and via shortcut key combinations. This is the text that is used for the menu entry.

Shortcut:

Assign a keyboard shortcut to this shell script for even faster access. FLUID does not check if a shortcut is already used elsewhere. Try to avoid collisions, especially when the script is part of a project file.

Store:

Choose where to store the settings of this shell script, either in the user preferences or as part of the `.fl` project file.

Condition:

Shell scripts can be quite different for different platforms hosting FLUID. This choice limits scripts to specific platforms. Multiple scripts can have the same shortcut if they have different conditions.

Shell Script:

This is a text field for the shell script. The `[v]` pulldown menu has a list of variables that are replaced with the corresponding value before running the script. The zoom button gives access to a much larger shell script editor.

The options below are a list of actions that can be executed before running the script.

9.5 Internationalization

The *Locale* tab can be used to configure optional internationalization. FLUID supports GNU `gettext` and POSIX `catgets`.

FLUID supports internationalization (I18N for short) of label strings and tooltips used by widgets. The GNU gettext option also supports deferred translation of statically initialized menu item labels. The setting panel (`Alt+p`) provides access to the I18N options.
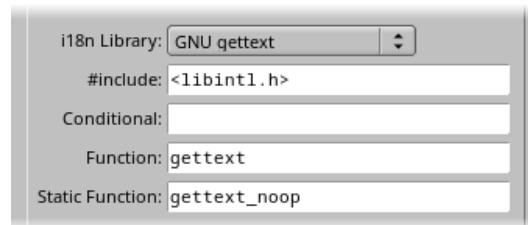

Figure 9.6 I18N With GNU gettext

FLUID supports three methods of I18N: none, GNU gettext, and POSIX catgets. The "none" method is the default and just passes the label strings as-is to the widget constructors.

The "GNU gettext" method uses GNU gettext (or a similar text-based I18N library) to retrieve a localized string before calling the widget constructor.

The GNU gettext option adds some preprocessor code to the source file:
```
#include <libintl.h>
#ifndef gettext_noop
#  define gettext_noop(text) text
#endif
```

and the gettext call around strings in the source code:
```
new Fl_Button(50, 50, 54, 40, "Button");
// ->
new Fl_Button(50, 50, 54, 40, gettext("Button"));
```

FLUID's code support for GNU gettext is limited to calling a function or macro to retrieve the localized label; you still need to call `setlocale()` and `textdomain()` or `bindtextdomain()` to select the appropriate language and message file.

Include: controls the header file to include for I18N; by default this is <**libintl.h**>, the standard I18N file for GNU gettext.

Conditional: If this field contains a macro name, i18n will only be compiled into the product if this macro is defined. The build system should define the macro only if all required headers and libraries are available. If the macro is not defined, no headers are included and `gettext` passes text through untranslated.

Function: controls the function (or macro) that will retrieve the localized message; by default the `gettext` function will be called.

Static Function: names a macro that will mark static text fields for extraction with the `xgettext` tool. The default macro name is `gettext_noop` and will be defined as `#define gettext_noop(text) text` right after the `#include` statement. FLUID will call `gettext` on static texts later, after the textdomain was set by the user.

See also

GNU gettext special cases
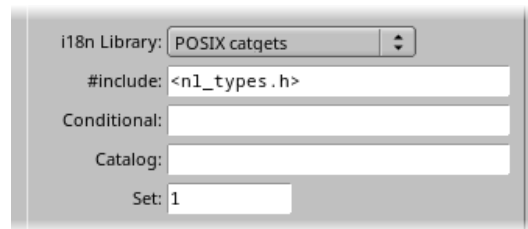

Figure 9.7 I18N With POSIX catgets

The "POSIX catgets" method uses the POSIX catgets function to retrieve a numbered message from a message catalog before calling the widget constructor.

FLUID's code support for POSIX catgets allows you to use a global message file for all interfaces or a file specific to each `.fl` file; you still need to call `setlocale()` to select the appropriate language.

