FLTK 1.0.4 Programming Manual

Revision 11

Written by Michael Sweet, Craig P. Earls, and Bill Spitzak

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Preface

This manual describes the Fast Light Tool Kit ("FLTK") version 1.0, a C++ Graphical User Interface ("GUI") toolkit for UNIX and Microsoft Windows. Each of the chapters in this manual is designed as a tutorial for using FLTK, while the appendices provide a convenient reference for all FLTK widgets, functions, and operating system interfaces.

Organization

This manual is organized into the following chapters and appendices:

- Appendix G - Software License

Conventions

The following typeface conventions are used in this manual:

- Function and constant names are shown in **`bold courier type`**
- Code samples and commands are shown in `regular courier type`

Abbreviations

The following abbreviations are used in this manual:

X11

The X Window System version 11.

Xlib

The X Window System interface library.

WIN32

The Microsoft Windows 32-bit Application Programmer's Interface.

Copyrights and Trademarks

1 - Introduction to FLTK

The Fast Light Tool Kit ("FLTK", pronounced "fulltick") is a LGPL'd C++ graphical user interface toolkit for X (UNIX®), OpenGL®, and Microsoft® Windows® NT 4.0, 95, or 98. It was originally developed by Mr. Bill Spitzak and is currently maintained by a small group of developers across the world with a central repository in the US.

History of FLTK

It has always been Bill's belief that the GUI API of all modern systems is much too high level. Toolkits (even FL) are *not* what should be provided and documented as part of an operating system. The system only has to provide arbitrary shaped but featureless windows, a powerful set of graphics drawing calls, and a simple *unalterable* method of delivering events to the owners of the windows. NeXT (if you ignored NextStep) provided this, but they chose to hide it and tried to push their own baroque toolkit instead...

Many of the ideas in FLTK were developed on a NeXT (but *not* using NextStep) in 1987 in a C toolkit Bill called "views". Here he came up with passing events downward in the tree and having the handle routine return a value indicating the used the event, and the table-driven menus. In general he was trying to prove that complex UI ideas could be entirely implemented in a user space toolkit, with no knowledge or support by the system.

After going to film school for a few years, Bill worked at Sun Microsystems on the (doomed) NeWS project. Here he found an even better and cleaner windowing system, and he reimplemented "views" atop that. NeWS did have an unnecessarily complex method of delivering events which hurt it. But the designers did admit that perhaps the user could write just as good of a button as they could, and officially exposed the lower level interface.

With the death of NeWS Bill realized that he would have to live with X. The biggest problem with X is the "window manager", which means that the toolkit can no longer control the window borders or drag the window around.

At Digital Domain Bill discovered another toolkit, "Forms". Forms was similar to his work, but provided many more widgets, since it was used in many real applications, rather then as theoretical work. He decided to use Forms, except he integrated his table-driven menus into it. Several very large programs were created using this version of Forms.

The need to switch to OpenGL and GLX, portability, and a desire to use C++ subclassing required a rewrite of Forms. This produced the first version of FLTK. The conversion to C++ required so many changes it made it impossible to recompile any Forms objects. Since it was incompatible anyway, Bill decided to incorporate his older ideas as much as possible by simplifying the lower level interface and the event passing mechanisim.

Bill received permission to release it for free on the Internet, with the GNU general public license. Response from Internet users indicated that the Linux market dwarfed the SGI and high-speed GL market, so he rewrote it to use X for all drawing, greatly speeding it up on these machines. That is the version you have now.

Digital Domain has since withdrawn support for FLTK. While Bill is no longer able to actively develop it, he still contributes to FLTK in his free time and is a part of the FLTK development team.

Features

FLTK was designed to be statically linked. This was done by splitting it into many small objects and desigining it so that functions that are not used do not have pointers to them in the parts that are used, and thus do not get linked in. This allows you to make an easy-to-install program, or to modify FLTK to the exact requirements of your application, without worrying about bloat. FLTK works fine as a shared library, though, and has started being included on Linux distributions.

Here are some of the core features unique to FLTK:

- sizeof(Fl_Widget) == 40 to 48.
- The "core" (the "hello" program compiled & linked with a static FLTK library using gcc on a 486 and then stripped) is 110K.
- The FLUID program (which includes every widget) is 372k.
- Written directly atop Xlib (or WIN32) for maximum speed, and carefully optimized for code size and performance.
- Precise low-level compatability between the X11 and WIN32 version (only about 10% of the code is different).
- Interactive user interface builder program. Output is human-readable and editable C++ source code.
- Support for the X11 double buffering extension (emulation if not available and under Windows.)
- Support for X11 overlay hardware (emulation if none and under WIN32.)
- Very small & fast portable 2-D drawing library to hide Xlib and WIN32.
- OpenGL/Mesa drawing area widget.
- Support for OpenGL overlay hardware on both X11 and WIN32. Emulation if none.
- Text input fields with Emacs key bindings, X cut & paste, and foreign letter compose!
- Compatibility header file for the GLUT library.
- Compatibility header file for the XForms library.

- Much too much to list here...

Licensing

FLTK comes with complete free source code. FLTK is available under the terms of the GNU Library General Public License. Contrary to popular belief, it can be used in commercial software! (Even Bill Gates could use it.)

What Does "FLTK" Mean?

FLTK was originally designed to be compatible with the Forms Library written for SGI machines. In that library all the functions and structures started with "fl_". This naming was extended to all new methods and widgets in the C++ library, and this prefix was taken as the name of the library. It is almost impossible to search for "FL" on the Internet, due to the fact that it is also the abbreviation for Florida. After much debating and searching for a new name for the toolkit, which was already in use by several people, Bill came up with "FLTK", and even a bogus excuse that it stands for "The Fast Light Tool Kit".

Building and Installing FLTK Under UNIX

In most cases you can just type "make". This will run configure with the default of no options and then compile everything.

FLTK uses GNU autoconf to configure itself for your UNIX platform. The main things that the configure script will look for are the X11 and OpenGL (or Mesa) header and library files. If these cannot be found in the standard include/library locations you'll need to define the `CFLAGS`, `CXXFLAGS`, and `LDFLAGS` environment variables. For the Bourne and Korn shells you'd use:

```
CFLAGS=-Iincludedir; export CFLAGS
CXXFLAGS=-Iincludedir; export CXXFLAGS
LDFLAGS=-Llibdir; export LDFLAGS
```

For C shell and tcsh, use:

```
setenv CFLAGS "-Iincludedir"
setenv CXXFLAGS "-Iincludedir"
setenv LDFLAGS "-Llibdir"
```

By default configure will look for a C++ compiler named `CC`, `c++`, `g++`, or `gcc` in that order. To use another compiler you need to set the `CXX` environment variable:

```
CXX=xlC; export xlC
setenv CXX "xlC"
```

The `CC` environment variable can also be used to override the default C compiler (`cc` or `gcc`), which is used for a few FLTK source files.

You can run configure yourself to get the exact setup you need. Type "./configure <options>", where options are:

--enable-debug

Enable debugging code & symbols

--enable-shared

Enable generation of shared libraries

--bindir=/path

Set the location for executables [default = /usr/local/bin]

--libdir=/path

Set the location for libraries [default = /usr/local/lib]

--includedir=/path

Set the location for include files. [default = /usr/local/include]

--prefix=/dir

Set the directory prefix for files [default = /usr/local]

When the configure script is done you can just run the "make" command. This will build the library, FLUID tool, and all of the test programs.

To install the library, become root and type "make install". This will copy the "fluid" executable to "bindir", the header files to "includedir", and the library files to "libdir".

Building FLTK Under Microsoft Windows

There are two ways to build FLTK under Microsoft Windows. The first is to use the Visual C++ 5.0 project files under the "visualc" directory. Just open (or double-click on) the "fltk.dsw" file to get the whole shebang.

The second method is to use a GNU-based development tool with the files in the "makefiles" directory. To build using one of these tools simply copy the appropriate makeinclude and config files to the main directory and do a make:

```
copy makefiles\Makefile.<env> Makefile
make
```

Using the Visual C++ DLL Library

The "fltkdll.dsp" project file builds a DLL-version of the FLTK library. Because of name mangling differences between PC compilers (even between different versions of Visual C++!) you can only use the DLL that is generated with the same version compiler that you built it with.

When compiling an application or DLL that uses the FLTK DLL, you will need to define the

`FL_DLL` preprocessor symbol to get the correct linkage commands embedded within the FLTK header files.

Building FLTK Under OS/2

The current OS/2 build requires XFree86 for OS/2 to work. A native Presentation Manager version has not been implemented yet (volunteers are welcome!).

To build the XFree86 version of FLTK for OS/2, copy the appropriate makeinclude and config files to the main directory and do a make:

```
copy makefiles\Makefile.os2x Makefile
make
```

Internet Resources

FLTK is available on the 'net in a bunch of locations:

WWW

http://www.fltk.org

FTP

ftp://ftp.fltk.org/pub/fltk

ftp://ftp.easysw.com/pub/fltk

ftp://ftp.funet.fi/mirrors/ftp.easysw.com/pub/fltk

ftp.northamerica.net/pub/ESP/fltk

EMail

fltk@fltk.org [see instructions below]

fltk-bugs@fltk.org [for reporting bugs]

To send a message to the FLTK mailing list ("fltk@fltk.org") you must first join the list. Non-member submissions are blocked to avoid problems with unsolicited email.

To join the FLTK mailing list, send a message to "majordomo@fltk.org" with "subscribe fltk" in the message body. A digest of this list is available by subscribing to the "fltk-digest" mailing list.

Reporting Bugs

To report a bug in FLTK, send an email to "fltk-bugs@fltk.org". Please include the FLTK version, operating system & version, and compiler that you are using when describing the bug or problem.

For general support and questions, please use the FLTK mailing list at "fltk@fltk.org".

2 - FLTK Basics

This chapter will teach you the basics of compiling programs that use FLTK.

Naming

All public symbols in FLTK start with the characters 'F' and 'L':

- Functions are either `Fl::foo()` or `fl_foo()`.
- Class and type names are capitalized: `Fl_Foo`.
- Constants and enumerations are uppercase: `FL_FOO`.
- All header files start with <FL/...>.

Header Files

The proper way to include FLTK header files is:

```
#include <FL/Fl_xyz.H>
```

Microsoft Windows developers please note: case *is* significant under other operating systems, and the C standard uses the forward slash (/) to separate directories. *Do not do any of the following:*

```
#include <FL\Fl_xyz.H>
#include <fl/fl_xyz.h>
#include <Fl/fl_xyz.h>
```

Compiling Programs with Standard Compilers

Under UNIX (and under Microsoft Windows when using the GNU development tools) you will probably need to tell the compiler where to find the header files. This is usually done using the `-I` option:

```
CC -I/usr/local/include ...
gcc -I/usr/local/include ...
```

Similarly, when linking your application you will need to tell the compiler to use the FLTK library:

```
CC ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
gcc ... -L/usr/local/lib -lfltk -lXext -lX11 -lm
```

Compiling Programs with Microsoft Visual C++

In Visual C++ you will need to tell the compiler where to find the FLTK header files. This can be done by selecting "Settings" from the "Project" menu and then changing the "Preprocessor" settings under the "C/C++" tab. Similarly, you will need to add the FLTK library to the "Link" settings.

You can build your Microsoft Windows applications as Console or WIN32 applications. If you want to use the standard C `main()` function as the entry point, FLTK includes a `WinMain()` function that will call your `main()` function for you.

Note: The Visual C++ 5.0 optimizer is known to cause problems with many programs. We only recommend using the "Favor Small Code" optimization setting. The Visual C++ 6.0 optimizer seems to be much better and can be used with the "optimized for speed" setting.

Writing Your First FLTK Program

All programs must include the file `<FL/Fl.H>`. In addition the program must include a header file for each FLTK class it uses. Listing 1 shows a simple "Hello, World!" program that uses FLTK to display the window.

Listing 1 - "hello.cxx"

```
#include <FL/Fl.H>
#include <FL/Fl_Window.H>
#include <FL/Fl_Box.H>

int main(int argc, char **argv) {
  Fl_Window *window = new Fl_Window(300,180);
  Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
  box->box(FL_UP_BOX);
  box->labelsize(36);
  box->labelfont(FL_BOLD+FL_ITALIC);
  box->labeltype(FL_SHADOW_LABEL);
  window->end();
  window->show(argc, argv);
  return Fl::run();
}
```

After including the required header files, the program then creates a window:

```
Fl_Window *window = new Fl_Window(300,180);
```

and a box with the "Hello, World!" string in it:

```
Fl_Box *box = new Fl_Box(20,40,260,100,"Hello, World!");
```

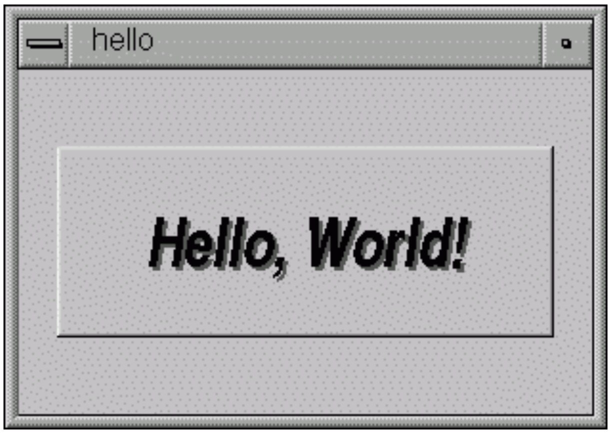
Next, we set the type of box and the size, font, and style of the label:

```
box->box(FL_UP_BOX);
box->labelsize(36);
box->labelfont(FL_BOLD+FL_ITALIC);
box->labeltype(FL_SHADOW_LABEL);
```

Finally, we show the window and enter the FLTK event loop:

```
window->end();
window->show(argc, argv);
return Fl::run();
```

The resulting program will display the window below. You can quit the program by closing the window or pressing the ESCape key.



Creating the Widgets

The widgets are created using the C++ `new` operator. For most widgets the arguments to the constructor are:

```
Fl_Widget(x, y, width, height)
```

The `x` and `y` parameters determine where the widget or window is placed on the screen. In FLTK the top left corner of the window or screen is the origin (i.e. x = 0, y = 0) and the units are in pixels.

The `width` and `height` parameters determine the size of the widget or window in pixels. The maximum widget size is typically governed by the underlying window system or hardware.

`label` is a pointer to a character string to label the widget with or `NULL`. If not specified the label defaults to `NULL`. The label string must be in static storage (such as a string constant) because FLTK does not make a copy of it (it just uses the pointer).

Get/Set Methods

`box->box(FL_UP_BOX)` sets the type of box the Fl_Box draws, changing it from the default of `FL_NO_BOX`, which means that no box is drawn. In our "Hello, World!" example we use `FL_UP_BOX`, which means that a raised button border will be drawn around the widget. You can learn more about boxtypes in Chapter 3.

You could examine the boxtype in by doing `box->box()`. Fltk uses method name overloading to make short names for get/set methods. A "set" method is always of the form "void name(type)", and a "get" method is always of the form "type name() const".

Almost all of these set/get pairs are very fast and short inline functions and thus very efficient. However, *the "set" methods do not call redraw()*, you have to call it yourself. This greatly reduces code size and execution time. The only common exception is `value()`, this does redraw() if necessary.

Labels

All widgets support labels. In the case of window widgets, the label is used for the label in the title bar. Our example program calls the `labelfont`, `labelsize`, and `labeltype` methods.

The `labelfont` method sets the typeface and style that is used for the label, which for this example we are using `FL_BOLD` and `FL_ITALIC`. You can also specify typefaces directly.

The `labelsize` method sets the height of the font in pixels.

The `labeltype` method sets the type of label. FLTK supports normal, embossed, shadowed, symbol, and image labels.

A complete list of all label options can be found in Chapter 3.

Showing the Window

The `show()` method shows the widget or window. For windows you can also provide the command-line arguments to allow users to customize the appearance, size, and position of your windows.

The Main Event Loop

FLTK provides the `Fl:run()` method to enter a standard event processing loop. This is equivalent to the following code:

```
while (Fl::wait());
```

`Fl::run()` does not return until all of the windows under FLTK control are closed (either by the user or your program).

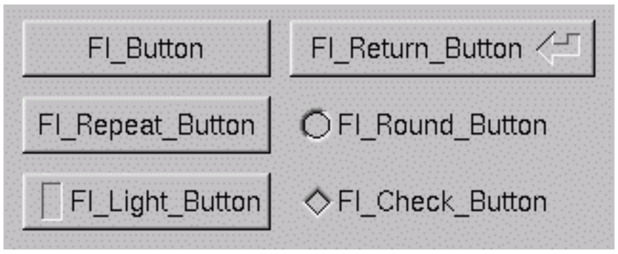
3 - Common Widgets and Attributes

This chapter describes many of the widgets that are provided with FLTK and covers how to query and set the standard attributes.

Buttons

FLTK provides many types of buttons:

- `Fl_Button` - A standard push button.
- `Fl_Check_Button` - A button with a check box.
- `Fl_Light_Button` - A push button with a light.
- `Fl_Repeat_Button` - A push button that repeats when held.
- `Fl_Return_Button` - A push button that is activated by the Enter key.
- `Fl_Round_Button` - A button with a check circle.



For all of these buttons you just need to include the corresponding `<FL/Fl_xyz_Button.H>` header file.

The constructor takes the bounding box of the button and optionally a label string:

```
Fl_Button *button = new Fl_Button(x, y, width, height, "label");
Fl_Light_Button *lbutton = new Fl_Light_Button(x, y, width, height);
Fl_Round_Button *rbutton = new Fl_Round_Button(x, y, width, height, "label");
```

Each button has an associated `type()` which allows it to behave as a push button, toggle button, or radio button:

```
button->type(0);
lbutton->type(FL_TOGGLE_BUTTON);
rbutton->type(FL_RADIO_BUTTON);
```

For toggle and radio buttons, the `value()` method returns the current button state (0 = off, 1 = on). The `set()` and `clear()` methods can be used on toggle buttons to turn a toggle button on or off, respectively. Radio buttons can be turned on with the `setonly()` method; this will also turn off other radio buttons in the same group.

Text

FLTK provides several text widgets for displaying and receiving text:

- `Fl_Input` - A standard one-line text input field.
- `Fl_Output` - A standard one-line text output field.
- `Fl_Multiline_Input` - A standard multi-line text input field.
- `Fl_Multiline_Output` - A standard multi-line text output field.

The `Fl_Output` and `Fl_Multiline_Output` widgets allow the user to copy text from the output field but not change it.

The `value()` method is used to get or set the string that is displayed:

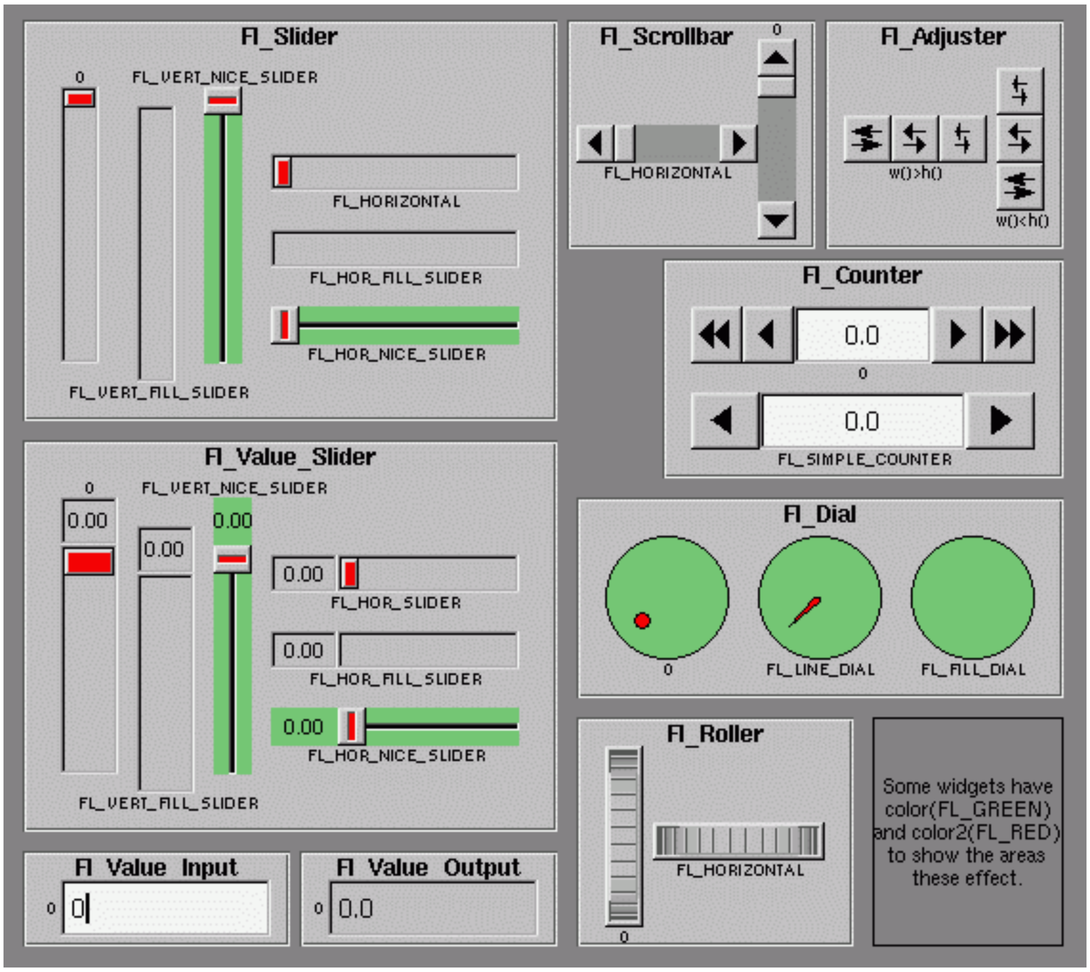
```
Fl_Input *input = new Fl_Input(x, y, width, height, "label");
input->value("Now is the time for all good men...");
```

The string is copied to the widget's own storage when you set the `value()` of the widget.

Valuators

Unlike text widgets, valuators keep track of numbers instead of strings. FLTK provides the following valuators:

- `Fl_Counter` - A widget with arrow buttons that shows the current value.
- `Fl_Dial` - A round knob.
- `Fl_Roller` - An SGI-like dolly widget.
- `Fl_Scrollbar` - A standard scrollbar widget.
- `Fl_Slider` - A scrollbar with a knob.
- `Fl_Value_Slider` - A slider that shows the current value.



The `value()` method gets and sets the current value of the widget. The `minimum()` and `maximum()` methods set the range of values that are reported by the widget.

Groups

The `Fl_Group` widget class is used as a general purpose "container" widget. Besides grouping radio buttons, the groups are used to encapsulate windows, tabs, and scrolled windows. The following group classes are available with FLTK:

- `Fl_Double_Window` - A double-buffered window on the screen.
- `Fl_Gl_Window` - An OpenGL window on the screen.
- `Fl_Group` - The base container class; can be used to group any widgets together.
- `Fl_Scroll` - A scrolled window area.
- `Fl_Tabs` - Displays child widgets as tabs.
- `Fl_Window` - A window on the screen.

Setting the Size and Position of Widgets

The size and position of widgets is usually set when you create them. You can access them with the `x()`, `y()`, `w()`, and `h()` methods.

You can change the size and position by using the `position()`, `resize()`, and `size()` methods:

```
button->position(x, y);
group->resize(x, y, width, height);
window->size(width, height);
```

If you change a widget's size or position after it is displayed you will have to call `redraw()` on the widget's parent.

Colors

FLTK stores the colors of widgets as an 8-bit number that is an index into a color palette of 256 colors. This is *not* the X or WIN32 colormap, but instead is an internal table with fixed contents.

There are symbols for naming some of the more common colors:

- `FL_BLACK` (this is the default label color)
- `FL_RED`
- `FL_GREEN`
- `FL_YELLOW`
- `FL_BLUE`
- `FL_MAGENTA`
- `FL_CYAN`
- `FL_WHITE` (this is the default background color of text widgets)
- `FL_GRAY` (this is the default background color of most widgets)

The widget color can be set using the `color()` method:

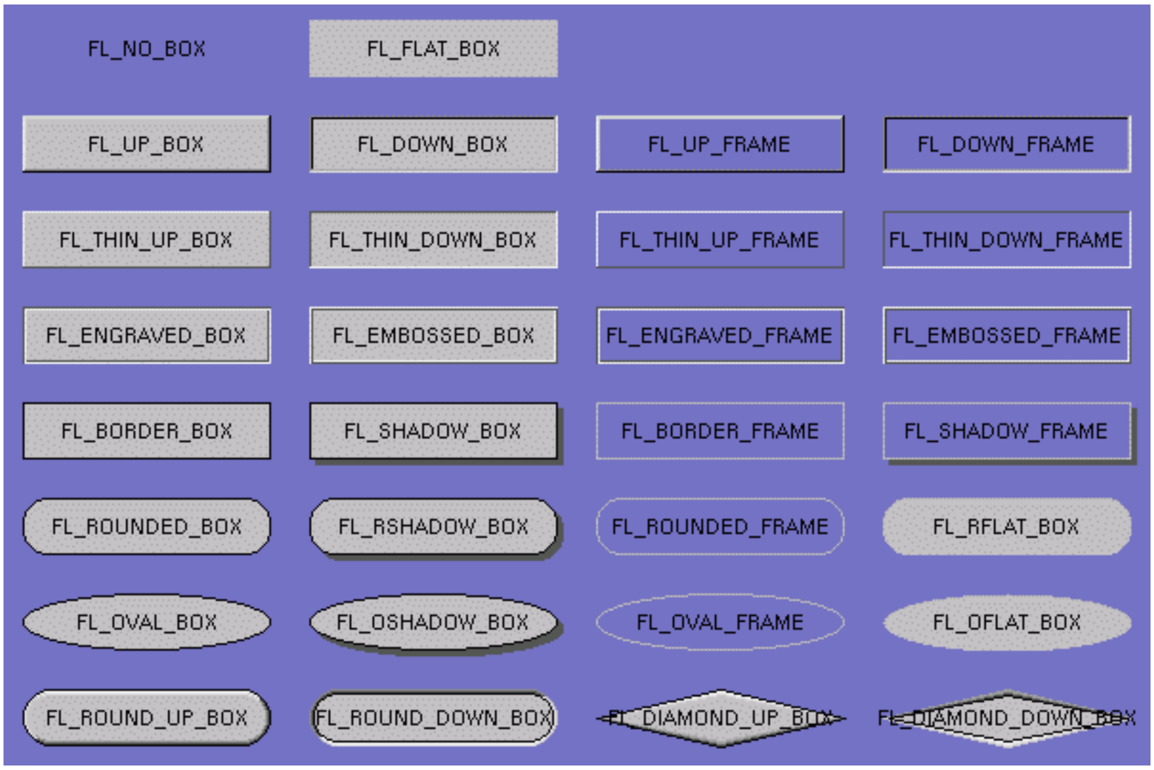
```
button->color(FL_RED);
```

Similarly, the label color can be set using the `labelcolor()` method:

```
button->labelcolor(FL_WHITE);
```

Box Types

The type `Fl_Boxtype` stored and returned in `Fl_Widget::box()` is an enumeration defined in `<Enumerations.H>`:



`FL_NO_BOX` means nothing is drawn at all, so whatever is already on the screen remains. The `FL_..._FRAME` types only draw their edges, leaving the interior unchanged. In the above diagram the blue color is the area that is not drawn by the box.

Making your own Boxtypes

Warning: this interface may change in future versions of fltk!

You can define your own boxtypes by making a small function that draws the box and adding it to the table of boxtypes.

The Drawing Function

The drawing function is passed the bounding box and background color for the widget:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
...
}
```

A simple drawing function might fill a rectangle with the given color and then draw a black outline:

```
void xyz_draw(int x, int y, int w, int h, Fl_Color c) {
  fl_color(c);
  fl_rectf(x, y, w, h);
  fl_color(FL_BLACK);
  fl_rect(x, y, w, h);
}
```

Adding Your Box Type

The `Fl::set_boxtype()` method adds or replaces the specified box type:

```
#define XYZ_BOX FL_FREE_BOXTYPE

Fl::set_boxtype(XYZ_BOX, xyz_draw, 1, 1, 2, 2);
```

The last 4 arguments to `Fl::set_boxtype()` are the offsets for the bounding box that should be subtracted when drawing the label inside the box.

Labels and Label Types

The `label()`, `align()`, `labelfont()`, `labelsize()`, and `labeltype()` methods control the labeling of widgets.

label()

The `label()` method sets the string that is displayed for the label. For the `FL_SYMBOL_LABEL` and image label types the string contains the actual symbol or image data.

align()

The `align()` method positions the label. The following constants are defined (they may be OR'd together as needed):

- `FL_ALIGN_CENTER` - center the label in the widget.
- `FL_ALIGN_TOP` - align the label at the top of the widget.
- `FL_ALIGN_BOTTOM` - align the label at the bottom of the widget.
- `FL_ALIGN_LEFT` - align the label to the left of the widget.
- `FL_ALIGN_RIGHT` - align the label to the right of the widget.
- `FL_ALIGN_INSIDE` - align the label inside the widget.
- `FL_ALIGN_CLIP` - clip the label to the widget's bounding box.
- `FL_ALIGN_WRAP` - wrap the label text as needed.

labeltype()

The `labeltype()` method sets the type of the label. The following standard label types are included:

- `FL_NORMAL_LABEL` - draws the text.
- `FL_NO_LABEL` - does nothing
- `FL_SYMBOL_LABEL` - draws "@xyz" labels, see "Symbol Labels"
- `FL_SHADOW_LABEL` - draws a drop shadow under the text
- `FL_ENGRAVED_LABEL` - draws edges as though the text is engraved
- `FL_EMBOSSED_LABEL` - draws edges as thought the text is raised

To make bitmaps or pixmaps you use a method on the `Fl_Bitmap` or `Fl_Pixmap` objects.

Making Your Own Label Types

Warning: this interface may change in future versions of fltk!

Label types are actually indexes into a table of functions that draw them. The primary purpose of this is to let you reuse the `label()` pointer as a pointer to arbitrary data such as a bitmap or pixmap. You can also use this to draw the labels in ways inaccessible through the `fl_font` mechanisim (e.g. `FL_ENGRAVED_LABEL`) or with program-generated letters or symbology.

Label Type Functions

To setup your own label type you will need to write two functions to draw and measure the label. The draw function is called with a pointer to a `Fl_Label` structure containing the label information, the bounding box for the label, and the label alignment:

```
void xyz_draw(Fl_Label *label, int x, int y, int w, int h, Fl_Align align) {
...
}
```

The label should be drawn *inside* this bounding box, even if `FL_ALIGN_INSIDE` is not enabled. The function is not called if the label value is `NULL`.

The measure function is called with a pointer to a `Fl_Label` structure and references to the width and height:

```
void xyz_measure(Fl_Label *label, int &w, int &h) {
...
}
```

It should measure the size of the label and set `w` and `h` to the size it will occupy.

Adding Your Label Type

The `Fl::set_labeltype` method creates a label type using your draw and measure functions:

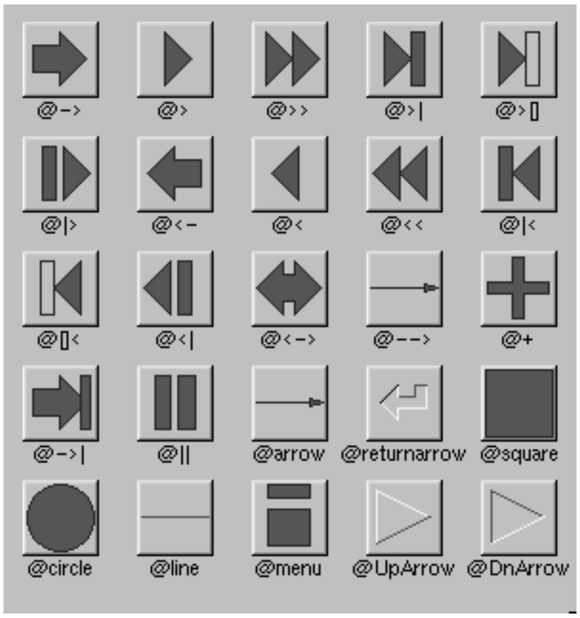
```
#define XYZ_LABEL FL_FREE_LABELTYPE

Fl::set_labeltype(XYZ_LABEL, xyz_draw, xyz_measure);
```

The label type number `n` can be any integer value starting at the constant `FL_FREE_LABELTYPE`. Once you have added the label type you can use the `labeltype()` method to select your label type.

The `Fl::set_labeltype` method can also be used to overload an existing label type such as `FL_NORMAL_LABEL`.

Symbol Labels

The `FL_SYMBOL_LABEL` label type uses the `label()` string to look up a small drawing procedure in a hash table. For historical reasons the string always starts with '@'; if it starts with something else (or the symbol is not found) the label is drawn normally:



The @ sign may be followed by the following optional "formatting" characters, in this order:

- '#' forces square scaling, rather than distortion to the widget's shape.
- +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
- [1-9] - rotates by a multiple of 45 degrees. '6' does nothing, the others point in the direction of that key on a numeric keypad.

Callbacks

Callbacks are functions that are called when the value of a widget changes. A callback function is sent a `Fl_Widget` pointer of the widget that changed and optionally a pointer to data of some sort:

```
void xyz_callback(Fl_Widget *w, void *data) {
...
}
```

The `callback()` method sets the callback function for a widget. You can optionally pass a pointer to some data needed for the callback:

```
int xyz_data;

button->callback(xyz_callback, data);
```

Normally callbacks are performed only when the value of the widget changes. You can change this using the `when()` method:

```
button->when(FL_WHEN_NEVER);
button->when(FL_WHEN_CHANGED);
button->when(FL_WHEN_RELEASE);
button->when(FL_WHEN_RELEASE_ALWAYS);
button->when(FL_WHEN_ENTER_KEY);
button->when(FL_WHEN_ENTER_KEY_ALWAYS);
button->when(FL_WHEN_CHANGED | FL_WHEN_NOT_CHANGED);
```

Shortcuts

Shortcuts are key sequences that activate widgets (usually buttons or menu items). The `shortcut()` method sets the shortcut for a widget:

```
button->shortcut(FL_Enter);
button->shortcut(FL_SHIFT + 'b');
button->shortcut(FL_CTRL + 'b');
button->shortcut(FL_ALT + 'b');
button->shortcut(FL_CTRL + FL_ALT + 'b');
button->shortcut(0); // no shortcut
```

The shortcut value is the key event value (the ASCII value or one of the special keys like `FL_Enter`) combined with any modifiers (like shift, alt, and control).

4 - Designing a Simple Text Editor

This chapter takes you through the design of a simple FLTK-based text editor.

Determining the Goals of the Text Editor

Since this will be the first big project you'll be doing with FLTK, lets define what we want our text editor to do:

1. Menubar/menus for all functions.
2. Edit a single text file.
3. Load from a file.
4. Save to a file.
5. Cut/copy/delete/paste functions.
6. Search and replace functions.
7. Keep track of when the file has been changed.

Designing the Main Window

Now that we've outlined the goals for our editor, we can begin with the design of our GUI. Obviously the first thing that we need is a window:

```
Fl_Window *window;

window = new Fl_Window(640, 480, "Text Editor");
```

Variables

Our text editor will need some global variables to keep track of things:

```
Fl_Window          *window;
Fl_Menu_Bar        *menubar;
Fl_Multiline_Input *input;
Fl_Window          *replace_dlg;
Fl_Input           *replace_find;
Fl_Input           *replace_with;
Fl_Button          *replace_all;
Fl_Return_Button   *replace_next;
Fl_Button          *replace_cancel;

int                changed = 0;
char               filename[1024] = "";
char               search[256] = "";
```

The `window` variable is our top-level window described previously. We'll cover the other variables as we build the application.

Menubars and Menus

The first goal requires us to use a menubar and menus that define each function the editor needs to perform. The `Fl_Menu_Item` structure is used to define the menus and items in a menubar:

```
Fl_Menu_Item menuitems[] = {
  { "&File", 0, 0, 0, FL_SUBMENU },
    { "&New",        FL_ALT + 'n', (Fl_Callback *)new_cb },
    { "&Open...",    FL_ALT + 'o', (Fl_Callback *)open_cb, 0, FL_MENU_DIVIDER },
    { "&Save",       FL_ALT + 's', (Fl_Callback *)save_cb },
    { "Save &As...", FL_ALT + FL_SHIFT + 's', (Fl_Callback *)saveas_cb, 0, FL_MENU_DIVIDER },
    { "&Quit", FL_ALT + 'q', (Fl_Callback *)quit_cb },
    { 0 },

  { "&Edit", 0, 0, 0, FL_SUBMENU },
    { "&Undo",       FL_ALT + 'z', (Fl_Callback *)undo_cb, 0, FL_MENU_DIVIDER },
    { "Cu&t",        FL_ALT + 'x', (Fl_Callback *)cut_cb },
    { "&Copy",       FL_ALT + 'c', (Fl_Callback *)copy_cb },
    { "&Paste",      FL_ALT + 'v', (Fl_Callback *)paste_cb },
    { "&Delete",     0, (Fl_Callback *)delete_cb },
    { 0 },

  { "&Search", 0, 0, 0, FL_SUBMENU },
    { "&Find...",       FL_ALT + 'f', (Fl_Callback *)find_cb },
    { "F&ind Again",    FL_ALT + 'g', (Fl_Callback *)find2_cb },
    { "&Replace...",    FL_ALT + 'r', (Fl_Callback *)replace_cb },
    { "Re&place Again", FL_ALT + 't', (Fl_Callback *)replace2_cb },
    { 0 },

  { 0 }
};
```

Once we have the menus defined we can create the `Fl_Menu_Bar` widget and assign the menus to it with:

```
Fl_Menu_Bar *menubar = new Fl_Menu_Bar(0, 0, 640, 30);
```

```
menubar->menu(menuitems);
```

We'll define the callback functions later.

Editing the Text

To keep things simple our text editor will use the `Fl_Multiline_Input` widget to edit the text:

```
Fl_Multiline_Input *input = new Fl_Multiline_Input(0, 30, 640, 450);
```

So that we can keep track of changes to the file, we also want to add a "changed" callback:

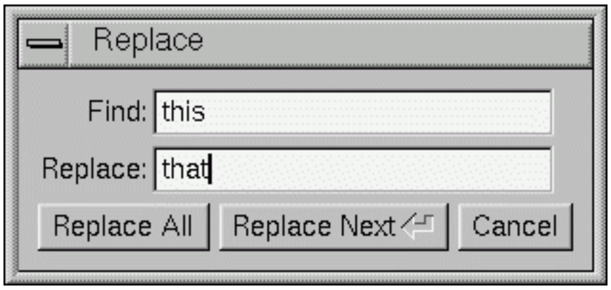
```
input->callback(changed_cb);
input->when(FL_WHEN_CHANGED);
```

Finally, we want to use a mono-spaced font like `FL_COURIER`:

```
input->textfont(FL_COURIER);
```

The Replace Dialog

We can use the FLTK convenience functions for many of the editor's dialogs, however the replace dialog needs its own custom window. To keep things simple we will have a "find" string, a "replace" string, and "replace all", "replace next", and "cancel" buttons. The strings are just `Fl_Input` widgets, the "replace all" and "cancel" buttons are `Fl_Button` widgets, and the "replace next " button is a `Fl_Return_Button` widget:




```
Fl_Window *replace_dlg = new Fl_Window(300, 105, "Replace");
Fl_Input *replace_find = new Fl_Input(70, 10, 200, 25, "Find:");
Fl_Input *replace_with = new Fl_Input(70, 40, 200, 25, "Replace:");
Fl_Button *replace_all = new Fl_Button(10, 70, 90, 25, "Replace All");
Fl_Button *replace_next = new Fl_Button(105, 70, 120, 25, "Replace Next");
Fl_Button *replace_cancel = new Fl_Button(230, 70, 60, 25, "Cancel");
```

Callbacks

Now that we've defined the GUI components of our editor, we need to define our callback functions.

changed_cb()

This function will be called whenever the user changes any text in the `input` widget:

```
void changed_cb(void) {
  set_changed(1);
}
```

The `set_changed()` function is one that we will write to set the changed status on the current file. We're doing it this way because some of the other callbacks will set the changed status to 0, and also because we want to show the changed status in the window's title bar.

copy_cb()

This callback function will call `input->copy()` to copy the currently selected text to the clipboard:

```
void copy_cb(void) {
  input->copy();
}
```

cut_cb()

This callback function will call `input->copy()` to copy the currently selected text to the clipboard and then `input->cut()` to delete it:

```
void cut_cb(void) {
  input->copy();
  input->cut();
}
```

delete_cb()

This callback function will call `input->cut()` to delete the selected text:

```
void delete_cb(void) {
  input->cut();
}
```

find_cb()

This callback function asks for a search string using the `fl_input()` convenience function and then calls the `find2_cb()` function to find the string:

```
void find_cb(void) {
  const char *val;

  val = fl_input("Search String:", search);
  if (val != NULL) {
    // User entered a string - go find it!
    strcpy(search, val);
    find2_cb();
  }
```

```
}
```

find2_cb()

This function will find the next occurrence of the search string. If the search string is blank then we want to pop up the search dialog:

```
void find2_cb(void) {
  const char *val, *found;
  int pos;

  if (search[0] == '\0') {
    // Search string is blank; get a new one...
    find_cb();
    return;
  }

  val   = input->value() + input->mark();
  found = strstr(val, search);

  if (found != NULL) {
    // Found a match; update the position and mark...
    pos = input->mark() + found - val;
    input->position(pos, pos + strlen(search));
  }
  else fl_alert("No occurrences of \'%s\' found!", search);
}
```

If the search string cannot be found we use the `fl_alert()` convenience function to display a message to that effect.

new_cb()

This callback function will clear the input widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void new_cb(void) {
  if (changed)
    if (!check_save()) return;

  filename[0] = '\0';
  input->value("");
  set_changed(0);
}
```

open_cb()

This callback function will ask the user for a filename and then load the specified file into the input widget and current filename. It also calls the `check_save()` function to give the user the opportunity to save the current file first as needed:

```
void open_cb(void) {
  char *newfile;

  if (changed)
```

```
    if (!check_save()) return;

  newfile = fl_file_chooser("Open File?", "*", filename);
  if (newfile != NULL) load_file(newfile);
}
```

We call the `load_file()` function to actually load the file.

paste_cb()

This callback function will send a FL_PASTE message to the `input` widget using the `Fl::paste()` method:

```
void paste_cb(void) {
  Fl::paste(*input);
}
```

quit_cb()

The quit callback will first see if the current file has been modified, and if so give the user a chance to save it. It then hides the main window:

```
void quit_cb(void) {
  if (changed)
    if (!check_save())
      return;

  window->hide();
}
```

replace_cb()

The replace callback just shows the replace dialog:

```
void replace_cb(void) {
  replace_dlg->show();
}
```

replace2_cb()

This callback will replace the next occurence of the replacement string. If nothing has been entered for the replacement string, then the replace dialog is displayed instead:

```
void replace2_cb() {
  const char *find, *val, *found;
  int pos;

  find = replace_find->value();
  if (find[0] == '\0') {
    // Search string is blank; get a new one...
    replace_dlg->show();
    return;
  }
```

```
  val   = input->value() + input->position();
  found = strstr(val, find);

  if (found != NULL) {
    // Found a match; update the position and replace text...
    pos = input->position() + found - val;
    input->replace(pos, pos + strlen(find), replace_with->value());
    input->position(pos + strlen(replace_with->value()));
  }
  else fl_alert("No occurrences of \'%s\' found!", find);
}
```

replall_cb()

This callback will replace all occurences of the search string in the file:

```
void replall_cb() {
  const char *find, *val, *found;
  int pos;
  int times;

  find = replace_find->value();
  if (find[0] == '\0') {
    // Search string is blank; get a new one...
    replace_dlg->show();
    return;
  }

  input->position(0);
  times = 0;

  // Loop through the whole string
  do {
    val   = input->value() + input->position();
    found = strstr(val, find);

    if (found != NULL) {
      // Found a match; update the position and replace text...
      times ++;
      pos = input->position() + found - val;
      input->replace(pos, pos + strlen(find), replace_with->value());
      input->position(pos + strlen(replace_with->value()));
    }
  } while (found != NULL);

  if (times > 0) fl_message("Replaced %d occurrences.", times);
  else fl_alert("No occurrences of \'%s\' found!", find);
}
```

replcan_cb()

This callback just hides the replace dialog:

```
void replcan_cb() {
  replace_dlg->hide();
}
```

save_cb()

This callback saves the current file. If the current filename is blank it calls the "save as" callback:

```
void save_cb(void) {
  if (filename[0] == '\0') {
    // No filename - get one!
    saveas_cb();
    return;
  }
  else save_file(filename);
}
```

The `save_file()` function saves the current file to the specified filename.

saveas_cb()

This callback asks the user for a filename and saves the current file:

```
void saveas_cb(void) {
  char *newfile;

  newfile = fl_file_chooser("Save File As?", "*", filename);
  if (newfile != NULL) save_file(newfile);
}
```

The `save_file()` function saves the current file to the specified filename.

undo_cb()

The undo callback just calls the `undo()` method:

```
void undo_cb(void) {
  input->undo();
}
```

Other Functions

Now that we've defined the callback functions, we need our support functions to make it all work:

check_save()

This function checks to see if the current file needs to be saved. If so, it asks the user if they want to save it:

```
int check_save(void) {
  if (!changed) return 1;

  if (fl_ask("The current file has not been saved.\n"
             "Would you like to save it now?")) {
    // Save the file...
    save_cb();
```

```
    return !changed;
  }
  else return (1);
}
```

load_file()

This function loads the specified file into the `input` widget:

```
void load_file(char *newfile) {
  FILE *fp;
  char buffer[8192];
  int  nbytes;
  int  pos;

  input->value("");

  fp = fopen(newfile, "r");
  if (fp != NULL) {
    // Was able to open file; let's read from it...
    strcpy(filename, newfile);
    pos = 0;

    while ((nbytes = fread(buffer, 1, sizeof(buffer), fp)) > 0) {
      input->replace(pos, pos, buffer, nbytes);
      pos += nbytes;
    }

    fclose(fp);
    input->position(0);
    set_changed(0);
  } else {
    // Couldn't open file - say so...
    fl_alert("Unable to open \'%s\' for reading!");
  }
}
```

When loading the file we use the `input->replace()` method to "replace" the text at the end of the buffer. The `pos` variable keeps track of the end of the buffer.

save_file()

This function saves the current buffer to the specified file:

```
void save_file(char *newfile) {
  FILE *fp;

  fp = fopen(newfile, "w");
  if (fp != NULL) {
    // Was able to create file; let's write to it...
    strcpy(filename, newfile);

    if (fwrite(input->value(), 1, input->size(), fp) < 1) {
      fl_alert("Unable to write file!");
      fclose(fp);
      return;
    }
```

```
    fclose(fp);
    set_changed(0);
  } else {
    // Couldn't open file - say so...
    fl_alert("Unable to create \'%s\' for writing!");
  }
}
```

set_changed()

This function sets the `changed` variable and updates the window label accordingly:

```
void set_changed(int c) {
  if (c != changed) {
    char title[1024];
    char *slash;

    changed = c;

    if (filename[0] == '\0') strcpy(title, "Untitled");
    else {
      slash = strrchr(filename, '/');
      if (slash == NULL) slash = strrchr(filename, '\\');

      if (slash != NULL) strcpy(title, slash + 1);
      else strcpy(title, filename);
    }

    if (changed) strcat(title, " (modified)");

    window->label(title);
  }
}
```

Compiling the Editor

The complete source for our text editor can be found in the `test/editor.cxx` source file. Both the Makefile and Visual C++ workspace include the necessary rules to build the editor. You can also compile it using a standard compiler with:

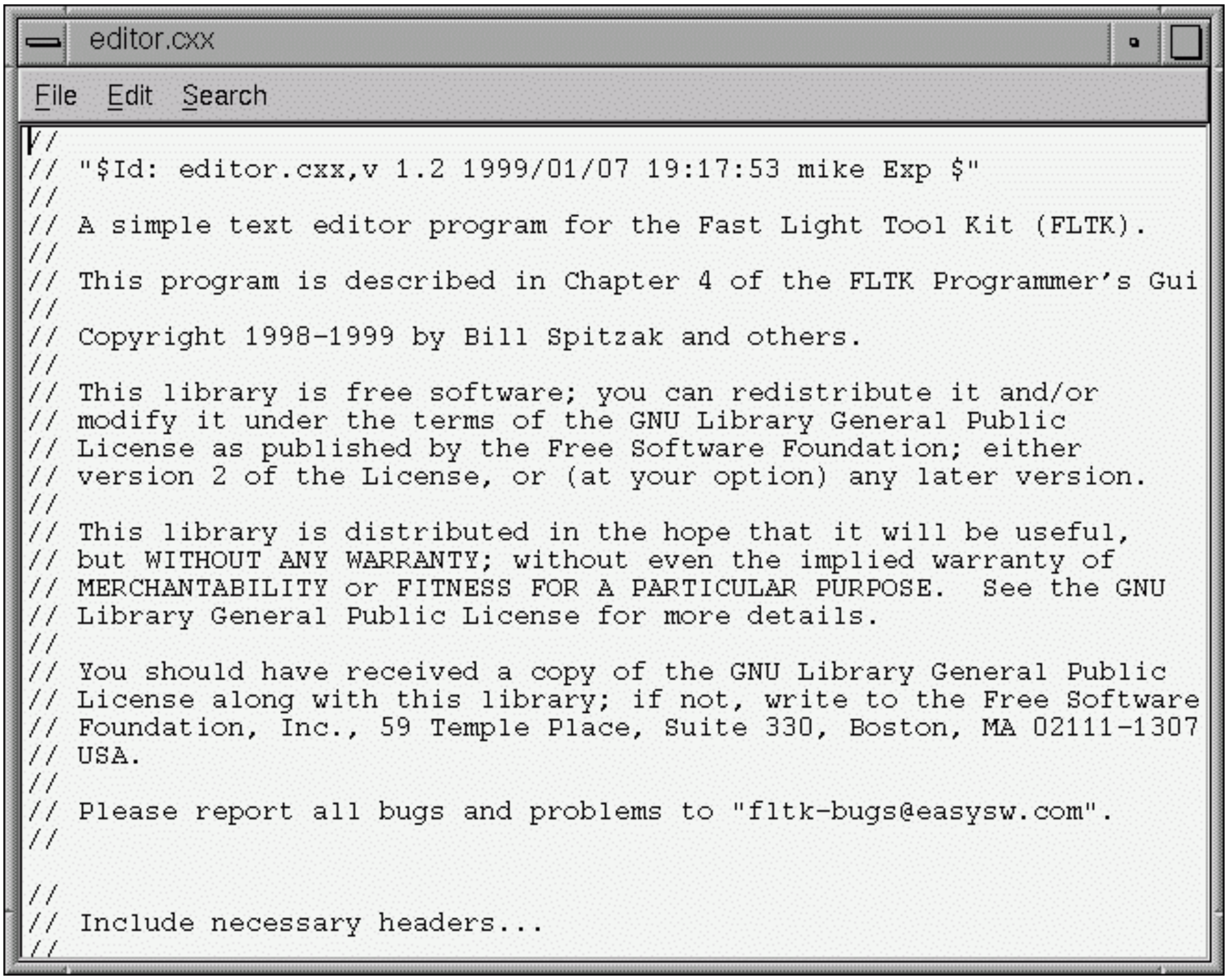
```
CC -o editor editor.cxx -lfltk -lXext -lX11 -lm
```

As noted in Chapter 1, you may need to include compiler and linker options to tell them where to find the FLTK library. Also, the `CC` command may also be called `gcc` or `c++` on your system.

Congratulations, you've just built your own text editor!

The Final Product

The final editor window should look like the image below:


editor.cxx
File Edit Search
//
// "$Id: editor.cxx,v 1.2 1999/01/07 19:17:53 mike Exp $"
//
// A simple text editor program for the Fast Light Tool Kit (FLTK).
//
// This program is described in Chapter 4 of the FLTK Programmer's Gui
//
// Copyright 1998-1999 by Bill Spitzak and others.
//
// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Library General Public
// License as published by the Free Software Foundation; either
// version 2 of the License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
// Library General Public License for more details.
//
// You should have received a copy of the GNU Library General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA.
//
// Please report all bugs and problems to "fltk-bugs@easysw.com".
//
//
// Include necessary headers...
//

5 - Drawing Things in FLTK

This chapter covers the drawing functions that are provided with FLTK.

When Can You Draw Things in FLTK?

There are only certain places you can execute drawing code in FLTK. Calling these functions at other places will result in undefined behavior!

- The most common is inside the virtual method `Fl_Widget::draw()`. To write code here, you must subclass one of the existing `Fl_Widget` classes and implement your own version of `draw()`.
- You can also write boxtypes and labeltypes. These are small procedures that can be called by existing `Fl_Widgetdraw()` methods. These "types" are identified by an 8-bit index that is stored in the widget's `box()` , `labeltype()`, and possibly other properties.
- You can call `Fl_Window::make_current()` to do incremental update of a widget. Use `Fl_Widget::window()` to find the window.

FLTK Drawing Functions

To use the drawing functions you must first include the `<FL/fl_draw.H>` header file. FLTK provides the following types of drawing functions:

- Clipping
- Colors
- Fast Shapes
- Complex Shapes
- Text
- Images
- Overlay

Clipping

You can limit all your drawing to a rectangular region by calling `fl_clip`, and put the drawings back by using `fl_pop_clip`. This rectangle is measured in pixels (it is unaffected by the current transformation matrix).

In addition, the system may provide clipping when updating windows, this clip region may be more complex than a simple rectangle.

void fl_push_clip(int x, int y, int w, int h)

Intersect the current clip region with a rectangle and push this new region onto the stack.

void fl_push_no_clip()

Pushes an empty clip region on the stack so nothing will be clipped.

void fl_pop_clip()

Restore the previous clip region. *You must call `fl_pop_clip()` once for every time you call `fl_clip()`. If you return to FLTK with the clip stack not empty unpredictable results occur.*

int fl_not_clipped(int x, int y, int w, int h)

Returns true if any of the rectangle intersects the current clip region. If this returns false you don't have to draw the object. *Under X this returns 2 if the rectangle is partially clipped, and 1 if it is entirely inside the clip region.*

int fl_clip_box(int x, int y, int w, int h, int &X, int &Y, int &W, int &H)

Intersect the rectangle `x,y,w,h` with the current clip region and returns the bounding box of the result in `X,Y,W,H`. Returns non-zero if the resulting rectangle is different than the original. This can be used to limit the necessary drawing to a rectangle. `W` and `H` are set to zero if the rectangle is completely outside the region.

Colors

void fl_color(Fl_Color)

Set the color for all subsequent drawing operations. `Fl_Color` is an enumeration type, and all values are in the range 0-255. This is *not* the X or WIN32 pixel, it is an index into an internal table! The table provides several general colors, a 24-entry gray ramp, and a 5x8x5 color cube. All of these are named with symbols in `<FL/Enumerations.H>`.

For colormapped displays, a color cell will be allocated out of `fl_colormap` the first time you use a color. If the colormap fills up then a least-squares algorithm is used to find the closest color.

Fl_Color fl_color()

Returns the last `fl_color()` that was set. This can be used for state save/restore.

void fl_color(uchar r, uchar g, uchar b)

Set the color for all subsequent drawing operations. The closest possible match to the RGB color is used. The RGB color is used directly on TrueColor displays. For colormap visuals the nearest index in the gray ramp or color cube is used.

Fast Shapes

These are used to draw almost all the FLTK widgets. They draw on exact pixel boundaries and are as fast as possible, and their behavior will be duplicated exactly on any platform FLTK is ported to. It is undefined whether these are affected by the transformation matrix, so you should only call these while it is the identity.

void fl_rectf(int x, int y, int w, int h)

Color a rectangle that exactly fills the given bounding box.

void fl_rectf(int x, int y, int w, int h, uchar r, uchar g, uchar b)

Color a rectangle with "exactly" the passed `r,g,b` color. On screens with less than 24 bits of color this is done by drawing a solid-colored block using `fl_draw_image()` so that dithering is produced.

void fl_rect(int x, int y, int w, int h)

Draw a 1-pixel border *inside* this bounding box.

void fl_line(int x, int y, int x1, int y1)
void fl_line(int x, int y, int x1, int y1, int x2, int y2)

Draw one or two 1-pixel thick lines between the given points.

void fl_loop(int x, int y, int x1, int y1, int x2, int y2)
void fl_loop(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Outline a 3 or 4-sided polygon with 1-pixel thick lines.

void fl_polygon(int x, int y, int x1, int y1, int x2, int y2)
void fl_polygon(int x, int y, int x1, int y1, int x2, int y2, int x3, int y3)

Fill a 3 or 4-sided polygon. The polygon must be convex.

void fl_xyline(int x, int y, int x1, int y1)
void fl_xyline(int x, int y, int x1, int y1, int x2)
void fl_xyline(int x, int y, int x1, int y1, int x2, int y3)

Draw 1-pixel wide horizontal and vertical lines. A horizontal line is drawn first, then a vertical, then a horizontal.

void fl_yxline(int x, int y, int y1)
void fl_yxline(int x, int y, int y1, int x2)
void fl_yxline(int x, int y, int y1, int x2, int y3)

Draw 1-pixel wide vertical and horizontal lines. A vertical line is drawn first, then a horizontal, then a vertical.

void fl_arc(int x, int y, int w, int h, double a1, double a2)
void fl_pie(int x, int y, int w, int h, double a1, double a2)

High-speed ellipse sections. These functions match the rather limited circle drawing code provided by X and WIN32. The advantage over using `fl_arc` is that they are faster because they often use the hardware, and they draw much nicer small circles, since the small sizes are often hard-coded bitmaps.

If a complete circle is drawn it will fit inside the passed bounding box. The two angles are measured in degrees counterclockwise from 3'oclock and are the starting and ending angle of the arc, `a2` must be greater or equal to `a1`.

`fl_arc()` draws a 1-pixel thick line (notice this has a different number of arguments than the `fl_arc()` described below.

`fl_pie()` draws a filled-in pie slice. This slice may extend outside the line drawn by `fl_arc`, to avoid this use `w - 1` and `h - 1`.

Complex Shapes

These functions let you draw arbitrary shapes with 2-D linear transformations. The functionality matches that found in Adobe® PostScript™. The exact pixels that are filled is less defined than for the previous calls so that FLTK can take advantage of drawing hardware. On both X and WIN32 the transformed vertices are rounded to integers before drawing the line segments: this severely limits the accuracy of these functions for complex graphics, so use OpenGL when greater accuracy and/or performance is required.

void fl_push_matrix()
void fl_pop_matrix()

Save and restore the current transformation. The maximum depth of the stack is 4.

void fl_scale(float x, float y)
void fl_scale(float x)
void fl_translate(float x, float y)
void fl_rotate(float d)
void fl_mult_matrix(float a, float b, float c, float d, float x, float y)

Concatenate another transformation onto the current one. The rotation angle is in degrees (not radians) and is counter-clockwise.

void fl_begin_line()
void fl_end_line()

Start and end drawing 1-pixel thick lines.

void fl_begin_loop()
void fl_end_loop()

Start and end drawing a closed sequence of 1-pixel thick lines.

void fl_begin_polygon()
void fl_end_polygon()

Start and end drawing a convex filled polygon.

void fl_begin_complex_polygon()
void fl_gap()
void fl_end_complex_polygon()

Start and end drawing a complex filled polygon. This polygon may be concave, may have holes in it, or may be several disconnected pieces. Call `fl_gap()` to seperate loops of the path (it is unnecessary but harmless to call `fl_gap()` before the first vertex, after the last one, or several times in a row). For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. This mostly means that holes should be drawn in the opposite direction of the outside.

`fl_gap()` should only be called between `fl_begin_complex_polygon()` and `fl_end_complex_polygon()`. To outline the polygon, use `fl_begin_loop()` and replace each `fl_gap()` with `fl_end_loop();fl_begin_loop()`.

void fl_vertex(float x, float y)

Add a single vertex to the current path.

void fl_curve(float x, float y, float x1, float y1, float x2, float y2, float x3, float y3)

Add a series of points on a Bezier curve to the path. The curve ends (and two of the points) are at `x,y` and `x3,y3`.

void fl_arc(float x, float y, float r, float start, float end)

Add a series of points to the current path on the arc of a circle (you can get elliptical paths by using scale and rotate before calling this). `x,y` are the center of the circle, and `r` is its radius. `fl_arc()` takes `start` and `end` angles that are measured in degrees counter-clockwise from 3 o'clock. If `end` is less than `start` then it draws the arc in a clockwise direction.

void fl_circle(float x, float y, float r)

`fl_circle()` is equivalent to `fl_arc(...,0,360)` but may be faster. It must be the *only* thing in the path: if you want a circle as part of a complex polygon you must use `fl_arc()`. *This draws incorrectly if the transformation is both rotated and non-square scaled.*

Text

All text is drawn in the current font. It is undefined whether this location or the characters are modified by the current transformation.

void fl_draw(const char *, float x, float y)
void fl_draw(const char *, int n, float x, float y)

Draw a nul-terminated string or an array of `n` characters starting at the given location.

void fl_draw(const char *, int x, int y, int w, int h, Fl_Align)

Fancy string drawing function which is used to draw all the labels. The string is formatted and aligned inside the passed box. Handles '\t' and '\n', expands all other control characters to ^X, and aligns inside or against the edges of the box. See `Fl_Widget::align()` for values for `align`. The value `FL_ALIGN_INSIDE` is ignored, as this function always prints inside the box.

void fl_measure(const char *, int &w, int &h)

Measure how wide and tall the string will be when printed by the `fl_draw(...align)` function. If the incoming `w` is non-zero it will wrap to that width.

int fl_height()

Recommended minimum line spacing for the current font. You can also just use the value of `size` passed to `fl_font()`.

int fl_descent()

Recommended distance above the bottom of a `fl_height()` tall box to draw the text at so it looks centered vertically in that box.

float fl_width(const char*)
float fl_width(const char*, int n)
float fl_width(uchar)

Return the pixel width of a nul-terminated string, a sequence of `n` characters, or a single character in the current font.

const char *fl_shortcut_label(ulong)

Unparse a shortcut value as used by `Fl_Button` or `Fl_Menu_Item` into a human-readable string like "Alt+N". This only works if the shortcut is a character key or a numbered function key. If the shortcut is zero an empty string is returned. The return value points at a static buffer that is overwritten with each call.