This option adds some preprocessor code to the source file:
```
#include <nl_types.h>
// Initialize I18N stuff now for menus...
#include <locale.h>
static char *_locale = setlocale(LC_MESSAGES, "");
static nl_catd _catalog = catopen("", 0);
```

and the catgets call around strings in the source code:
```
new Fl_Button(50, 50, 54, 40, "Button");
// ->
new Fl_Button(50, 50, 54, 40, catgets(_catalog,1,6,"Button"));
```

Include: controls the header file to include for I18N; by default this is <**nl_types.h**>, the standard I18N file for POSIX catgets.

Conditional: include the header file only if this preprocessor macro is defined.

Catalog: controls the name of the catalog file variable to use when retrieving localized messages; by default the file field is empty which forces a local (static) catalog file to be used for all of the windows defined in your `.fl` file.

Set: controls the set number in the catalog file. The default set is 1 and rarely needs to be changed.

9.6   User Interface Preferences

Figure 9.8 User Settings Tab

This tab lets users change the font and color of text in the widget browser. The settings are stored in the user preferences.

All changes are directly visible in the widget browser.

Chapter 10

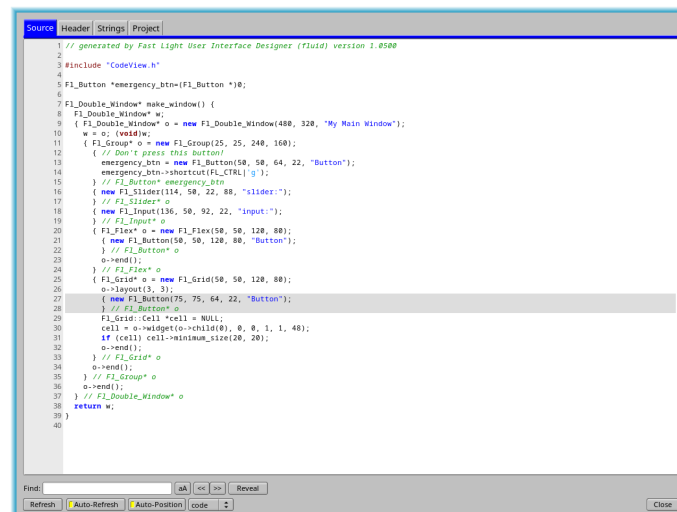# Code View Panel

The Code View Panel

Figure 10.1 Code View Panel

The Code View panel shows all code files that can be generated by FLUID.

The Code View window can be activated via the main menu: *Edit > Show Source Code* . FLUID will remember the state and dimensions of the Code View panel.

If the Auto-Refresh option is selected, the code views will be updated automatically while editing the project.

Code View has four tabs. The first tab shows the source code. Inlined data is omitted in the code view for brevity.

The second tab shows the content of the header file. The size of inline data is not calculated and shown as -1.

The third tab shows the list of labels and tooltips as they would be written to a translation file, using the selected project internationalization method.

The fourth tab previews the contents of the `.fl` project file.

10.1 Code View Find

Figure 10.2 Find in Code

This group of buttons makes it easy to find any text in the Source, Header, or Project file. Press *Reveal* to select the widget that generated the indicated line of code.

Find: enter any text you may want to find in the current tab

aA: press this button to activate case sensitive search

$<<$, $>>$: find the previous or next occurrence

Reveal: clicking this button reveals the widget that generated the selected code in the widget browser

10.2 Code View Settings

Figure 10.3 Code View Settings

Refresh: preview the code in the selected tab as it would be generated for the project in its current state

Auto-Refresh: Automatically refresh the code view when the project changes. The Auto Refresh is designed to use relatively little resources, even when continuously updating the selected code tab. The Code View window can usually stay open and auto refresh during the entire design process, even for relatively complex GUIs.

Auto-Position: highlight and reposition to the source code generated by the currently selected widget whenever the selection changes

code...: choose the type of code that is highlighted. In source files, *static* code is generated by Menu Items, *code* refers to widget creation code, *code1* is the part before possible children, *code2* is code generated after children. In header files, *static* highlights `#include` statements generated by a widget, *code* refers to the widget declaration.