Fonts

void fl_font(int face, int size)

Set the current font, which is then used by the routines described above. You may call this outside a draw context if necessary to call `fl_width()`, but on X this will open the display.

The font is identified by a `face` and a `size`. The size of the font is measured in `pixels` (not "points"). Lines should be spaced `size` pixels apart (or more).

The `face` is an index into an internal table. Initially only the first 16 faces are filled in. There are symbolic names for them: `FL_HELVETICA`, `FL_TIMES`, `FL_COURIER`, and modifier values `FL_BOLD` and `FL_ITALIC` which can be added to these, and `FL_SYMBOL` and `FL_ZAPF_DINGBATS` . Faces greater than 255 cannot be used in `Fl_Widget` labels, since it stores the index as a byte.

int fl_font()
int fl_size()

Returns the face and size set by the most recent call to `fl_font(a,b)`. This can be used to save/restore the font.

Overlays

void fl_overlay_rect(int x, int y, int w, int h)
void fl_overlay_clear()

These functions allow you to draw interactive selection rectangles without using the overlay hardware. FLTK will XOR a single rectangle outline over a window. Calling this will erase any previous rectangle (by XOR'ing it), and then draw the new one. Calling `fl_overlay_clear()` will erase the rectangle without drawing a new one.

Using this is tricky. You should make a widget with both a `handle()` and `draw()` method. `draw()` should call `fl_overlay_clear()` before doing anything else. Your `handle()` method should call `window()->make_current()` and then `fl_overlay_rect()` after `FL_DRAG` events, and should call `fl_overlay_clear()` after a `FL_RELEASE` event.

Images

To draw images, you can either do it directly from data in your memory, or you can create `Fl_Bitmap`, `Fl_Image`, or `Fl_Pixmap` objects. The advantage of drawing directly is that it is more intuitive, and it is faster if the image data changes more often than it is redrawn. The advantage of using the object is that FLTK will cache translated forms of the image (on X it uses a server pixmap) and thus redrawing is *much* faster.

Direct Image Drawing

It is undefined whether the location or drawing of the image is affected by the current transformation, so you should only call these when it is the identity.

void fl_draw_image(const uchar *, int X, int Y, int W, int H, int D = 3, int LD = 0)
void fl_draw_image_mono(const uchar *, int X, int Y, int W, int H, int D = 1, int LD = 0)

Draw an 8-bit per color RGB or luminance image. The pointer points at the "r" data of the top-left pixel. Data must be in `r,g,b` order. `X,Y` are where to put the top-left corner. `W` and `H` define the size of the image. `D` is the delta to add to the pointer between pixels, it may be any value greater or equal to `3`, or it can be negative to flip the image horizontally. `LD` is the delta to add to the pointer between lines (if 0 is passed it uses `W * D`), and may be larger than `W * D` to crop data, or negative to flip the image vertically.

It is highly recommended that you put the following code before the first `show()` of *any* window in your program to get rid of the dithering if possible:

```
Fl::visual(FL_RGB);
```

Gray scale (1-channel) images may be drawn. This is done if `abs(D)` is less than 3, or by calling `fl_draw_image_mono()`. Only one 8-bit sample is used for each pixel, and on screens with different numbers of bits for red, green, and blue only gray colors are used. Setting `D` greater than 1 will let you display one channel of a color image.

The X version does not support all possible visuals. If FLTK cannot draw the image in the current visual it will abort. FLTK supports any visual of 8 bits or less, and all common TrueColor visuals up to 32 bits.

typedef void (*fl_draw_image_cb)(void *, int x, int y, int w, uchar *)
void fl_draw_image(fl_draw_image_cb, void *, int X, int Y, int W, int H, int D = 3)
void fl_draw_image_mono(fl_draw_image_cb, void *, int X, int Y, int W, int H, int D = 1)

Call the passed function to provide each scan line of the image. This lets you generate the image as it is being drawn, or do arbitrary decompression of stored data (provided it can be decompressed to individual scan lines easily).

The callback is called with the `void *` user data pointer (this can be used to point at a structure of information about the image), and the `x`, `y`, and `w` of the scan line desired from the image. 0,0 is the upper-left corner (*not* `X, Y`). A pointer to a buffer to put the data into is passed. You must copy `w` pixels from scanline `y`, starting at pixel `x` , to this buffer.

Due to cropping, less than the whole image may be requested. So `x` may be greater than zero, the first `y` may be greater than zero, and `w` may be less than `W`. The buffer is long enough to store the entire `W * D` pixels, this is for convienence with some decompression schemes where you must decompress the entire line at once: decompress it into the buffer, and then if `x` is not zero, copy the data over so the `x`'th pixel is at the start of the buffer.

You can assume the `y`'s will be consecutive, except the first one may be greater than zero.

If `D` is 4 or more, you must fill in the unused bytes with zero.

int fl_draw_pixmap(char **data, int X, int Y, Fl_Color = FL_GRAY)

Draws XPM image data, with the top-left corner at the given position. The image is dithered on 8-bit displays so you won't lose color space for programs displaying both images and pixmaps. This function returns zero if there was any error decoding the XPM data.

To use an XPM, do:

```
#include "foo.xpm"
...
fl_draw_pixmap(foo, X, Y);
```

In the current version the XPM data is converted to 24-bit RGB color and passed through `fl_draw_image()`. This is obviously not the most efficient way to do it, and has the same visual limitations as listed above for `fl_draw_image()`. Transparent colors are replaced by the optional `Fl_Color` argument (this may change in the future).

int fl_measure_pixmap(char **data, int &w, int &h)

An XPM image contains the dimensions in its data. This function finds and returns the width and height. The return value is non-zero if it parsed the dimensions ok, and zero if there is any problem.

class Fl_Bitmap

This object encapsulates the width, height, and bits of an X bitmap (XBM), and allows you to make an `Fl_Widget` use a bitmap as a label, or to just draw the bitmap directly. Under X it will create an offscreen pixmap the first time it is drawn, and copy this each subsequent time it is drawn.

Fl_Bitmap(const char *bits, int W, int H)
Fl_Bitmap(const uchar *bits, int W, int H)

Construct using an X bitmap. The bits pointer is simply copied to the object, so it must point at persistent storage. The two constructors are provided because various X implementations disagree about the type of bitmap data. To use an XBM file use:

```
#include "foo.xbm"
...
Fl_Bitmap bitmap = new Fl_Bitmap(foo_bits, foo_width, foo_height);
```

~Fl_Bitmap()

The destructor will destroy any X pixmap created. It does not do anything to the bits data.

void draw(int x, int y, int w, int h, int ox = 0, int oy = 0)

`x,y,w,h` indicates a destination rectangle. `ox,oy,w,h` is a source rectangle. This source rectangle from the bitmap is drawn in the destination. 1 bits are drawn with the current color, 0 bits are unchanged. The source rectangle may extend outside the bitmap (i.e. `ox` and `oy` may be negative and `w` and `h` may be bigger than the bitmap) and this area is left unchanged.

void draw(int x, int y)

Draws the bitmap with the upper-left corner at `x,y`. This is the same as doing `draw(x,y,this->w,this->h,0,0)`.

void label(Fl_Widget *)

Change the `label()` and the `labeltype()` of the widget to draw the bitmap. 1 bits will be drawn with the `labelcolor()` , zero bits will be unchanged. You can use the same bitmap for many widgets.

class Fl_Pixmap

This object encapsulates the data from an XPM image, and allows you to make an `Fl_Widget` use a pixmap as a label, or to just draw the pixmap directly. *Under X it will create an offscreen pixmap the first time it is drawn, and copy this each subsequent time it is drawn* .

The current implementation converts the pixmap to 24-bit RGB data and uses `fl_draw_image()` to draw it. Thus you will get dithered colors on an 8 bit screen.

Fl_Pixmap(char *const* data)

Construct using XPM data. The data pointer is simply copied to the object, so it must point at persistent storage. To use an XPM file do:

```
#include <FL/Fl_Pixmap.H>
#include "foo.xpm"
...
Fl_Pixmap pixmap = new Fl_Pixmap(foo);
```

~Fl_Pixmap()

The destructor will destroy any X pixmap created. It does not do anything to the data.

void draw(int x, int y, int w, int h, int ox = 0, int oy = 0)

`x,y,w,h` indicates a destination rectangle. `ox,oy,w,h` is a source rectangle. This source rectangle is copied to the destination. The source rectangle may extend outside the pixmap (i.e. `ox` and `oy` may be negative and `w` and `h` may be bigger than the pixmap) and this area is left unchanged.

void draw(int x, int y)

Draws the image with the upper-left corner at `x,y`. This is the same as doing `draw(x,y,this->w,this->h,0,0)`.

void label(Fl_Widget *)

Change the `label()` and the `labeltype()` of the widget to draw the pixmap. You can use the same pixmap for many widgets.

class Fl_Image

This object encapsulates a full-color RGB image, and allows you to make an `Fl_Widget` use an image as a label, or to just draw the image directly. *Under X it will create an offscreen pixmap the first time it is drawn, and copy this each subsequent time it is drawn.*

Fl_Image(const uchar *data, int W, int H, int D = 3, int LD = 0)

Construct using a pointer to RGB data. `W` and `H` are the size of the image in pixels. `D` is the delta between pixels (it may be more than 3 to skip alpha or other data, or negative to flip the image left/right). `LD` is the delta between lines (it may be more than `D * W` to crop images, or negative to flip the image vertically). The data pointer is simply copied to the object, so it must point at persistent storage.

~Fl_Image()

The destructor will destroy any X pixmap created. It does not do anything to the data.

void draw(int x, int y, int w, int h, int ox = 0, int oy = 0)

`x,y,w,h` indicates a destination rectangle. `ox,oy,w,h` is a source rectangle. This source rectangle is copied to the destination. The source rectangle may extend outside the image (i.e. `ox` and `oy` may be negative and `w` and `h` may be bigger than the image) and this area is left unchanged.

void draw(int x, int y)

Draws the image with the upper-left corner at `x,y`. This is the same as doing `draw(x,y,this->w,this->h,0,0)`.

void label(Fl_Widget *)

Change the `label()` and the `labeltype()` of the widget to draw the image. You can use the same image for many widgets.

6 - Handling Events

This chapter discusses the FLTK event model and how to handle events in your program or widget.

The FLTK Event Model

Events are identified by the integer argument passed to the `Fl_Widget::handle()` virtual method. Other information about the most recent event is stored in static locations and acquired by calling the `Fl::event_*()` methods. This static information remains valid until the next event is read from window system (i.e. it is ok to look at it outside of the `handle()` method).

Mouse Events

FL_PUSH

A mouse button has gone down with the mouse pointing at this widget. You can find out what button by calling `Fl::event_button()`. You find out the mouse position by calling `Fl::event_x()` and `Fl::event_y()`.

A widget indicates that it "wants" the mouse click by returning non-zero from its `handle()` method. It will then become the `Fl::pushed()` widget and will get `FL_DRAG` and the matching `FL_RELEASE` events. If `handle()` returns zero then FLTK will try sending the `FL_PUSH` to another widget.

FL_DRAG

The mouse has moved with a button held down. The current button state is in `Fl::event_state()`. The mouse position is in `Fl::event_x()` and `Fl::event_y()`.

To receive `FL_DRAG` events you must also respond to the `FL_PUSH` and `FL_RELEASE` events.

FL_RELEASE

A mouse button has been released. You can find out what button by calling `Fl::event_button()`.

FL_MOVE

The mouse has moved without any mouse buttons held down. This event is sent to the `Fl::belowmouse()` widget.

Focus Events

FL_ENTER

The mouse has been moved to point at this widget. This can be used for highlighting feedback. If a widget wants to highlight or otherwise track the mouse, it indicates this by returning non-zero from its `handle()` method. It then becomes the `Fl::belowmouse()` widget and will receive `FL_MOVE` and `FL_LEAVE` events.

FL_LEAVE

The mouse has moved out of the widget.

FL_FOCUS

This indicates an *attempt* to give a widget the keyboard focus.

If a widget wants the focus, it should change itself to display the fact that it has the focus, and return non-zero from its `handle()` method. It then becomes the `Fl::focus()` widget and gets `FL_KEYBOARD` and `FL_UNFOCUS` events.

The focus will change either because the window manager changed which window gets the focus, or because the user tried to navigate using tab, arrows, or other keys. You can check `Fl::event_key()` to figure out why it moved. For navigation it will be the key pressed and for instructions from the window manager it will be zero.

FL_UNFOCUS

Sent to the previous `Fl::focus()` widget when another widget gets the focus.

Keyboard Events

FL_KEYBOARD

A key press. The key pressed can be found in `Fl::event_key()`. The text that the key should insert can be found with `Fl::event_text()` and its length is in `Fl::event_length()`. If you use the key `handle()` should return 1. If you return zero then FLTK assummes you ignored the key. It will then attempt to send it to a parent widget. If none of them want it, it will change the event into a `FL_SHORTCUT` event.

To receive `FL_KEYBOARD` events you must also respond to the `FL_FOCUS` and `FL_UNFOCUS` events.

FL_SHORTCUT

If the `Fl::focus()` widget is zero or ignores an `FL_KEYBOARD` event then FLTK tries sending this event to every widget it can, until one of them returns non-zero. `FL_SHORTCUT` is first sent to the `belowmouse()` widget, then its parents and siblings, and eventually to every widget in the window, trying to find an object that returns non-zero. FLTK tries really hard to not to ignore any keystrokes!

You can also make "global" shortcuts by using `Fl::add_handler()`. A global shortcut will work no matter what windows are displayed or which one has the focus.

Widget Events

FL_DEACTIVATE

This widget is no longer active, due to `deactivate()` being called on it or one of its parents. `active()` may still be true after this, the widget is only active if `active()` is true on it and all its parents (use `active_r()` to check this).

FL_ACTIVATE

This widget is now active, due to `activate()` being called on it or one of its parents.

FL_HIDE

This widget is no longer visible, due to `show()` being called on it or one of its parents, or due to a parent window being restored. *Child `Fl_Windows` respond to this by actually creating the window if not done already, so if you subclass a window, be sure to pass `FL_SHOW` to the base class `handle()` method!*

Clipboard Events

FL_PASTE

You should get this event some time after you call `Fl::paste()`. The contents of `Fl::event_text()` is the text to insert and the number of characters is in `Fl::event_length()`.

FL_SELECTIONCLEAR

The Fl::selection_owner() will get this event before the selection is moved to another widget. This indicates that some other widget or program has claimed the selection.

Fl::event_*() methods

FLTK keeps the information about the most recent event in static storage. This information is good until the next event is processed. Thus it is valid inside `handle()` and `callback()` methods.

These are all trivial inline functions and thus very fast and small:

- `Fl::event_button`
- `Fl::event_clicks`
- `Fl::event_inside`
- `Fl::event_is_click`
- `Fl::event_key`
- `Fl::event_length`
- `Fl::event_state`
- `Fl::event_text`
- `Fl::event_x`
- `Fl::event_x_root`
- `Fl::event_y`
- `Fl::event_y_root`
- `Fl::get_key`
- `Fl::get_mouse`
- `Fl::test_shortcut`

Event Propagation

FLTK follows very simple and unchangeable rules for sending events. The major innovation is that widgets can indicate (by returning 0 from the `handle()` method) that they are not interested in an event, and FLTK can then send that event elsewhere. This eliminates the need for "interests" (event masks or tables), and this is probably the main reason FLTK is much smaller than other toolkits.

Most events are sent directly to the `handle()` method of the `Fl_Window` that the window system says they belong to. The window (actually the `Fl_Group` that `Fl_Window` is a subclass of) is responsible for sending the events on to any child widgets. To make the `Fl_Group` code somewhat easier, FLTK sends some events (`FL_DRAG`, `FL_RELEASE`, `FL_KEYBOARD`, `FL_SHORTCUT`, `FL_UNFOCUS`, and

FL_LEAVE) directly to leaf widgets. These procedures control those leaf widgets:

- Fl::add_handler
- Fl::belowmouse
- Fl::focus
- Fl::grab
- Fl::modal
- Fl::pushed
- Fl::release
- Fl_Widget::take_focus

FLTK Compose-Character Sequences

The Fl_Input widget lets you type all the characters in the standard ISO-8859-1 character set. Most fonts will display these characters correctly.

To insert them, type the [compose] key and then one or two characters. The two characters can be in either order. The [compose] key is any of: Ctrl+Q, the righthand control key, or any key your X server calls XK_Multi_key.

Keys	Char	Keys	Char	Keys	Char	Keys	Char	Keys	Char	Keys	Char
space	nbsp	*	°	A`	À	D-	Ð	a`	à	d-	ð
!	¡	+-	±	A'	Á	N~	Ñ	a'	á	n~	ñ
%	¢	2	²	A^	Â	O`	Ò	a^	â	o`	ò
#	£	3	³	A~	Ã	O'	Ó	a~	ã	o'	ó
$	¤	'	´	A:	Ä	O^	Ô	a:	ä	o^	ô
y=	¥	u	µ	A*	Å	O~	Õ	a*	å	o~	õ
|	¦	p	¶	AE	Æ	O:	Ö	ae	æ	o:	ö
&	§	.	·	C,	Ç	x	×	c,	ç	-:	÷
:	¨	,	¸	E`	È	O/	Ø	e`	è	o/	ø
c	©	1	¹	E'	É	U`	Ù	e'	é	u`	ù
a	ª	o	º	E^	Ê	U'	Ú	e^	ê	u'	ú
<<	«	>>	»	E:	Ë	U^	Û	e:	ë	u^	û
~	¬	14	¼	I`	Ì	U:	Ü	i`	ì	u:	ü
-	-	12	½	I'	Í	Y'	Ý	i'	í	y'	ý
r	®	34	¾	I^	Î	TH	Þ	i^	î	th	þ
_	¯	?	¿	I:	Ï	ss	ß	i:	ï	y:	ÿ

For instance, to type "á" type [compose][a]['] or [compose]['][a].

The character "nbsp" (non-breaking space) is typed by using [compose][space].

The single-character sequences may be followed by a space if necessary to remove ambiguity. For instance, if you really want to type "ª~" rather than "ã" you must type [compose][a][space][~].

If you wish to use the compose function in your own code, your widget's `handle()` method must call `fl_compose()` in response to `FL_KEYPRESS` events:

`int fl_compose(int state, char c, int &del, char *buffer, int &ins)`

Starts or adds a single ASCII character to a compose sequence. This will return the number of old bytes to delete and a set of new bytes to insert, and a new `state` value. If this returns zero you can ignore the result (which just says to insert the character unchanged) and handle the keystroke yourself.

`state` must either be the return value of the last call to `fl_compose()` or zero to start a new compose sequence. Be sure to reset to zero if the user ever moves the cursor.

`c` is the ASCII character that the user typed.

`del` is set to the number of bytes to delete backwards. This will always be less or equal to the `ins` from the last call to `fl_compose()`, and will be zero if `state` is zero.

`buffer` will have the first `ins` bytes set to the data to insert and display (it is *not* nul-terminated).

`ins` will be the number of characters to insert.

7 - Adding and Extending Widgets

This chapter describes how to add your own widgets or extend existing widgets in FLTK.

Subclassing

New widgets are created by *subclassing* an existing FLTK widget, typically `Fl_Widget` for controls and `Fl_Group` for composite widgets.

A control widget typically interacts with the user to receive and/or display a value of some sort.

A composite widget widget holds a list of child widgets and handles moving, sizing, showing, or hiding them as needed. `Fl_Group` is the main composite widget widget class in FLTK, and all of the other composite widgets (`Fl_Pack`, `Fl_Scroll`, `Fl_Tabs`, `Fl_Tile`, and `Fl_Window`) are subclasses of it.

You can also subclass other existing widgets to provide a different look or user-interface. For example, the button widgets are all subclasses of `Fl_Button` since they all interact with the user via a mouse button click. The only difference is the code that draws the face of the button.

Making a Subclass of Fl_Widget

Your subclasses can directly descend from `Fl_Widget` or any subclass of `Fl_Widget`. `Fl_Widget` has only four virtual methods, and overriding some or all of these may be necessary.

The Constructor

The constructor should have the following arguments:

```
MyClass(int x, int y, int w, int h, const char *label = 0);
```

This will allow the class to be used in FLUID without problems.

The constructor must call the constructor for the base class and pass the same arguments:

```
MyClass::MyClass(int x, int y, int w, int h, const char *label)
: Fl_Widget(x, y, w, h, label) {
// do initialization stuff...
}
```

`Fl_Widget`'s protected constructor sets `x()`, `y()`, `w()`, `h()`, and `label()` to the passed values and initializes the other instance variables to:

```
type(0);
box(FL_NO_BOX);
color(FL_GRAY);
selection_color(FL_GRAY);
labeltype(FL_NORMAL_LABEL);
labelstyle(FL_NORMAL_STYLE);
labelsize(FL_NORMAL_SIZE);
labelcolor(FL_BLACK);
align(FL_ALIGN_CENTER);
callback(default_callback,0);
flags(ACTIVE|VISIBLE);
```

Protected Methods of Fl_Widget

The following methods are provided for subclasses to use:

- `Fl_Widget::clear_visible`
- `Fl_Widget::damage`
- `Fl_Widget::draw_box`
- `Fl_Widget::draw_label`
- `Fl_Widget::set_flag`
- `Fl_Widget::set_visible`
- `Fl_Widget::test_shortcut`
- `Fl_Widget::type`

void Fl_Widget::damage(uchar mask)
void Fl_Widget::damage(uchar mask, int x, int y, int w, int h)
uchar Fl_Widget::damage()

The first form indicates that a partial update of the object is needed. The bits in mask are OR'd into `damage()`. Your `draw()` routine can examine these bits to limit what it is drawing. The public method `Fl_Widget::redraw()` simply does `Fl_Widget::damage(FL_DAMAGE_ALL)`, but the implementation of your widget can call the private `damage(n)`.

The second form indicates that a region is damaged. If only these calls are done in a window (no calls to `damage(n)`) then FLTK will clip to the union of all these calls before drawing anything. This can greatly speed up incremental displays. The mask bits are OR'd into `damage()` unless this is a `Fl_Window` widget.

The third form returns the bitwise-OR of all `damage(n)` calls done since the last `draw()`.

When redrawing your widgets you should look at the damage bits to see what parts of your widget need redrawing. The `handle()` method can then set individual damage bits to limit the amount of drawing that needs to be done:

```
MyClass::handle(int event) {
  ...
  if (change_to_part1) damage(1);
  if (change_to_part2) damage(2);
  if (change_to_part3) damage(4);
}

MyClass::draw() {
  if (damage() FL_DAMAGE_ALL) {
    ... draw frame/box and other static stuff ...
  }

  if (damage() (FL_DAMAGE_ALL | 1)) draw_part1();
  if (damage() (FL_DAMAGE_ALL | 2)) draw_part2();
  if (damage() (FL_DAMAGE_ALL | 4)) draw_part3();
}
```

void Fl_Widget::draw_box() const
void Fl_Widget::draw_box(Fl_Boxtype b, ulong c) const

The first form draws this widget's `box()`, using the dimensions of the widget. The second form uses `b` as the box type and `c` as the color for the box.

void Fl_Widget::draw_label() const
void Fl_Widget::draw_label(int x, int y, int w, int h) const
void Fl_Widget::draw_label(int x, int y, int w, int h, Fl_Align align) const

This is the usual function for a `draw()` method to call to draw the widget's label. It does not draw the label if it is supposed to be outside the box (on the assumption that the enclosing group will draw those labels).

The second form uses the passed bounding box instead of the widget's bounding box. This is useful so "centered" labels are aligned with some feature, like a moving slider.

The third form draws the label anywhere. It acts as though `FL_ALIGN_INSIDE` has been forced on so the label will appear inside the passed bounding box. This is designed for parent groups to draw labels with.

void Fl_Widget::set_flag(SHORTCUT_LABEL)

Modifies `draw_label()` so that '&' characters cause an underscore to be printed under the next letter.

void Fl_Widget::set_visible()
void Fl_Widget::clear_visible()

Fast inline versions of `Fl_Widget::hide()` and `Fl_Widget::show()`. These do not send the `FL_HIDE` and `FL_SHOW` events to the widget.

int Fl_Widget::test_shortcut() const
static int Fl_Widget::test_shortcut(const char *s)

The first version tests `Fl_Widget::label()` against the current event (which should be a `FL_SHORTCUT` event). If the label contains a '&' character and the character after it matches the key press, this returns true. This returns false if the `SHORTCUT_LABEL` flag is off, if the label is `NULL` or does not have a '&' character in it, or if the keypress does not match the character.

The second version lets you do this test against an arbitrary string.

uchar Fl_Widget::type() const
void Fl_Widget::type(uchar t)

The property `Fl_Widget::type()` can return an arbitrary 8-bit identifier, and can be set with the protected method `type(uchar t)`. This value had to be provided for Forms compatibility, but you can use it for any purpose you want. Try to keep the value less than 100 to not interfere with reserved values.

FLTK does not use RTTI (Run Time Typing Infomation), to enhance portability. But this may change in the near future if RTTI becomes standard everywhere.

If you don't have RTTI you can use the clumsy FLTK mechanisim, by having `type()` use a unique value. These unique values must be greater than the symbol `FL_RESERVED_TYPE` (which is 100). Look through the header files for `FL_RESERVED_TYPE` to find an unused number. If you make a subclass of `Fl_Window` you must use `FL_WINDOW + n` (`n` must be in the range 1 to 7).

Handling Events

The virtual method `int Fl_Widget::handle(int event)` is called to handle each event passed to the widget. It can:

- Change the state of the widget.
- Call `Fl_Widget::redraw()` if the widget needs to be redisplayed.
- Call `Fl_Widget::damage(n)` if the widget needs a partial-update (assumming you provide support for this in your `Fl_Widget::draw()` method).
- Call `Fl_Widget::do_callback()` if a callback should be generated.
- Call `Fl_Widget::handle()` on child widgets.

Events are identified by the integer argument. Other information about the most recent event is stored in static locations and aquired by calling the `Fl::event_*()` functions. This information remains valid until another event is handled.

Here is a sample `handle()` method for a widget that acts as a pushbutton and also accepts the keystroke 'x' to cause the callback:

```
int MyClass::handle(int event) {
  switch(event) {
    case FL_PUSH:
      highlight = 1;
      redraw();
      return 1;
    case FL_DRAG: {
        int t = Fl::event_inside(this);
        if (t != highlight) {
          highlight = t;
          redraw();
        }
      }
      return 1;
    case FL_RELEASE:
      if (highlight) {
        highlight = 0;
        redraw();
        do_callback();
        // never do anything after a callback, as the callback
        // may delete the widget!
      }
      return 1;
    case FL_SHORTCUT:
      if (Fl::event_key() == 'x') {
        do_callback();
        return 1;
      }
      return 0;
    default:
      return 0;
  }
}
```

You must return non-zero if your `handle()` method uses the event. If you return zero it indicates to the parent widget that it can try sending the event to another widget.

Drawing the Widget

The `draw()` virtual method is called when FLTK wants you to redraw your widget. It will be called if and only if `damage()` is non-zero, and `damage()` will be cleared to zero after it returns. `draw()` should be declared protected, so that it can't be called from non-drawing code.

`damage()` contains the bitwise-OR of all the `damage(n)` calls to this widget since it was last drawn. This can be used for minimal update, by only redrawing the parts whose bits are set. FLTK will turn on the `FL_DAMAGE_ALL` bit if it thinks the entire widget must be redrawn (e.g. for an expose event).

Expose events (and the above `damage(b,x,y,w,h)`) will cause `draw()` to be called with FLTK's clipping turned on. You can greatly speed up redrawing in some cases by testing `fl_not_clipped(x,y,w,h)` or `fl_clip_box(...)` and skipping invisible parts.

Besides the protected methods described above, FLTK provides a large number of basic drawing functions, which are described below.

Resizing the Widget

The `resize(int x, int y, int w, int h)` method is called when the widget is being resized or moved. The arguments are the new position, width, and height. `x()`, `y()`, `w()`, and `h()` still remain the old size. You must call `resize()` on your base class with the same arguments to get the widget size to actually change.

This should *not* call `redraw()`, at least if only the `x()` and `y()` change. This is because composite widgets like `Fl_Scroll` may have a more efficient way of drawing the new position.

Making a Composite Widget

A "composite" widget contains one or more "child" widgets. To make a composite widget you should subclass `Fl_Group` . It is possible to make a composite object that is not a subclass of `Fl_Group`, but you'll have to duplicate the code in `Fl_Group` anyways.

Instances of the child widgets may be included in the parent:

```
class MyClass : public Fl_Group {
  Fl_Button the_button;
  Fl_Slider the_slider;
  ...
};
```

The constructor has to initialize these instances. They are automatically `add()`ed to the group, since the `Fl_Group` constructor does `begin()`. *Don't forget to call `end()` or use the `Fl_End` pseudo-class:*

```
MyClass::MyClass(int x, int y, int w, int h) :
  Fl_Group(x, y, w, h),
  the_button(x + 5, y + 5, 100, 20),
  the_slider(x, y + 50, w, 20)
{
  ...(you could add dynamically created child widgets here)...
  end(); // don't forget to do this!
}
```

The child widgets need callbacks. These will be called with a pointer to the children, but the widget itself may be found in the `parent()` pointer of the child. Usually these callbacks can be static private methods, with a matching private method:

```
void MyClass::slider_cb(Fl_Widget* v, void *) { // static method
  ((MyClass*)(v->parent())->slider_cb();
}
void MyClass::slider_cb() { // normal method
  use(the_slider->value());
}
```

If you make the `handle()` method, you can quickly pass all the events to the children using the `Fl_Group::handle()` method. You don't need to override `handle()` if your composite widget does nothing other than pass events to the children:

```
int MyClass::handle(int event) {
  if (Fl_Group::handle(event)) return 1;
```

```
  ... handle events that children don't want ...
}
```

If you override `draw()` you need to draw all the children. If `redraw()` or `damage()` is called on a child, `damage(FL_DAMAGE_CHILD)` is done to the group, so this bit of `damage()` can be used to indicate that a child needs to be drawn. It is fastest if you avoid drawing anything else in this case:

```
int MyClass::draw() {
  Fl_Widget *const*a = array();
  if (damage() == FL_DAMAGE_CHILD) { // only redraw some children
    for (int i = children(); i --; a ++) update_child(**a);
  } else { // total redraw
    ... draw background graphics ...
    // now draw all the children atop the background:
    for (int i = children_; i --; a ++) {
      draw_child(**a);
      draw_outside_label(**a); // you may not want to do this
    }
  }
}
```

`Fl_Group` provides some protected methods to make drawing easier:

- draw_child
- draw_outside_label
- update_child

void Fl_Group::draw_child(Fl_Widget&)

This will force the child's `damage()` bits all to one and call `draw()` on it, then clear the `damage()`. You should call this on all children if a total redraw of your widget is requested, or if you draw something (like a background box) that damages the child. Nothing is done if the child is not `visible()` or if it is clipped.

void Fl_Group::draw_outside_label(Fl_Widget&) const

Draw the labels that are *not* drawn by `draw_label()`. If you want more control over the label positions you might want to call `child->draw_label(x,y,w,h,a)`.

void Fl_Group::update_child(Fl_Widget&)

Draws the child only if its `damage()` is non-zero. You should call this on all the children if your own damage is equal to FL_DAMAGE_CHILD. Nothing is done if the child is not `visible()` or if it is clipped.

Cut and Paste Support

FLTK provides routines to cut and paste ASCII text (in the future this may be UTF-8) between applications:

- `Fl::paste`
- `Fl::selection`
- `Fl::selection_length`
- `Fl::selection_owner`

It may be possible to cut/paste non-ASCII data by using `Fl::add_handler()`.

Making a subclass of Fl_Window

You may want your widget to be a subclass of `Fl_Window`. This can be useful if your widget wants to occupy an entire window, and can also be used to take advantage of system-provided clipping, or to work with a library that expects a system window ID to indicate where to draw.

Subclassing `Fl_Window` is almost exactly like subclassing `Fl_Widget`, and in fact you can easily switch a subclass back and forth. Watch out for the following differences:

1. `Fl_Window` is a subclass of `Fl_Group` so *make sure your constructor calls `end()`* (unless you actually want children added to your window).
2. When handling events and drawing, the upper-left corner is at 0,0, not `x(),y()` as in other `Fl_Widget`'s. For instance, to draw a box around the widget, call `draw_box(0, 0, w(), h())`, rather than `draw_box(x(), y(), w(), h())`.

You may also want to subclass `Fl_Window` in order to get access to different visuals or to change other attributes of the windows. See "Appendix F - Operating System Issues" for more information.

8 - Programming with FLUID

This chapter shows how to use the Fast Light User-Interface Designer ("FLUID") to create your GUIs.

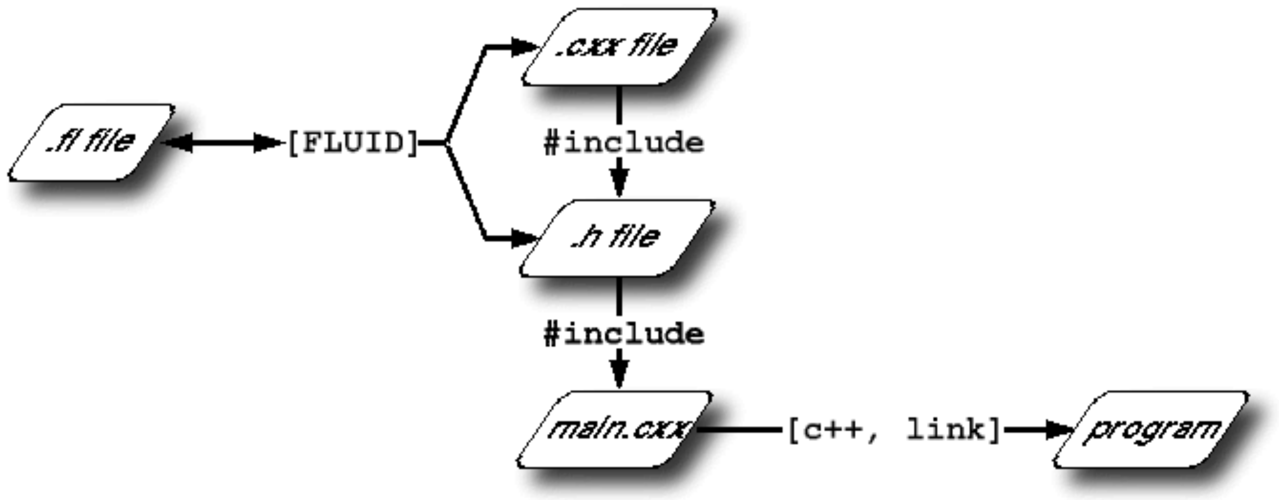
What is FLUID?

The Fast Light User Interface Designer, or FLUID, is a graphical editor that is used to produce FLTK source code.

FLUID edits and saves its state in `.fl` files. These files are text, and you can (with care) edit them in a text editor, perhaps to get some special effects.

FLUID can "compile" the `.fl` file into a `.cxx` and a `.h` file. The `.cxx` file defines all the objects from the `.fl` file and the `.h` file declares all the global ones.

A simple program can be made by putting all your code (including a `main()` function) into the `.fl` file and thus making the `.cxx` file a single source file to compile. Most programs are more complex than this, so you write other `.cxx` files that call the FLUID functions. These `.cxx` files must `#include` the `.h` file or they can `#include` the `.cxx` file so it still appears to be a single source file.



Normally the FLUID file defines one or more functions or classes which output C++ code. Each function defines a one or more FLTK windows, and all the widgets that go inside those windows.

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 assigns a pointer to the widget to the name, but 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.

Widgets may either call a named callback function that you write in another source file, or you can supply a small piece of C++ source and FLUID will write a private callback function into the `.cxx` file.

Running FLUID Under UNIX

To run FLUID under UNIX, type:

```
fluid filename.fl &
```

to edit the `.fl` file `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. You can run FLUID without any name, in which case you will be editing an unnamed blank setup (but you can use save-as to write it to a file).

You can provide any 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
```

Changing the colors may be useful to see what your interface will look at if the user calls it with the same switches.

In the current version, if you don't go into the background (with '&' then you will be able to abort FLUID by typing ^C on the terminal. It will exit immediately, losing any changes.

Running FLUID Under Microsoft Windows

To run FLUID under WIN32, double-click on the *FLUID.exe* file. You can also run FLUID from the Command Prompt window (FLUID always runs in the background under WIN32).

Compiling `.fl` files

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

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

This will read the `filename.fl` file and write *filename.cxx* and *filename.h*. The directory will be stripped, so they are written to the current directory always. If there are any errors reading or writing the files it will print the error and exit with a non-zero code. In a makefile you can use a line like this:

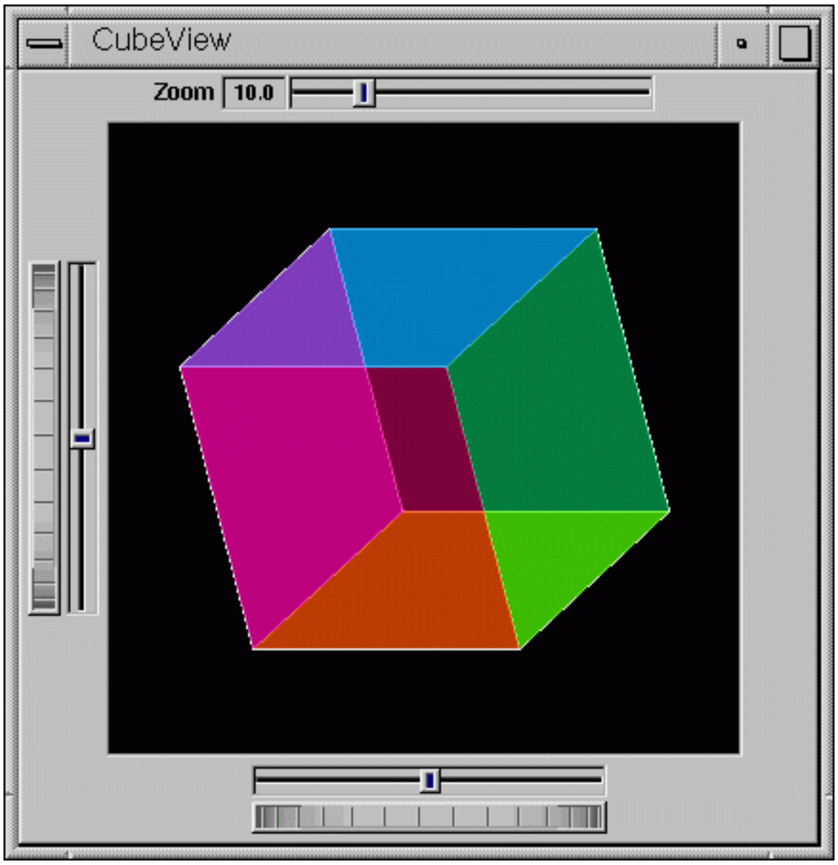
```
my_panels.h my_panels.cxx: my_panels.fl
        FLUID -c my_panels.fl
```

Some versions of make will accept rules like this to allow all `.fl` files found to be compiled:

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

A Short Tutorial

FLUID is an amazingly powerful little program. However, this power comes at a price, as it is not always obvious how to accomplish seemingly simple tasks with it. This tutorial will show you how to generate a complete user interface class with FLUID that is used for the CubeView program below.



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 CubeView.

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 necessary.
2. Use FLUID to set callbacks member functions of a custom widget classes.
3. Subclass an `Fl_Gl_Window` to suit your purposes.

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 the 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":

```
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(float angle){vAng=angle;};
    // Return the rotation about the vertical (y ) axis.
    float v_angle(){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(float angle){hAng=angle;};
    // the rotation about the horizontal (x ) axis.
    float h_angle(){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(float 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(float y){yshift=y;};
    /* The widget class draw() override.
     * The draw() function initialize 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. The color is #defined by
     * CUBECOLOR.
     */
    void drawCube();

    float vAng,hAng; float 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 "cube" demo included with FLTK.

```
#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)
{
    vAng = 0.0; hAng=0.0; size=10.0;
    /* 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;
};

// The color used for the edges of the bounding cube.
#define CUBECOLOR 255,255,255,255

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();
```

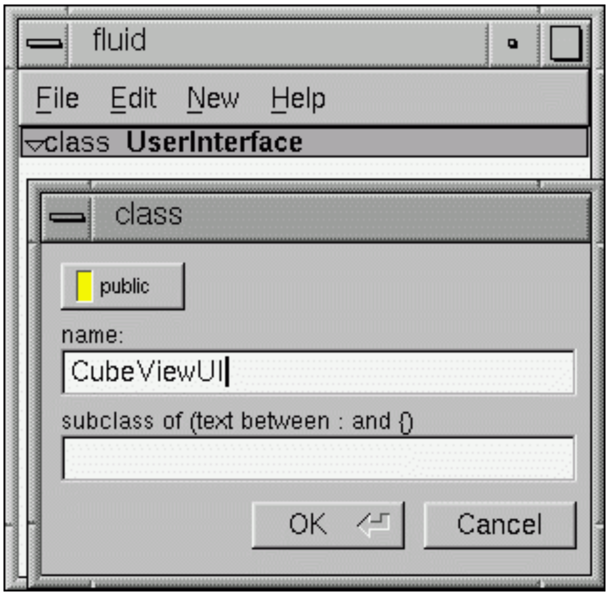
```
    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINES);
      glVertex3fv(boxv0);
      glVertex3fv(boxv1);

      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,w(),h());
        glOrtho(-10,10,-10,10,-20000,10000); glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    }

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix(); glTranslatef(xshift, yshift, 0);
    glRotatef(hAng,0,1,0); glRotatef(vAng,1,0,0);
    glScalef(float(size),float(size),float(size)); drawCube();
    glPopMatrix();
};
```

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.