Chapter 11

Tutorials

11.1 Hello, World!

The first FLUID tutorial explains the FLUID basics. It creates a single `main()` function that opens an application window with a static text box inside.

After launching FLUID we want to make sure that two very useful tool windows are open. The "Widget Bin" gives quick access to all available widgets and functional types. It can be opened via the main menu: **Edit** > **Show Widget Bin**, or using the shortcut **Alt-B**.

The second very helpful tool box is the "Code View". The Code View gives a preview of the code as it will be generated by FLUID. All changes in the layout or in attributes are reflected immediately in the Code View. Choose **Edit** > **Show Code View** or press **Alt-C** to get this toolbox. Make sure that *Auto-Refresh* and *Auto-Position* are active in the Code View.

Let's start Hello World from scratch. If there is already a previous project loaded, select **File** > **New** or **Ctrl-N**.

Before we can create a window, we need a "C" function that can be called when we run the program. Select **New** > **Code** > **Function/Method...** or click on the ⬭ image in the widget bin.

A function is added as a first line to the widget tree in the main window, and a "Function/Method Properties" dialog box will pop up. For our Hello World program, delete all text in the top "Name(args)" text field. If this field is left empty, FLUID will generate a `main(argc, argv)` function for us.
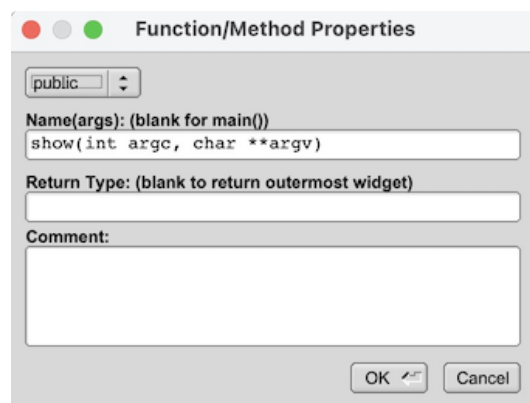

Figure 11.1 Function/Method Properties Dialog

Click OK to apply the changes you made in the Function Properties dialog. You can get this dialog back at any time by selecting the function in the main window and pressing **F1**, or by double-clicking it.

Note that the function will not show up in the Code View yet. FLUID will not generate code for functions that don't have any content, and only create a forward declaration in the header file, assuming that the function will be implemented inside another module.

Keep the `main` function selected and add an `Fl_Window` to the function by selecting **New** > **Group** > **Window...**, by clicking the ▭ image in the Widget Bin, or by dragging the Group/Window image from the Widget Bin onto the desktop.

A new application window will open up on the desktop. The thin red outline around the window indicates that it is selected. Dragging the red line will resize the window. Take a look at the Code View: the main function is now generated, including code to instantiate our `Fl_Window`.

To edit all the attributes of our window, make sure it is selected and press **F1**, or double-click the entry in the main FLUID window, or double-click the window itself. The "Widget Properties" dialog box will pop up. Enter some text in the "Label:" text box and see how the label is updated immediately in the window itself, in the widget list, and in the Code View.

Adding a static text to our window is just as easy. Put an `Fl_Box` into our window by selecting **New** > **Other** > **Box**, or by clicking on the ▭ image in the Widget Bin, or by dragging the Other/Box image from the Widget Bin into our newly created window.

Most importantly, enter the text "Hello, World!" in the "Label:" field of the Box Widget Properties panel to achieve the goal of this tutorial. Now is also a great time to experiment with label styles, label fonts, sizes, colors, etc. .

Finally, we should save our project as an `.fl` project file somewhere. Once the project is saved, select **File** > **Write Code** or press **Shift-Ctrl-C** to write our source code and header file to the same directory.

Compile the program using a Makefile, CMake file, or fltk-config as described in the FLTK manual and the `README.txt` files in the FLTK source tree.