Adding the Class Constructor

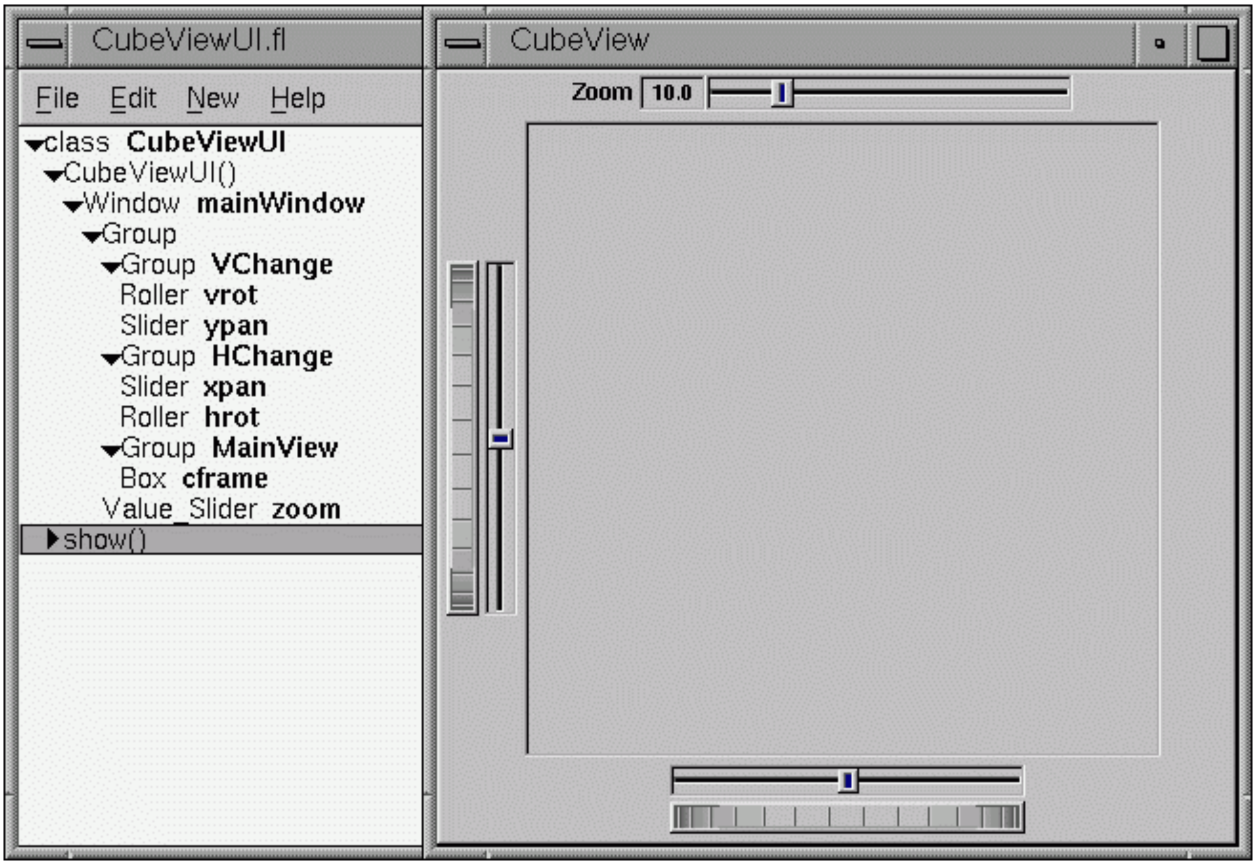
Click on the CubeViewUI class in the FLUID window and add a new method by selecting **New->Code->Function/Method.** The name of the function will also be CubeViewUI. FLUID will understands that this will be the constructor for the class and will generate the appropriate code. Make sure you declare the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



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

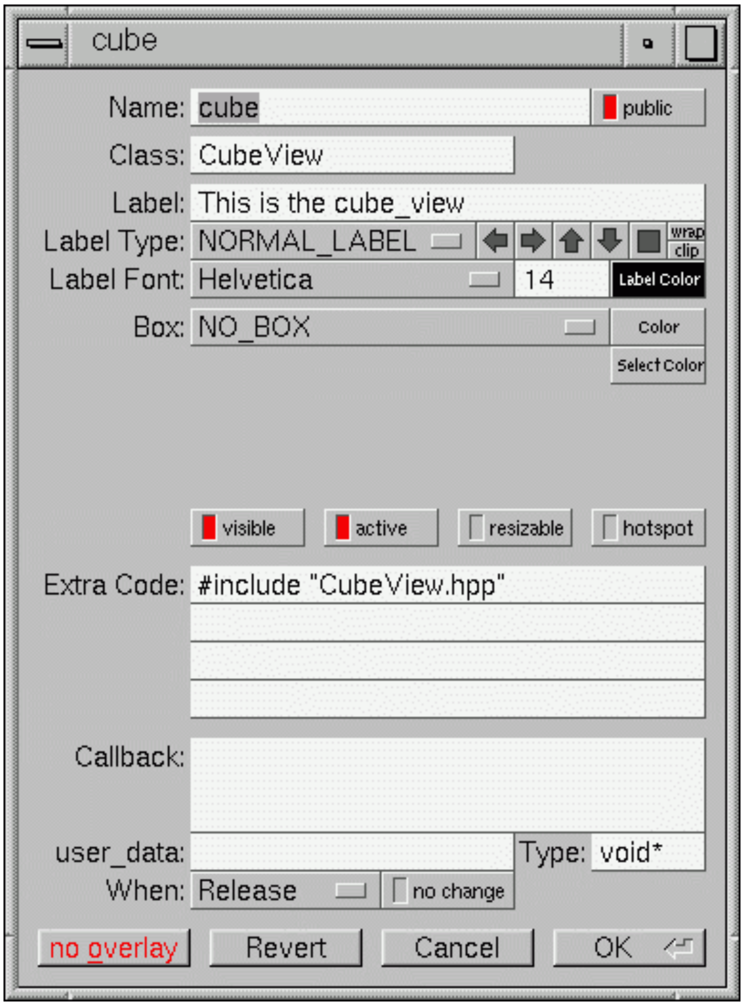
Adding the CubeView Widget

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 a `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.



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 in 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();
```

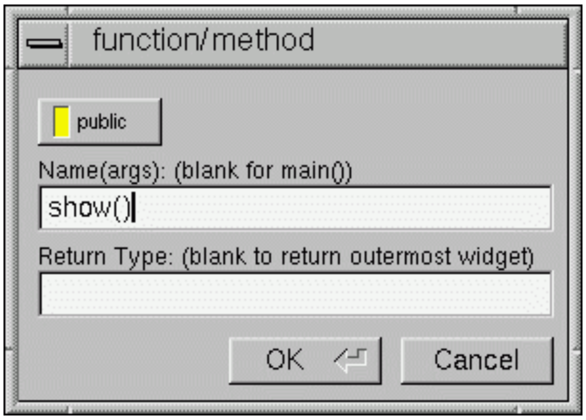
We call `cube->redraw()` after changing the value to update the CubeView window. CubeView could easily be modified to do this, but it is nice to keep this exposed in the case where you may want to do more than one view change only redrawing once saves a lot of time.

There is no reason no 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. An 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`.



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

Adding Constructor Initialization Code

If you need to add code to initialize 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.

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.

FLUID Reference

The Widget Browser

The main window shows a menu bar and a scrolling browser of all the defined widgets. The name of the `.fl` file being edited is shown in the window title.

The 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 "main_panel" in the example), or by their *type* and *label* (such as "Button "the green"").

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).

Menu Items

The menu bar at the top is duplicated as a pop-up menu on any displayed window. The shortcuts for all the menu items work in any window. The menu items are:

File/Open... (Alt+o)

Discards the current editing session and reads in a different `.fl` 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/Save (Alt+s)

Writes the current data to the `.fl` file. If the file is unnamed then FLUID will ask for a filename.

File/Save As...(Alt+Shift+S)

Asks for a new filename and saves the file.

File/Merge... (Alt+i)

Inserts 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/Write Code (Alt+Shift+C)

"Compiles" the data into 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/Quit (Alt+q)

Exits FLUID. You are asked for confirmation if you have changed the current data.

Edit/Undo (Alt+z)

This isn't implemented yet. You should do save often so you can recover from any mistakes you make.

Edit/Cut (Alt+x)

Deletes 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 (Alt+c)

Copies the selected widgets and all of their children to the "clipboard" file.

Edit/Paste (Alt+c)

Pastes 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/Select All (Alt+a)

Selects 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 Alt+a will select larger and larger groups of widgets until everything is selected.

Edit/Open... (F1 or double click)

Displays the current widget in the attributes panel. If the widget is a window and it is not visible then the window is shown instead.

Edit/Sort

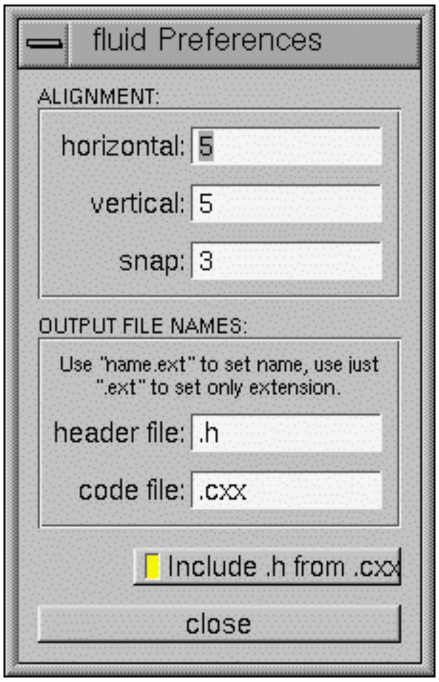
Sorts the selected widgets into left to right, top to bottom order. You need to do this to make navigation keys in FLTK work correctly. You may then fine-tune the sorting with "Earlier" and "Later". This does not affect the positions of windows or functions.

Edit/Earlier (F2)

Moves 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)

Moves all of the selected widgets one later in order among the children of their parent (if possible).



Edit/Group (F7)

Creates a new `Fl_Group` and make all the currently selected widgets children of it.

Edit/Ungroup (F8)

Deletes the parent group if all the children of a group are selected.

Edit/Overlays on/off (Alt+Shift+O)

Toggles 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/Preferences (Alt+p)

Displays the preferences panel. The alignment preferences control the grid 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.

The output filenames control the extensions or names of the files the are generated by FLUID. If you check the "Include .h from .cxx" button the code file will include the header file automatically.

New/Code/Function

Creates 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 a 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.

New/Window

Creates 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 100x100. You can resize it to whatever size you require.

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

New/...

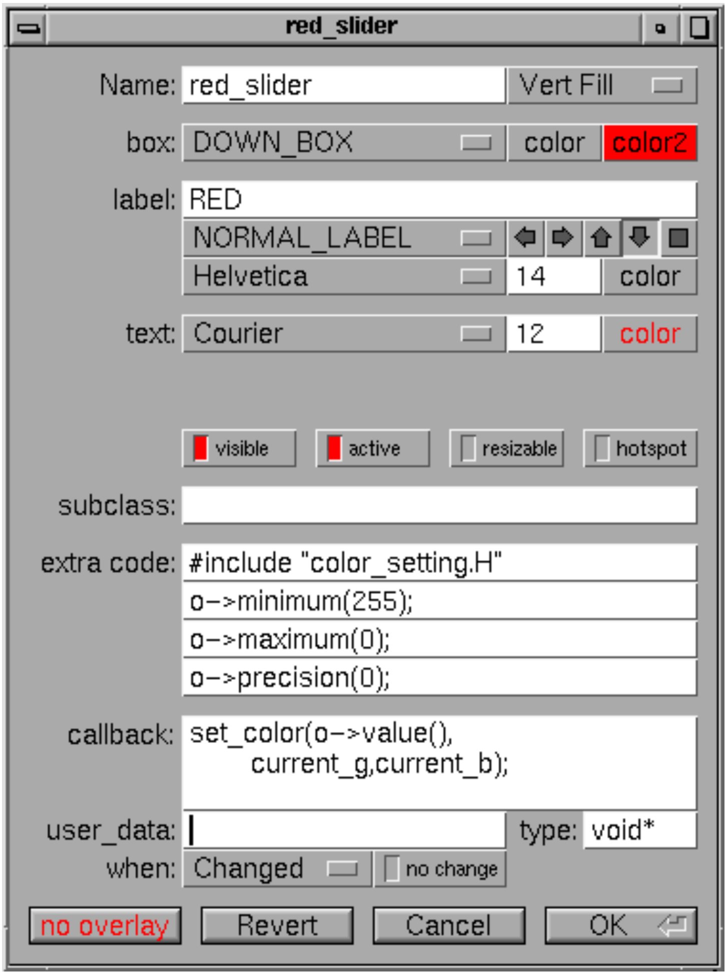
All other items on the New menu are subclasses of `Fl_Widget`. Creating them will add them to the currently selected group or window, or the group or window containing the currently selected widget. The initial dimensions and position are chosen by copying the current widget, if possible.

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

Help/About FLUID

Pops up a panel showing the version of FLUID.

The Widget Panel



When you double-click on a widget or a set of widgets you will get the "widget attribute panel".

When you change attributes using this panel, the changes are reflected immediately in the window. It is useful to hit the "no overlay" button (or type Alt+Shift+O) to hide the red overlay so you can see the widgets more accurately, especially when setting the box type.

If you have several widgets selected, they may have different values for the fields. In this case the value for *one* of the widgets is shown. But if you change this value, *all* of the selected widgets are changed to the new value.

Hitting "OK" makes the changes permanent. Selecting a different widget also makes the changes permanent. FLUID checks for simple syntax errors such as mismatched parenthesis in any code before saving any text.

"Revert" or "Cancel" put everything back to when you last brought up the panel or hit OK. However in the current version of FLUID, changes to "visible" attributes (such as the color, label, box) are not undone by revert or cancel. Changes to code like the callbacks are undone, however.

Widget Attributes

Name (text field)

Name of a variable to declare, and to store a pointer to this widget into. This variable will be of type "<class>*". If the name is blank then no variable is created.

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 that in the array must be the same type.

Type (upper-right pulldown menu)

Some classes have subtypes that modify their appearance or behavior. You pick the subtype off of this menu.

Box (pulldown menu)

The boxtype to draw as a background for the widget.

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.

Color

The color to draw the box with.

Color2

Some widgets will use this 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.

Label

String to print next to or inside the button.

You can put newlines into the string to make multiple lines. The easiest way is by typing Ctrl+j.

Label style (pull down menu)

How to draw the label. Normal, shadowed, engraved, and embossed change the appearance of the text. "symbol" requires the label to start with an '@' sign to draw a named symbol.

From this menu you can also pick "Image...". This lets you use the contents of a GIF, XPM, or XBM image file to label the widget.

Label Alignment (Buttons)

Where to draw the label. The arrows put it on that side of the widget, you can combine the to put it in the corner. The "box" button puts the label inside the widget, rather than outside.

Label Font

Font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. Your program can change the actual font used by these "slots" in case you want some font other than the 16 provided.

Label Size

Pixel size (height) for the font to draw the label in. Ignored by symbols, bitmaps, and pixmaps. To see the result without dismissing the panel, type the new number and then Tab.

Label Color

Color to draw the label. Ignored by pixmaps (bitmaps, however, do use this color as the foreground color).

Text Font, Size, and Color

Some widgets display text, such as input fields, pull-down menus, and browsers.

Visible

If you turn this off then the widget is hidden initially. Don't change this for windows or for the immediate children of a Tabs group.

Active

If you turn this off then the widget is deactivated initially.

Resizable

If a window is resizable or has an immediate child that is resizable, then the user will be able to resize it. 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.

Only one child can be resizable. Turning this on turns it off for the other children.

You can get more complex behavior by making invisible boxes the resizable widget, or by using hierarchies of groups. Unfortunately the only way to test it is to compile the program. Resizing the FLUID window is *not* the same as what will happen in the user program.

Hotspot

Each window may have exactly one hotspot (turning this on will turn off any others). This will cause it 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()`.

Subclass

This is how you use your own subclasses of `Fl_Widget`. Whatever identifier you type in here will be the class that is instantiated.

In addition, no `#include` header file is put in the `.h` file. You must provide a `#include` line as the first line of the "Extra Code" which declares your subclass.

The class must be similar to the class you are spoofing. It does not have to be a subclass. It is sometimes useful to change this to another FLTK class. Currently the only way to get a double-buffered window is to change this field for the window to "Fl_Double_Window" and to add "#include <FL/Fl_Double_Window.h>" to the extra code.

Extra Code

These four fields let you type in literal lines of code to dump into the `.h` or `.cxx` files.

If the text starts with a `#` or the word `extern` then FLUID thinks this is an "include" line, and it is written to the `.h` file. If the same include line occurs several times then only one copy is written.

All other lines are "code" lines. The current widget is pointed to by the local variable `o`. The window being constructed is pointed to by the local variable `w`. You can also access any arguments passed to the function here, and any named widgets that are before this one.

FLUID will check for matching parenthesis, 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.

Callback

This can either be the name of a function, or a small snippet of code. If you enter anything but letters, numbers, and the underscore then FLUID treats it as code.

A name names a function in your own code. It must be declared as `void name(<class>*,void*)`.

A code snippet is inserted into a static function in the `.cxx` output file. The function prototype is `void name(class *o, void *v)` so that you can refer to the widget as `o` and the `user_data()` as `v`. FLUID will check for matching parenthesis, 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 the callback is blank then no callback is set.

user_data

This is a value for the `user_data()` of the widget. If blank the default value of zero is used. This can be any piece of C code that can be cast to a `void` pointer.

User Data Type

The `void *` in the callback function prototypes is replaced with this. You may want to use `long` for old XForms code. Be warned that anything other than `void *` is not guaranteed to work! However on most architectures other pointer types are ok, and `long` is usually ok, too.

When

When to do the callback. This can be "never", "changed", "release", "enter key", or "no change". The value of "enter key" is only useful for text input fields. The "no change" button means the callback is done on the matching event even if the data is not changed.

There are other rare but useful values for the `when()` field that are not in the menu. You should use the extra code fields to put these values in.

Selecting and Moving Widgets

Double-clicking a window name in the browser will display it, if not displayed yet. From this display you can select widgets, sets of widgets, and move or resize them. To close a window either double-click it or type Esc.

To select a widget, click it. To select several widgets drag a rectangle around them. Holding down shift will toggle the selection of the widgets instead.

You cannot pick hidden widgets. You also cannot choose some widgets if they are completely overlapped by later widgets. Use the browser to select these widgets.

The selected widgets are shown with a red "overlay" line around them. You can move the widgets by dragging this box. Or you can resize them by dragging the outer edges and corners. Hold down the Alt key while dragging the mouse to defeat the snap-to-grid effect for fine positioning.

If there is a tab box displayed you can change which child is visible by clicking on the file tabs. The child you pick is selected.

The arrow, tab, and shift+tab keys "navigate" the selection. Left, right, tab, or shift+tab move to the next or previous widgets in the hierarchy. Hit the right arrow enough and you will select every widget in the window. Up/down widgets move to the previous/next widgets that overlap horizontally. 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 arrow keys to move between input fields.

To "open" a widget, double click it. To open several widgets select them and then type F1 or pick "Edit/Open" off the pop-up menu.

Type Alt+o to temporarily toggle the overlay off without changing the selection, so you can see the widget borders.

You can resize the window by using the window manager border controls. FLTK will attempt to round the window size to the nearest multiple of the grid size and makes it big enough to contain all the widgets (it does this using illegal X methods, so it is possible it will barf with some window managers!). Notice that the actual window in your program may not be resizable, and if it is, the effect on child widgets may be different.

The panel for the window (which you get by double-clicking it) is almost identical to the panel for any other Fl_Widget. There are three extra items:

Border

This button turns the window manager border on or off. On most window managers you will have to close the window and reopen it to see the effect.

xclass

The string typed into here is passed to the X window manager as the class. This can change the icon or window decorations. On most (all?) window managers you will have to close the window and reopen it to see the effect.

Image Labels

Selecting "Image..." off the label style pull-down menu will bring up a file chooser from which you pick the image file. If an image has already been chosen, you can change the image used by picking "Image..." again. The name of the image will appear in the "label" field, but you can't edit it.

The *contents* of the image file are written to the `.cxx` file, so if you wish to distribute the C code, you only need to copy the `.cxx` file, not the images. If many widgets share the same image then only one copy is written.

However the *file name* is stored in the `.fl` file, so to read the `.fl` file you need the image files as well. Filenames are relative to the location the `.fl` file is (not necessarily the current directory). I recommend you either put the images in the same directory as the `.fl` file, or use absolute path names.

Notes for all image types

FLUID runs using the default visual of your X server. This may be 8 bits, which will give you dithered images. You may get better results in your actual program by adding the code "Fl::visual(FL_RGB)" to your code right before the first window is displayed.

All widgets with the same image on them share the same code and source X pixmap. Thus once you have put an image on a widget, it is nearly free to put the same image on many other widgets.

If you are using a painting program to edit an image: the only way to convince FLUID to read the image file again is to remove the image from all widgets that are using it (including ones in closed windows), which will cause it to free its internal copy, and then set the image again. You may find it easier to exit FLUID and run it again.

Don't rely on how FLTK crops images that are outside the widget, as this may change in future versions! The cropping of inside labels will probably be unchanged.

To more accurately place images, make a new "box" widget and put the image in that as the label. This is also how you can put both an image and text label on the same widget. If your widget is a button, and you want the image inside it, you must change the button's boxtype to FL_UP_FRAME (or another frame), otherwise when it is pushed it will erase the image.

XBM (X bitmap files)

FLUID will read X bitmap files. These files have C source code to define a bitmap. Sometimes they are stored with the ".h" or ".bm" extension rather than the standard ".xbm".

FLUID will output code to construct an Fl_Bitmap widget and use it to label the widget. The '1' bits in the bitmap are drawn using the label color of the widget. You can change the color in FLUID. The '0' bits are transparent.

The program "bitmap" on the X distribution does an ok job of editing bitmaps.

XPM (X pixmap files)

FLUID will read X pixmap files as used by the libxpm library. These files have C source code to define a pixmap. The filenames usually have a ".xpm" extension.

FLUID will output code to construct an Fl_Pixmap widget and use it to label the widget. The label color of the widget is ignored, even for 2-color images that could be a bitmap.

XPM files can mark a single color as being transparent. Currently FLTK and FLUID simulate this transparency rather badly. It will use the color() of the widget as the background, and all widgets using the same pixmap are assummed to have the same color. This may be fixed in the future or on non-X systems.

I have not found any good editors for small iconic pictures. For pixmaps I have used XPaint. This (and most other) painting programs are designed for large full color images and are difficult to use to edit an image of small size and few colors.

GIF files

FLUID will also read GIF image files. These files are often used on html documents to make icons. This lets you use nice icons that you steal off the net in your user interface.

FLUID converts these into (modified) XPM format and uses an Fl_Pixmap widget to label the widget. Transparency is handled the same as for xpm files. Notice that the conversion removes the compression, so the code may be much bigger than the .gif file. Only the first image of an animated gif file is used.

Behavior and performance with large .gif files is not guaranteed!

9 - Using OpenGL

This chapter discusses using FLTK for your OpenGL applications.

Using OpenGL in FLTK

The easiest way to make an OpenGL display is to subclass `Fl_Gl_Window`. Your subclass must implement a `draw()` method which uses OpenGL calls to draw the display. Your main program should call `redraw()` when the display needs to change, and (somewhat later) FLTK will call `draw()`.

With a bit of care you can also use OpenGL to draw into normal FLTK windows. This allows you to use Gouraud shading for drawing your widgets. To do this you use the `gl_start()` and `gl_finish()` functions around your OpenGL code.

You must include FLTK's `<FL/gl.h>` header file. It will include the file `<GL/gl.h>`, define some extra drawing functions provided by FLTK, and include the `<windows.h>` header file needed by WIN32 applications.

Making a Subclass of Fl_Gl_Window

To make a subclass of Fl_Gl_Window, you must provide:

- A class definition.
- A `draw()` method.
- A `handle()` method (if you need to recieve input from the user).

If your subclass provides static controls in the window, they must be redrawn whenever the `FL_DAMAGE_ALL` bit is set in the value returned by `damage()`. For double-buffered windows you will need to surround the drawing code with the following code to make sure that both buffers are redrawn:

```
#ifndef MESA
glDrawBuffer(GL_FRONT_AND_BACK);
#endif // !MESA
... draw stuff here ...
#ifndef MESA
glDrawBuffer(GL_BACK);
#endif // !MESA
```

Note: If you are using the Mesa graphics library, the call to `glDrawBuffer()` is not required and will slow down drawing considerably. The preprocessor instructions shown above will optimize your code based upon the graphics library used.

Defining the Subclass

To define the subclass you just subclass the `Fl_Gl_Window` class:

```
class MyWindow : public Fl_Gl_Window {
  void draw();
  int handle(int);

public:
  MyWindow(int X, int Y, int W, int H, const char *L)
    : Fl_Gl_Window(X, Y, W, H, L) {}
};
```

The `draw()` and `handle()` methods are described below. Like any widget, you can include additional private and public data in your class (such as scene graph information, etc.)

The draw() Method

The `draw()` method is where you actually do your OpenGL drawing:

```
void MyWindow::draw() {
  if (!valid()) {
    ... set up projection, viewport, etc ...
    ... window size is in w() and h().
    ... valid() is turned on by FLTK after draw() returns
  }
  ... draw ...
}
```

The handle() Method

The `handle()` method handles mouse and keyboard events for the window:

```
int MyWindow::handle(int event) {
  switch(event) {
  case FL_PUSH:
    ... mouse down event ...
    ... position in Fl::event_x() and Fl::event_y()
```

```
    return 1;
  case FL_DRAG:
    ... mouse moved while down event ...
    return 1;
  case FL_RELEASE:
    ... mouse up event ...
    return 1;
  case FL_FOCUS :
  case FL_UNFOCUS :
    ... Return 1 if you want keyboard events, 0 otherwise
    return 1;
  case FL_KEYBOARD:
    ... keypress, key is in Fl::event_key(), ascii in Fl::event_text()
    ... Return 1 if you understand/use the keyboard event, 0 otherwise...
    return 1;
  case FL_SHORTCUT:
    ... shortcut, key is in Fl::event_key(), ascii in Fl::event_text()
    ... Return 1 if you understand/use the shortcut event, 0 otherwise...
    return 1;
  default:
    // tell FLTK that I don't understand other events
    return 0;
  }
}
```

When `handle()` is called, the OpenGL context is not set up! If your display changes, you should call `redraw()` and let `draw()` do the work. Don't call any OpenGL drawing functions from inside `handle()`!

You can call *some* OpenGL stuff like hit detection and texture loading functions by doing:

```
case FL_PUSH:
  make_current(); // make OpenGL context current
  if (!valid()) {
    ... set up projection exactly the same as draw ...
    valid(1); // stop it from doing this next time
  }
  ... ok to call NON-DRAWING OpenGL code here, such as hit
  detection, loading textures, etc...
```

Your main program can now create one of your windows by doing `new MyWindow(...)`. You can also use FLUID by:

1. Putting your class definition in a `MyWindow.H` file.
2. Creating a `Fl_Box` widget in FLUID.
3. In the widget panel fill in the "class" field with `MyWindow`. This will make FLUID produce constructors for your new class.
4. In the "Extra Code" field put `#include "MyWindow.H"`, so that the FLUID output file will compile.

You must put `glwindow->show()` in your main code after calling `show()` on the window containing the OpenGL window.

Using OpenGL in Normal FLTK Windows

You can put OpenGL code into an `Fl_Widget::draw()` method or into the code for a boxtype or other places with some care.

Most importantly, before you show *any* windows (including those that don't have OpenGL drawing) you **must** initialize FLTK so that it knows it is going to use OpenGL. You may use any of the symbols described for `Fl_Gl_Window::mode()` to describe how you intend to use OpenGL:

```
Fl::gl_visual(FL_RGB);
```

You can then put OpenGL drawing code anywhere you can draw normally by surrounding it with:

```
gl_start();
... put your OpenGL code here ...
gl_finish();
```

`gl_start()` and `gl_finish()` set up an OpenGL context with an orthographic projection so that 0,0 is the lower-left corner of the window and each pixel is one unit. The current clipping is reproduced with OpenGL `glScissor()` commands. These also synchronize the OpenGL graphics stream with the drawing done by other X, WIN32, or FLTK functions.

The same context is reused each time. If your code changes the projection transformation or anything else you should use `glPushMatrix()` and `glPopMatrix()` functions to put the state back before calling `gl_finish()`.

You may want to use `Fl_Window::current()->h()` to get the drawable height so that you can flip the Y coordinates.

Unfortunately, there are a bunch of limitations you must adhere to for maximum portability:

- You must choose a default visual with `Fl::gl_visual()`.
- You cannot pass `FL_DOUBLE` to `Fl::gl_visual()`.
- You cannot use `Fl_Double_Window` or `Fl_Overlay_Window`.

Do *not* call `gl_start()` or `gl_finish()` when drawing into an `Fl_Gl_Window`!

OpenGL Drawing Functions

FLTK provides some useful OpenGL drawing functions. They can be freely mixed with any OpenGL calls, and are defined by including `<FL/gl.H>` (which you should include instead of the OpenGL header `<GL/gl.h>`).

void gl_color(Fl_Color)

Set the current color to a FLTK color. *For color-index modes it will use `fl_xpixel(c)`, which is only right if this window uses the default colormap!*

void gl_rect(int x, int y, int w, int h)
void gl_rectf(int x, int y, int w, int h)

Outline or fill a rectangle with the current color. If `Fl_Gl_Window::ortho()` has been called, then the rectangle will exactly fill the pixel rectangle passed.

void gl_font(Fl_Font fontid, int size)

Set the current OpenGL font to the same font you get by calling `fl_font()`.

int gl_height()
int gl_descent()
float gl_width(const char *)
float gl_width(const char *, int n)
float gl_width(uchar)

Return information about the current OpenGL font.

void gl_draw(const char *)
void gl_draw(const char *, int n)

Draw a nul-terminated string or an array of `n` characters in the current OpenGL font at the current raster position.

void gl_draw(const char *, int x, int y)
void gl_draw(const char *, int n, int x, int y)
void gl_draw(const char *, float x, float y)
void gl_draw(const char *, int n, float x, float y)

Draw a nul-terminated string or an array of `n` characters in the current OpenGL font at the given position.

void gl_draw(const char *, int x, int y, int w, int h, Fl_Align)

Draw a string formatted into a box, with newlines and tabs expanded, other control characters changed to ^X, and aligned with the edges or center. Exactly the same output as `fl_draw()` .

Using OpenGL Optimizer with FLTK

OpenGL Optimizer is a scene graph toolkit for OpenGL available from Silicon Graphics for IRIX and Microsoft Windows. Versions are in the works for Solaris and HP-UX. It allows you to view large scenes without writing a lot of OpenGL code.

OptimizerWindow Class Definition

To use OpenGL Optimizer with FLTK you'll need to create a subclass of `Fl_Gl_Widget` that includes several state variables:

```
class OptimizerWindow : public Fl_Gl_Window {
  csContext *context_; // Initialized to 0 and set by draw()...
  csDrawAction *draw_action_; // Draw action...
```

```
  csGroup *scene_; // Scene to draw...
  csCamara *camera_; // Viewport for scene...

  void draw();

public:
  OptimizerWindow(int X, int Y, int W, int H, const char *L)
    : Fl_Gl_Window(X, Y, W, H, L) {
      context_ = (csContext *)0;
      draw_action_ = (csDrawAction *)0;
      scene_ = (csGroup *)0;
      camera_ = (csCamera *)0;
    }

  void scene(csGroup *g) { scene_ = g; redraw(); }

  void camera(csCamera *c) {
    camera_ = c;
    if (context_) {
      draw_action_->setCamera(camera_);
      camera_->draw(draw_action_);
      redraw();
    }
  }
};
```

The camera() Method

The `camera()` method sets the camera (projection and viewpoint) to use when drawing the scene. The scene is redrawn after this call.

The draw() Method

The `draw()` method performs the needed initialization and does the actual drawing:

```
void OptimizerWindow::draw() {
  if (!context_) {
    // This is the first time we've been asked to draw; create the
    // Optimizer context for the scene...

#ifdef WIN32
    context_ = new csContext((HDC)fl_getHDC());
    context_->ref();
    context_->makeCurrent((HDC)fl_getHDC());
#else
    context_ = new csContext(fl_display, fl_visual);
    context_->ref();
    context_->makeCurrent(fl_display, fl_window);
#endif // WIN32

    ... perform other context setup as desired ...

    // Then create the draw action to handle drawing things...

    draw_action_ = new csDrawAction;
    if (camera_) {
      draw_action_->setCamera(camera_);
```

```
      camera_->draw(draw_action_);
    }
  } else {
#ifdef WIN32
    context_->makeCurrent((HDC)fl_getHDC());
#else
    context_->makeCurrent(fl_display, fl_window);
#endif // WIN32
  }

  if (!valid()) {
    // Update the viewport for this context...
    context_->setViewport(0, 0, w(), h());
  }

  // Clear the window...

  context_->clear(csContext::COLOR_CLEAR | csContext::DEPTH_CLEAR,
                  0.0f,         // Red
                  0.0f,         // Green
                  0.0f,         // Blue
                  1.0f);        // Alpha

  // Then draw the scene (if any)...

  if (scene_)
    draw_action_->apply(scene_);
}
```

The scene() Method

The `scene()` method sets the scene to be drawn. The scene is a collection of 3D objects in a `csGroup`. The scene is redrawn after this call.

A - Widget Reference

This appendix describes all of the widget classes in FLTK. For a description of the `fl_` functions and `Fl::` methods, see Appendix B.

Alphabetical List of Classes

- Fl_Adjuster
- Fl_Box
- Fl_Browser
- Fl_Browser_
- Fl_Button
- Fl_Chart
- Fl_Check_Button
- Fl_Choice
- Fl_Clock
- Fl_Color_Chooser
- Fl_Counter
- Fl_Dial
- Fl_Double_Window
- Fl_End
- Fl_Float_Input
- Fl_Free
- Fl_Gl_Window
- Fl_Group
- Fl_Hold_Browser
- Fl_Input
- Fl_Input_
- Fl_Int_Input
- Fl_Light_Button
- Fl_Menu_
- Fl_Menu_Bar
- Fl_Menu_Button
- Fl_Menu_Item
- Fl_Menu_Window
- Fl_Multi_Browser
- Fl_Multiline_Input
- Fl_Multiline_Output
- Fl_Output
- Fl_Overlay_Window
- Fl_Pack
- Fl_Positioner
- Fl_Repeat_Button
- Fl_Return_Button
- Fl_Roller
- Fl_Round_Button
- Fl_Scroll
- Fl_Scrollbar
- Fl_Secret_Input
- Fl_Select_Browser
- Fl_Single_Window
- Fl_Slider
- Fl_Tabs
- Fl_Tile
- Fl_Timer
- Fl_Valuator
- Fl_Value_Input
- Fl_Value_Output
- Fl_Value_Slider
- Fl_Widget
- Fl_Window

Class Hierarchy

- Fl_End
- Fl_Menu_Item
- Fl_Widget
    - Fl_Box
    - Fl_Browser_
        - Fl_Browser
            - Fl_Hold_Browser
            - Fl_Multi_Browser
            - Fl_Select_Browser
    - Fl_Button
        - Fl_Check_Button
        - Fl_Light_Button
        - Fl_Repeat_Button
        - Fl_Return_Button
        - Fl_Round_Button
    - Fl_Chart
    - Fl_Clock
    - Fl_Free
    - Fl_Group
        - Fl_Color_Chooser
        - Fl_Pack
        - Fl_Scroll
        - Fl_Tabs
        - Fl_Tile
        - Fl_Window
            - Fl_Double_Window
            - Fl_Gl_Window
            - Fl_Menu_Window
            - Fl_Overlay_Window
            - Fl_Single_Window
    - Fl_Input_
        - Fl_Input
            - Fl_Float_Input
            - Fl_Int_Input
            - Fl_Multiline_Input
            - Fl_Secret_Input
        - Fl_Output
            - Fl_Multiline_Output
    - Fl_Menu_
        - Fl_Choice
        - Fl_Menu_Bar
        - Fl_Menu_Button
    - Fl_Positioner
    - Fl_Timer
    - Fl_Valuator
        - Fl_Adjuster
        - Fl_Counter
        - Fl_Dial

- ◊ Fl_Roller
- ◊ Fl_Slider
    - · Fl_Scrollbar
    - · Fl_Value_Slider
- ◊ Fl_Value_Input
- ◊ Fl_Value_Output

class Fl_Adjuster

Class Hierarchy

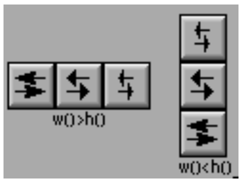
```
Fl_Valuator
   |
   +----Fl_Adjuster
```

Include Files

```
#include <FL/Fl_Adjuster.H>
```

Description

The `Fl_Adjuster` widget was stolen from Prisms, and has proven to be very useful for values that need a large dynamic range.



When you press a button and drag to the right the value increases. When you drag to the left it decreases. The largest button adjusts by `100 * step()`, the next by `10 * step()` and that smallest button by `step()`. Clicking on the buttons increments by 10 times the amount dragging by a pixel does. Shift + click decrements by 10 times the amount.

Methods

- Fl_Adjuster
- ~Fl_Adjuster
- soft

Fl_Adjuster::Fl_Adjuster(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Adjuster` widget using the given position, size, and label string. It looks best if one of the dimensions is 3 times the other.

virtual Fl_Adjuster::~Fl_Adjuster()

Destroys the valuator.

uchar Fl_Adjuster::soft() const
void Fl_Adjuster::soft(uchar)

If "soft" is turned on, the user is allowed to drag the value outside the range. If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

class Fl_Box

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Box
```

Include Files

```
#include <FL/Fl_Box.H>
```

Description

This widget simply draws its box, and possibly it's label. Putting it before some other widgets and making it big enough to surround them will let you draw a frame around them.

Methods

- Fl_Box
- ~Fl_Box

Fl_Box::Fl_Box(int x, int y, int w, int h, const char * = 0)
Fl_Box::Fl_Box(Fl_Boxtype b, int x, int y, int w, int h, const char *)

The first constructor sets `box()` to `FL_NO_BOX`, which means it is invisible. However such widgets are useful as placeholders or `Fl_Group::resizable()` values. To change the box to something visible, use `box(n)`.

The second form of the constructor sets the box to the specified box type.

Fl_Box::~Fl_Box(void)

The destructor removes the box.

class Fl_Browser

Class Hierarchy

```
Fl_Browser_
   |
   +----Fl_Browser
           |
           +----Fl_Hold_Browser, Fl_Multi_Browser, Fl_Select_Browser
```

Include Files

```
#include <FL/Fl_Browser.H>
```

Description

The `Fl_Browser` widget displays a scrolling list of text lines, and manages all the storage for the text. This is not a text editor or spreadsheet! But it is useful for showing a vertical list of named objects to the user.

Each line in the browser is identified by number. *The numbers start at one* (this is so that zero can be reserved for "no line" in the selective browsers). *Unless otherwise noted, the methods do not check to see if the passed line number is in range and legal. It must always be greater than zero and `<= size()`.*

Each line contains a null-terminated string of text and a `void *` data pointer. The text string is displayed, the `void *` pointer can be used by the callbacks to reference the object the text describes.

The base class does nothing when the user clicks on it. The subclasses `Fl_Select_Browser`, `Fl_Hold_Browser`, and `Fl_Multi_Browser` react to user clicks to select lines in the browser and do callbacks.

The base class called `Fl_Browser_` provides the scrolling and selection mechanisms of this and all the subclasses, but the dimensions and appearance of each item are determined by the subclass. You can use `Fl_Browser_` to display information other than text, or text that is dynamically produced from your own data structures. If you find that loading the browser is a lot of work or is inefficient, you may want to make a subclass of `Fl_Browser_`.

Methods

- Fl_Browser
- ~Fl_Browser
- add
- bottomline
- clear
- column_char
- column_widths
- data
- format_char
- hide
- insert
- load
- middleline
- move
- position
- remove
- show
- size
- text
- topline
- visible

Fl_Browser::Fl_Browser(int, int, int, int, const char * = 0)

The constructor makes an empty browser.

Fl_Browser::~Fl_Browser(void)

The destructor deletes all list items and destroys the browser.

void Fl_Browser::add(const char *, void * = 0)

Add a new line to the end of the browser. The text is copied using the `strdup()` function. It may also be `NULL` to make a blank line. The `void *` argument is returned as the `data()` of the new item.

void Fl_Browser::bottomline(int n)

Scrolls the browser so the bottom line in the browser is `n`.

void Fl_Browser::clear()

Remove all the lines in the browser.

uchar Fl_Browser::column_char() const
void Fl_Browser::column_char(char c)

The first form gets the current column separator character. By default this is `'\t'` (tab).