11.2 7GUIs, Task 1

In the first "Hello World" tutorial, we built an entire application in FLUID. It's a boring application though that does nothing except quitting when the close button in the window border is clicked.
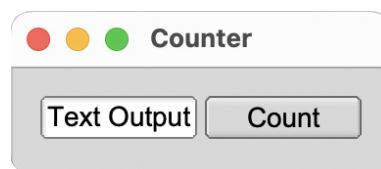

Figure 11.2 Task 1 of 7GUIs

The second tutorial will introduce callbacks by implementing task 1, "Counter", of 7GUIs. 7GUIs has been created as a spin-off of my master's thesis Comparison of Object-Oriented and Functional Programming for GUI Development at the Human-Computer Interaction group of the Leibniz Universität Hannover in 2014. 7GUIs defines seven tasks that represent typical challenges in GUI programming. https://eugenkiss.github.io/7guis/ .

Task 1 requires "Understanding the basic ideas of a language/toolkit. The task is to build a frame containing a label or read-only textfield T and a button B. Initially, the value in T is "0" and each click of B increases the value in T by one."

Our knowledge from tutorial 1 is enough to generate the `main()` function, and add an `Fl_Window`, an `Fl_↩`
`Output`, and an `Fl_Button`. To make life easy, FLUID comes with a built-in template for this tutorial. Just select
File > **New from Template...** and double-click "1of7GUIs" in the Template Panel.

We will need to reference the output widget in our callback, so let's assign a pointer to the widget to a global variable
and give that variable a name. Open the Widget Properties dialog by double-clicking the output widget. Change to
the "C++" tab, and enter "`counter_widget`" in the "Name:" field.

The "Count" button is the active element in our application. To tell the app what to do when the user clicks the
button, we create a callback function for that button. Open the widget properties dialog for the button. In the "C++"
tab, we find the input field "Callback:".

The callback is called exactly once every time the user clicks the button. Our strategy here is to read the current
value from the `counter_widget`, increment it by 1, and write it back to `counter_widget`. The FLTK docu-
mentation tells us that we can use `Fl_Output::ivalue()` to get text in `Fl_Output` as an integer, and we
can write it back by calling `Fl_Output::value(int)`. When the value is changed, FLTK will automatically
redraw the output widget for us. So here is the callback code:

```
int i = counter_widget->ivalue();
i++;
counter_widget->value(i);
```

That's it. This is a complete interactive desktop application. Compile, link, run, and test it to see how it works.

11.3   Cube View

This tutorial will show you how to generate a complete user interface class with FLUID that is used for the CubeView
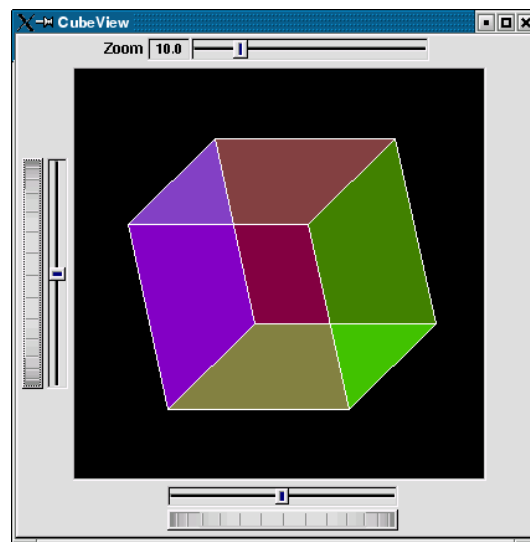program provided with FLTK.

Figure 11.3 CubeView demo

The window is of class CubeViewUI, and is completely generated by FLUID, including class member functions.
The central display of the cube is a separate subclass of Fl_Gl_Window called CubeView. CubeViewUI manages
CubeView using callbacks from the various sliders and rollers to manipulate the viewing angle and zoom of Cube↩
View.

At the completion of this tutorial you will (hopefully) understand how to:

1. Use FLUID to create a complete user interface class, including constructor and any member functions nec-
   essary.

2. Use FLUID to set callback member functions of custom widget classes.

3. Subclass an Fl_Gl_Window to suit your purposes.

11.3.1 The CubeView Class

The CubeView class is a subclass of Fl_Gl_Window. It has methods for setting the zoom, the *x* and *y* pan, and the rotation angle about the *x* and *y* axes.

You can safely skip this section as long as you realize that CubeView is a sublass of Fl_Gl_Window and will respond to calls from CubeViewUI, generated by FLUID.

The CubeView Class Definition

Here is the CubeView class definition, as given by its header file "test/CubeView.h":

```
#include <FL/Fl.H>
#include <FL/Fl_Gl_Window.H>
#include <FL/gl.h>

class CubeView : public Fl_Gl_Window {

public:
  CubeView(int x, int y, int w, int h, const char *l = 0);

  // This value determines the scaling factor used to draw the cube.
  double size;

  /* Set the rotation about the vertical (y) axis.
   *
   * This function is called by the horizontal roller in
   * CubeViewUI and the initialize button in CubeViewUI.
   */
  void v_angle(double angle) { vAng = angle; }

  // Return the rotation about the vertical (y) axis.
  double v_angle() const { return vAng; }

  /* Set the rotation about the horizontal (x) axis.
   *
   * This function is called by the vertical roller in
   * CubeViewUI and the initialize button in CubeViewUI.
   */

  void h_angle(double angle) { hAng = angle; }

  // The rotation about the horizontal (x) axis.
  double h_angle() const { return hAng; }

  /* Sets the x shift of the cube view camera.
   *
   * This function is called by the slider in CubeViewUI
   * and the initialize button in CubeViewUI.
   */
  void panx(double x) { xshift = x; }

  /* Sets the y shift of the cube view camera.
   *
   * This function is called by the slider in CubeViewUI
   * and the initialize button in CubeViewUI.
   */
  void pany(double y) { yshift = y; }

  /* The widget class draw() override.
   *
   * The draw() function initializes Gl for another round of
   * drawing, then calls specialized functions for drawing each
   * of the entities displayed in the cube view.
   */
  void draw();

private:
  /*  Draw the cube boundaries.
   *
   * Draw the faces of the cube using the boxv[] vertices,
   * using GL_LINE_LOOP for the faces.
   */
  void drawCube();

  double vAng, hAng;
```

```
  double xshift, yshift;

  float boxv0[3]; float boxv1[3];
  float boxv2[3]; float boxv3[3];
  float boxv4[3]; float boxv5[3];
  float boxv6[3]; float boxv7[3];
};
```

The CubeView Class Implementation

Here is the CubeView implementation. It is very similar to the "CubeView" demo included with FLTK.

```cpp
#include "CubeView.h"
#include <math.h>

CubeView::CubeView(int x, int y, int w, int h, const char *l)
  : Fl_Gl_Window(x, y, w, h, l)
{
  Fl::use_high_res_GL(1);
  vAng = 0.0;
  hAng = 0.0;
  size = 10.0;
  xshift = 0.0;
  yshift = 0.0;

  /* The cube definition. These are the vertices of a unit cube
   * centered on the origin.*/

  boxv0[0] = -0.5; boxv0[1] = -0.5; boxv0[2] = -0.5;
  boxv1[0] =  0.5; boxv1[1] = -0.5; boxv1[2] = -0.5;
  boxv2[0] =  0.5; boxv2[1] =  0.5; boxv2[2] = -0.5;
  boxv3[0] = -0.5; boxv3[1] =  0.5; boxv3[2] = -0.5;
  boxv4[0] = -0.5; boxv4[1] = -0.5; boxv4[2] =  0.5;
  boxv5[0] =  0.5; boxv5[1] = -0.5; boxv5[2] =  0.5;
  boxv6[0] =  0.5; boxv6[1] =  0.5; boxv6[2] =  0.5;
  boxv7[0] = -0.5; boxv7[1] =  0.5; boxv7[2] =  0.5;
}

void CubeView::drawCube() {
/* Draw a colored cube */
#define ALPHA 0.5
  glShadeModel(GL_FLAT);

  glBegin(GL_QUADS);
    glColor4f(0.0, 0.0, 1.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv1);
    glVertex3fv(boxv2);
    glVertex3fv(boxv3);

    glColor4f(1.0, 1.0, 0.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv4);
    glVertex3fv(boxv5);
    glVertex3fv(boxv1);

    glColor4f(0.0, 1.0, 1.0, ALPHA);
    glVertex3fv(boxv2);
    glVertex3fv(boxv6);
    glVertex3fv(boxv7);
    glVertex3fv(boxv3);

    glColor4f(1.0, 0.0, 0.0, ALPHA);
    glVertex3fv(boxv4);
    glVertex3fv(boxv5);
    glVertex3fv(boxv6);
    glVertex3fv(boxv7);

    glColor4f(1.0, 0.0, 1.0, ALPHA);
    glVertex3fv(boxv0);
    glVertex3fv(boxv3);
    glVertex3fv(boxv7);
    glVertex3fv(boxv4);

    glColor4f(0.0, 1.0, 0.0, ALPHA);
    glVertex3fv(boxv1);
    glVertex3fv(boxv5);
    glVertex3fv(boxv6);
    glVertex3fv(boxv2);
```

```
  glEnd();

  glColor3f(1.0, 1.0, 1.0);
  glBegin(GL_LINES);
    glVertex3fv(boxv0);
    glVertex3fv(boxv1);

    glVertex3fv(boxv1);
    glVertex3fv(boxv2);

    glVertex3fv(boxv2);
    glVertex3fv(boxv3);

    glVertex3fv(boxv3);
    glVertex3fv(boxv0);

    glVertex3fv(boxv4);
    glVertex3fv(boxv5);

    glVertex3fv(boxv5);
    glVertex3fv(boxv6);

    glVertex3fv(boxv6);
    glVertex3fv(boxv7);

    glVertex3fv(boxv7);
    glVertex3fv(boxv4);

    glVertex3fv(boxv0);
    glVertex3fv(boxv4);

    glVertex3fv(boxv1);
    glVertex3fv(boxv5);

    glVertex3fv(boxv2);
    glVertex3fv(boxv6);

    glVertex3fv(boxv3);
    glVertex3fv(boxv7);
  glEnd();
} // drawCube

void CubeView::draw() {
  if (!valid()) {
    glLoadIdentity();
    glViewport(0, 0, pixel_w(), pixel_h());
    glOrtho(-10, 10, -10, 10, -20050, 10000);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  }

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();

  glTranslatef((GLfloat)xshift, (GLfloat)yshift, 0);
  glRotatef((GLfloat)hAng, 0, 1, 0);
  glRotatef((GLfloat)vAng, 1, 0, 0);
  glScalef(float(size), float(size), float(size));

  drawCube();

  glPopMatrix();
}
```

11.3.2   The CubeViewUI Class

We will completely construct a window to display and control the CubeView defined in the previous section using FLUID.

Defining the CubeViewUI Class

Once you have started FLUID, the first step in defining a class is to create a new class within FLUID using the **New->Code->Class** menu item. Name the class "CubeViewUI" and leave the subclass blank. We do not need any inheritance for this window. You should see the new class declaration in the FLUID browser window.
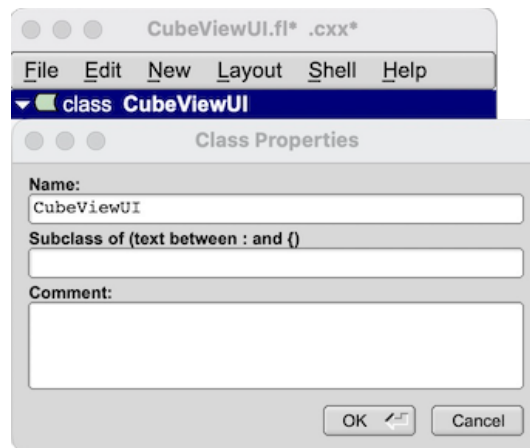
Figure 11.4 FLUID file for CubeView

Adding the Class Constructor

Click on the CubeViewUI class in the FLUID window and add a new method by selecting **New->Code->Function/Method.** The name of the function will also be CubeViewUI. FLUID will understand that this will be the constructor for the class and will generate the appropriate code. Make sure you declare the constructor public.

Then add a window to the CubeViewUI class. Highlight the name of the constructor in the FLUID browser window and click on **New->Group->Window**. In a similar manner add the following to the CubeViewUI constructor:

- A horizontal roller named `hrot`

- A vertical roller named `vrot`

- A horizontal slider named `xpan`

- A vertical slider named `ypan`

- A horizontal value slider named `zoom`

None of these additions need be public. And they shouldn't be unless you plan to expose them as part of the interface for CubeViewUI.
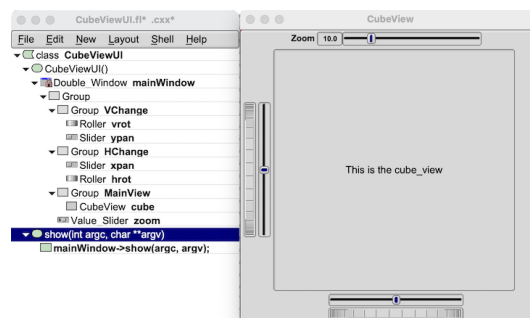
When you are finished you should have something like this:

Figure 11.5 FLUID window containing CubeView demo

We will talk about the `show()` method that is highlighted shortly.

Adding the CubeView Widget

What we have is nice, but does little to show our cube. We have already defined the CubeView class and we would like to show it within the CubeViewUI.

The CubeView class inherits the Fl_Gl_Window class, which is created in the same way as an Fl_Box widget. Use **New->Other->Box** to add a square box to the main window. This will be no ordinary box, however.

The Box properties window will appear. The key to letting CubeViewUI display CubeView is to enter CubeView in the **Class:** text entry box. This tells FLUID that it is not an Fl_Box, but a similar widget with the same constructor.

In the **Extra Code:** field enter `#include "CubeView.h"`

This `#include` is important, as we have just included CubeView as a member of CubeViewUI, so any public CubeView methods are now available to CubeViewUI.
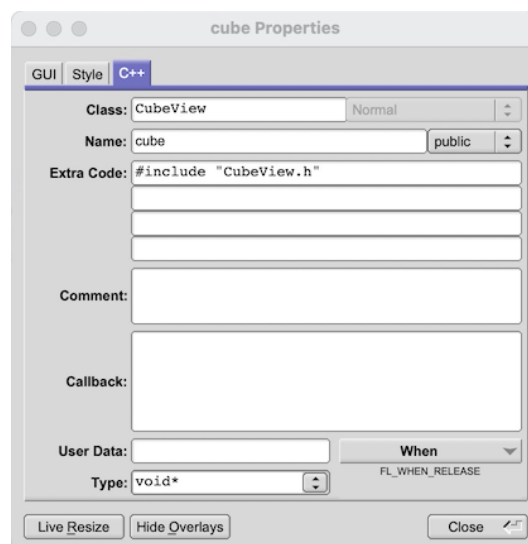
Figure 11.6 CubeView methods

Defining the Callbacks

Each of the widgets we defined before adding CubeView can have callbacks that call CubeView methods. You can call an external function or put a short amount of code in the **Callback** field of the widget panel. For example, the callback for the `ypan` slider is:

```
cube->pany(((Fl_Slider *)o)->value());
cube->redraw();
```

We call `cube->redraw()` after changing the value to update the CubeView window. CubeView could easily be modified to do this, but it is nice to keep this exposed. In the case where you may want to do more than one view change only redrawing once saves a lot of time.

There is no reason to wait until after you have added CubeView to enter these callbacks. FLUID assumes you are smart enough not to refer to members or functions that don't exist.

Adding a Class Method

You can add class methods within FLUID that have nothing to do with the GUI. As an example add a show function so that CubeViewUI can actually appear on the screen.

Make sure the top level CubeViewUI is selected and select **New->Code->Function/Method**. Just use the name `show()`. We don't need a return value here, and since we will not be adding any widgets to this method FLUID will assign it a return type of `void`.
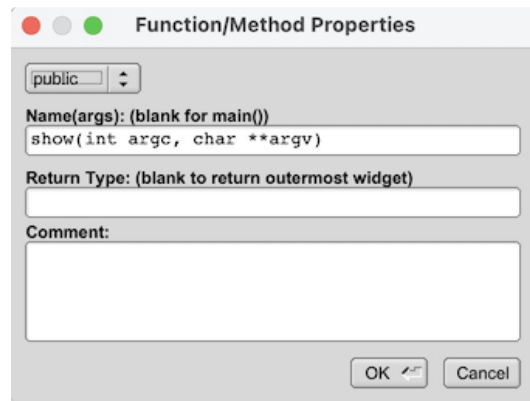

Figure 11.7 CubeView constructor

Once the new method has been added, highlight its name and select **New->Code->Code.** Enter the method's code in the code window.

11.3.3   Adding Constructor Initialization Code

If you need to add code to initialize a class, for example setting initial values of the horizontal and vertical angles in the CubeView, you can simply highlight the constructor and select **New->Code->Code**. Add any required code.

11.3.4   Generating the Code

Now that we have completely defined the CubeViewUI, we have to generate the code. There is one last trick to ensure this all works. Open the preferences dialog from **Edit->Preferences**.

At the bottom of the preferences dialog box is the key: **"Include Header from Code"**. Select that option and set your desired file extensions and you are in business. You can include the CubeViewUI.h (or whatever extension you prefer) as you would any other C++ class.

Chapter 12

Appendices

12.1 Keyboard Shortcuts

On Apple computers, use the Apple Command key instead of Ctrl.

Key Combo	Function
F1	open widget dialog
F2	move widget earlier in tree
F3	move widget later in tree
F7	move widgets into group
F8	ungroup widgets
Delete	delete selected widgets
Ctrl-1..9	load project from history
Ctrl-A	select all
Shift-Ctrl-A	select none
Alt-B	show or hide Widget Bin
Ctrl-C	copy widgets
Alt-C	show or hide Code View window
Shift-Ctrl-C	generate C++ code files
Ctrl-G	grid setting dialog
Ctrl-I	merge project file into current project
Ctrl-N	start a new project, close the current project
Shift-Ctrl-N	new project from template
Ctrl-O	open project file
Shift-Ctrl-O	toggle overlays
Ctrl-P	print all visible project windows
Alt-P	open FLUID settings dialog
Ctrl-Q	quit FLUID
Ctrl-S	save project
Shift-Ctrl-S	save project with new name
Ctrl-U	duplicate selected widgets
Ctrl-V	paste last copied widgets
Shift-Ctrl-W	write i18n translation file
Ctrl-X	cut selected widgets
Alt-X	show shell command settings

Ctrl-Z	undo
Shift-Ctrl-Z	redo

Action	Function in Layout Editor
`left mouse button (LMB)`	select one widget
`LMB-drag`	select multiple widgets with selection box
`Shift-LMB`	extend widget selection
`Shift-LMB-Drag`	toggle selection in selection box
`Shift-LMB-Drag`	resize window proportionally
`Tab`	select next widget
`Shift-Tab`	select previous widget
`Arrow`	move selected widgets by one unit
`Shift-Arrow`	resize by one unit
`Ctrl-Arrow`	move by grid units
`Shift-Ctrl-Arrow`	resize by grid units

12.2 .fl File Format

FLUID edits and saves its state in `.fl` project files. These files are text, and you can (with care) edit them in a text editor, perhaps to get some special effects. The `.fl` file format is described in detail in the file `fluid/README↩_fl.txt` which is part of the FLTK source code repository.

12.3 External Licenses

FLUID uses graphical images based on the Zendesk Garden Stroke icon set:

https://github.com/zendeskgarden Garden Stroke is licensed under the Apache License, Version 2.0.

FLUID includes templates based on 7GUIs:

7GUIs was created as a spin-off of the master's thesis Comparison of Object-Oriented and Functional Programming for GUI Development by Eugen Kiss at the Human-Computer Interaction group of the Leibniz Universität Hannover in 2014.

With kind permission by Prof. Dr. Michael Rohs.

Chapter 13

Todo List

Page Settings Dialog

Document the exact way the source and header file paths are calculated for interactive FLUID, and for FLUID launched from the command line.

Index