The second form sets the column separator to `c`. This will only have an effect if you also set `column_widths()`.

const int *Fl_Browser::column_widths() const
void Fl_Browser::column_widths(const int *w)

The first form gets the current column width array. This array is zero-terminated and specifies the widths in pixels of each column. The text is split at each `column_char()` and each part is formatted into it's own column. After the last column any remaining text is formatted into the space between the last column and the right edge of the browser, even if the text contains instances of `column_char()` . The default value is a one-element array of just a zero, which makes there are no columns.

The second form sets the current array to `w`. Make sure the last entry is zero.

void *Fl_Browser::data(int n) const
void Fl_Browser::data(int n, void *)

The first form returns the data for line `n`. If `n` is out of range this returns `NULL`.

The second form sets the data for line `n`.

uchar Fl_Browser::format_char() const
void Fl_Browser::format_char(char c)

The first form gets the current format code prefix character, which by default is `@`. A string of formatting codes at the start of each column are stripped off and used to modify how the rest of the line is printed:

- `@.` Print rest of line, don't look for more '@' signs
- `@@` Print rest of line starting with '@'
- `@l` Use a large (24 point) font
- `@m` Use a medium large (18 point) font
- `@s` Use a small (11 point) font
- `@b` Use a **bold** font (adds FL_BOLD to font)
- `@i` Use an *italic* font (adds FL_ITALIC to font)
- `@f` or `@t` Use a `fixed-pitch` font (sets font to FL_COURIER)
- `@c` Center the line horizontally
- `@r` Right-justify the text
- `@B0, @B1, ... @B255` Fill the backgound with fl_color(n)
- `@C0, @C1, ... @C255` Use fl_color(n) to draw the text
- `@F0, @F1, ...` Use fl_font(n) to draw the text
- `@S1, @S2, ...` Use point size n to draw the text
- `@u` or `@_` Underline the text.
- `@-` draw an engraved line through the middle.

Notice that the `@.` command can be used to reliably terminate the parsing. To print a random string in a random color, use `sprintf("@C%d@.%s", color, string)` and it will work even if the string starts with a digit or has the format character in it.

The second form sets the current prefix to `c`. Set the prefix to 0 to disable formatting.

void Fl_Browser::hide(int n)

Makes line `n` invisible, preventing selection by the user. The line can still be selected under program control.

void Fl_Browser::insert(int n, const char *, void * = 0)

Insert a new line *before* line `n`. If `n > size()` then the line is added to the end.

int Fl_Browser::load(const char *filename)

Clears the browser and reads the file, adding each line from the file to the browser. If the filename is `NULL` or a zero-length string then this just clears the browser. This returns zero if there was any error in opening or reading the file, in which case `errno` is set to the system error. The `data()` of each line is set to `NULL`.

void Fl_Browser::middleline(int n)

Scrolls the browser so the middle line in the browser is `n`.

void Fl_Browser::move(int to, int from)

Line `from` is removed and reinserted at `to`; `to` is calculated after the line is removed.

int Fl_Browser::position() const
void Fl_Browser::position(int p)

The first form returns the current vertical scrollbar position, where 0 corresponds to the top. If there is not vertical scrollbar then this will always return 0.

The second form sets the vertical scrollbar position to `p`.

void Fl_Browser::remove(int n)

Remove line `n` and make the browser one line shorter.

void Fl_Browser::show(int n)

Makes line `n` visible for selection.

int Fl_Browser::size() const

Returns how many lines are in the browser. The last line number is equal to this.

const char *Fl_Browser::text(int n) const
void Fl_Browser::text(int n, const char *)

The first form returns the text for line `n`. If `n` is out of range it returns `NULL`.

The second form sets the text for line `n`.

int Fl_Browser::topline() const
void Fl_Browser::topline(int n)

The first form returns the current top line in the browser. If there is no vertical scrollbar then this will always return 1.

The second form scrolls the browser so the top line in the browser is `n`.

int Fl_Browser::visible(int n) const

Returns a non-zero value if line `n` is visible.

class Fl_Browser_

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Browser_
           |
           +----Fl_Browser
```

Include Files

```
#include <FL/Fl_Browser_.H>
```

Description

This is the base class for browsers. To be useful it must be subclassed and several virtual functions defined. The Forms-compatable browser and the file chooser's browser are subclassed off of this.

This has been designed so that the subclass has complete control over the storage of the data, although because `next()` and `prev()` functions are used to index, it works best as a linked list or as a large block of characters in which the line breaks must be searched for.

A great deal of work has been done so that the "height" of a data object does not need to be determined until it is drawn. This is useful if actually figuring out the size of an object requires accessing image data or doing `stat()` on a file or doing some other slow operation.

Methods

- Fl_Browser_
- ~Fl_Browser_
- bbox
- deleting
- deselect
- display
- displayed
- draw
- find_item
- full_height
- full_width
- handle
- has_scrollbar
- hposition
- incr_height
- inserting
- item_draw
- item_first
- item_height
- item_next
- item_prev
- item_quick_height
- item_select
- item_selected
- item_width
- leftedge
- new_list
- position
- redraw_line
- redraw_lines
- replacing
- resize
- scrollbar_left
- scrollbar_right
- select
- select_only
- selection
- textcolor
- textfont
- textsize
- top

Sorry, documentation is incomplete at this time

Fl_Browser::Fl_Browser(int, int, int, int, const char * = 0)

The constructor makes an empty browser.

Fl_Browser::~Fl_Browser(void)

The destructor deletes all list items and destroys the browser.

void Fl_Browser_::has_scrollbar(int h)

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the widget. has_scrollbar() changes this based on the value of `h`:

- `0` - No scrollbars
- `Fl_Browser_::HORIZONTAL` - Only a horizontal scrollbar.
- `Fl_Browser_::VERTICAL` - Only a vertical scrollbar.
- `Fl_Browser_::BOTH` - The default is both scrollbars.
- `Fl_Browser_::HORIZONTAL_ALWAYS` - Horizontal scrollbar always on, vertical always off.
- `Fl_Browser_::VERTICAL_ALWAYS` - Vertical scrollbar always on, horizontal always off.
- `Fl_Browser_::BOTH_ALWAYS` - Both always on.

Fl_Color Fl_Browser_::textcolor() const
void Fl_Browser_::textcolor(Fl_Color color)

The first form gets the default text color for the lines in the browser.

The second form sets the default text color to `color`

Fl_Font Fl_Browser_::textfont() const
void Fl_Browser_::textfont(Fl_Font font)

The first form gets the default text font for the lines in the browser.

The second form sets the default text font to `font`

uchar Fl_Browser_::textsize() const
void Fl_Browser_::textsize(uchar size)

The first form gets the default text size for the lines in the browser.

The second form sets the default text size to `size`

class Fl_Button

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Button
           |
           +----Fl_Check_Button, Fl_Light_Button, Fl_Repeat_Button,
                Fl_Return_Button, Fl_Round_Button
```

Include Files

```
#include <FL/Fl_Button.H>
```

Description

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for `type()` and `when()`.

Buttons can also generate callbacks in response to `FL_SHORTCUT` events. The button can either have an explicit `shortcut()` value or a letter shortcut can be indicated in the `label()` with an '&' character before it. For the label shortcut it does not matter if *Alt* is held down, but if you have an input field in the same window, the user will have to hold down the *Alt* key so that the input field does not eat the event first as an `FL_KEYBOARD` event.

Methods

• Fl_Button	• clear	• set	• shortcut	• value
• ~Fl_Button	• down_box	• setonly	• type	• when

Fl_Button::Fl_Button(int x, int y, int w, int h, const char *label = 0)

The constructor creates the button using the position, size, and label.

Fl_Button::~Fl_Button(void)

The destructor removed the button.

int Fl_Button::clear()

Same as `value(0)`.

Fl_Boxtype Fl_Button::down_box() const
void Fl_Button::down_box(Fl_Boxtype bt)

The first form returns the current down box type, which is drawn when `value()` is non-zero.

The second form sets the down box type. The default value of 0 causes FLTK to figure out the correct matching down version of `box()` .

int Fl_Button::set()

Same as `value(1)`.

void Fl_Button::setonly()

Turns on this button and turns off all other radio buttons in the group (calling `value(1)` or `set()` does not do this).

ulong Fl_Button::shortcut() const
void Fl_Button::shortcut(ulong key)

The first form returns the current shortcut key for the button.

The second form sets the shortcut key to `key`. Setting this overrides the use of '&' in the `label()`. The value is a bitwise OR of a key and a set of shift flags, for example `FL_ALT | 'a'` , `FL_ALT | (FL_F + 10)`, or just `'a'`. A value of 0 disables the shortcut.

The key can be any value returned by `Fl::event_key()`, but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by `Fl::event_state()`. If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

uchar Fl_Button::type() const
void Fl_Button::type(uchar t)

The first form of `type()` returns the current button type, which can be one of:

- `0`: The value is unchanged.
- `FL_TOGGLE_BUTTON`: The value is inverted.
- `FL_RADIO_BUTTON`: The value is set to 1, and all other buttons in the current group with `type() == FL_RADIO_BUTTON` are set to zero.

The second form sets the button type to `t`.

char Fl_Button::value() const
int Fl_Button::value(int)

The first form returns the current value (0 or 1). The second form sets the current value.

Fl_When Fl_Widget::when() const
void Fl_Widget::when(Fl_When w)

Controls when callbacks are done. The following values are useful, the default value is `FL_WHEN_RELEASE`:

- `0`: The callback is not done, instead changed() is turned on.
- `FL_WHEN_RELEASE`: The callback is done after the user successfully clicks the button, or when a shortcut is typed.
- `FL_WHEN_CHANGED` : The callback is done each time the value() changes (when the user pushes and releases the button, and as the mouse is dragged around in and out of the button).

class Fl_Chart

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Chart
```

Include Files

```
#include <FL/Fl_Chart.H>
```

Description

This widget displays simple charts and is provided for Forms compatibility.

Methods

• Fl_Chart	• autosize	• clear	• maxsize	• size
• ~Fl_Chart	• bounds	• insert	• replace	• type
• add				

Fl_Chart::Fl_Chart(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Chart` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Chart::~Fl_Chart()

Destroys the `Fl_Chart` widget and all of its data.

void add(double value, const char *label = NULL, uchar color = 0)

The `add` method adds the `value` and optionally `label` and `color` to the chart.

uchar autosize(void) const
void autosize(uchar onoff)

The `autosize` method controls whether or not the chart will automatically adjust the bounds of the chart. The first form returns a boolean value that is non-zero if auto-sizing is enabled and zero is auto-sizing is disabled.

The second form of `autosize` sets the auto-sizing property to `onoff`.

void bounds(double *a, double *b)
void bounds(double a, double b)

The `bounds` method gets or sets the lower and upper bounds of the chart values to `a` and `b` respectively.

void clear(void)

The `clear` method removes all values from the chart.

void insert(int pos, double value, const char *label = NULL, uchar color = 0)

The `insert` method inserts a data value at the given position `pos`. Position 0 is the first data value.

int maxsize(void) const
void maxsize(int n)

The `maxsize` method gets or sets the maximum number of data values for a chart. If you do not call this method then the chart will be allowed to grow to any size depending on available memory.

void replace(int pos, double value, const char *label = NULL, uchar color = 0)

The `replace` method replaces data value `pos` with `value`, `label`, and `color`. Position 0 is the first data value.

int size(void) const

The `size` method returns the number of data values in the chart.

uchar type() const
void type(uchar t)

The first form of `type()` returns the current chart type. The chart type can be one of the following:

FL_BAR_CHART

Each sample value is drawn as a vertical bar.

FL_FILLED_CHART

The chart is filled from the bottom of the graph to the sample values.

FL_HORBAR_CHART

Each sample value is drawn as a horizontal bar.

FL_LINE_CHART

The chart is drawn as a polyline with vertices at each sample value.

FL_PIE_CHART

A pie chart is drawn with each sample value being drawn as a proportionate slice in the circle.

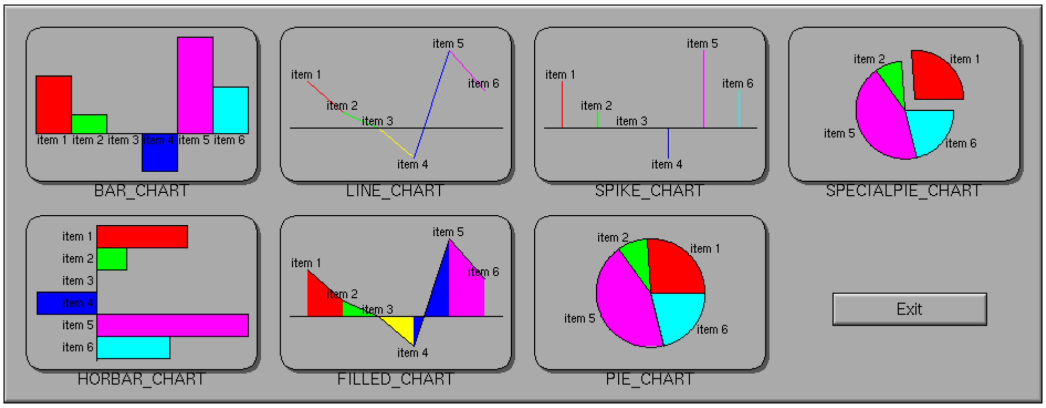
FL_SPECIALPIE_CHART

Like FL_PIE_CHART, but the first slice is separated from the pie.

FL_SPIKE_CHART

Each sample value is drawn as a vertical line.

The second form of `type()` sets the chart type to `t`.



class Fl_Check_Button

Class Hierarchy

```
Fl_Button
   |
   +----Fl_Check_Button
```

Include Files

```
#include <FL/Fl_Check_Button.H>
```

Description

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for `type()` and `when()`.

The `Fl_Check_Button` subclass display the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_DIAMOND_DOWN_BOX. The color of the light when on is controlled with `selection_color(), which defaults to FL_RED.`

Methods

- Fl_Check_Button
- ~Fl_Check_Button

Fl_Check_Button::Fl_Check_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Check_Button` widget using the given position, size, and label string.

Fl_Check_Button::~Fl_Check_Button()

The destructor deletes the check button.

class Fl_Choice

Class Hierarchy

```
Fl_Menu_
   |
   +----Fl_Choice
```

Include Files

```
#include <FL/Fl_Choice.H>
```

Description

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of `Fl_Menu_Item` objects. Motif calls this an OptionButton.

The only difference between this and a `Fl_Menu_Button` is that the name of the most recent chosen menu item is displayed inside the box, while the label is displayed outside the box. However, since the use of this is most often to control a single variable rather than do individual callbacks, some of the `Fl_Menu_Button` methods are redescribed here in those terms.

When the user picks an item off the menu the `value()` is set to that item and then the callback is done.

All three mouse buttons pop up the menu. The Forms behavior of the first two buttons to increment/decrement the choice is not implemented. This could be added with a subclass, however.

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the `label()`. See `Fl_Button` for a description of this.

Typing the `shortcut()` of any of the items will do exactly the same as when you pick the item with the mouse. The '&' character in item names are only looked at when the menu is popped up, however.

Methods

- Fl_Choice
- ~Fl_Choice
- clear_changed
- changed
- down_box
- set_changed
- value

Fl_Choice::Fl_Choice(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Choice` widget using the given position, size, and label string. The default boxtype is `FL_UP_BOX`.

The constructor sets `menu()` to `NULL`. See `Fl_Menu_` for the methods to set or change the menu.

virtual Fl_Choice::~Fl_Choice()

The destructor removes the `Fl_Choice` widget and all of its menu items.

int Fl_Choice::value() const
int Fl_Choice::value(int)
int Fl_Choice::value(const Fl_Menu *)

The value is the index into the `Fl_Menu` array of the last item chosen by the user. It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one. Changing it causes a `redraw()`.

int Fl_Widget::changed() const

This value is true if the user picks a different value. *It is turned off by `value()` and just before doing a callback (the callback can turn it back on if desired).*

void Fl_Widget::set_changed()

This method sets the `changed()` flag.

void Fl_Widget::clear_changed()

This method clears the `changed()` flag.

Fl_Boxtype Fl_Choice::down_box() const
void Fl_Choice::down_box(Fl_Boxtype b)

The first form gets the current down box, which is used when the menu is popped up. The default down box type is `FL_DOWN_BOX` The second form sets the current down box type to `b`.

class Fl_Clock

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Clock
```

Include Files

```
#include <FL/Fl_Clock.H>
```

Description

This widget provides a round analog clock display and is provided for Forms compatibility. It installs a 1-second timeout callback using `Fl::add_timeout()`.

Methods

- Fl_Clock
- ~Fl_Clock
- hour
- minute
- second
- value

Fl_Clock::Fl_Clock(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Clock` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Clock::~Fl_Clock()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Clock` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Clock`*first*, so that it is destroyed last.

int Fl_Clock::hour() const

Returns the current hour (0 to 23).

int Fl_Clock::minute() const

Returns the current minute (0 to 59).

int Fl_Clock::second() const

Returns the current second (0 to 60, 60 = leap second).

void Fl_Clock::value(ulong v)
void Fl_Clock::value(int h, int m, int s)
ulong Fl_Clock::value(void)

The first two forms of `value` set the displayed time to the given UNIX time value or specific hours, minutes, and seconds.

The third form of `value` returns the displayed time in seconds since the UNIX epoch (January 1, 1970).

class Fl_Color_Chooser

Class Hierarchy

```
Fl_Group
   |
   +----Fl_Color_Chooser
```

Include Files

```
#include <FL/Fl_Color_Chooser.H>
```

Description

The `Fl_Color_Chooser` widget provides a standard RGB color chooser. You can place any number of these into a panel of your own design. This widget contains the hue box, value slider, and rgb input fields from the above diagram (it does not have the color chips or the Cancel or OK buttons). The callback is done every time the user changes the rgb value. It is not done if they move the hue control in a way that produces the *same* rgb value, such as when saturation or value is zero.

Methods

- Fl_Color_Chooser
- ~Fl_Color_Chooser
- add

Fl_Color_Chooser::Fl_Color_Chooser(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Color_Chooser` widget using the given position, size, and label string. The recommended dimensions are 200x95. The color is initialized to black.

virtual Fl_Color_Chooser::~Fl_Color_Chooser()

The destructor removes the color chooser and all of its controls.

double Fl_Color_Chooser::hue() const

Return the current hue. 0 <= hue < 6. Zero is red, one is yellow, two is green, etc. *This value is convienent for the internal calculations - some other systems consider hue to run from zero to one, or from 0 to 360.*

double Fl_Color_Chooser::saturation() const

Returns the saturation. 0 <= saturation <= 1.

double Fl_Color_Chooser::value() const

Returns the value/brightness. 0 <= value <= 1.

double Fl_Color_Chooser::r() const

Returns the current red value. 0 <= r <= 1.

double Fl_Color_Chooser::g() const

Returns the current green value. 0 <= g <= 1.

double Fl_Color_Chooser::b() const

Returns the current blue value. 0 <= b <= 1.

int Fl_Color_Chooser::rgb(double, double, double)

Sets the current rgb color values. Does not do the callback. Does not clamp (but out of range values will produce psychedelic effects in the hue selector).

int Fl_Color_Chooser::hsv(double,double,double)

Set the hsv values. The passed values are clamped (or for hue, modulus 6 is used) to get legal values. Does not do the callback.

static void Fl_Color_Chooser::hsv2rgb(double, double, double, double&, double&, double&)

This *static* method converts HSV colors to RGB colorspace.

static void Fl_Color_Chooser::rgb2hsv(double, double, double, double&, double&, double&)

This *static* method converts RGB colors to HSV colorspace.

class Fl_Counter

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Counter
```

Include Files

```
#include <FL/Fl_Counter.H>
```

Description

The `Fl_Counter` widget is provided for forms compatibility. It controls a single floating point value.

Methods

- Fl_Counter
- ~Fl_Counter
- lstep
- type

Fl_Counter::Fl_Counter(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Counter` widget using the given position, size, and label string. The default type is `FL_NORMAL_COUNTER`.

virtual Fl_Counter::~Fl_Counter()

Destroys the valuator.

double Fl_Counter::lstep() const

Set the increment for the double-arrow buttons. The default value is 1.0.

type(uchar)

Sets the type of counter:

- `FL_NORMAL_COUNTER` - Displays a counter with 4 arrow buttons.
- `FL_SIMPLE_COUNTER` - Displays a counter with only 2 arrow buttons.

class Fl_Dial

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Dial
```

Include Files

```
#include <FL/Fl_Dial.H>
```

Description

The `Fl_Dial` widget provides a circular dial to control a single floating point value.

Methods

- Fl_Dial
- ~Fl_Dial
- angle1
- angle2
- angles
- type

Fl_Dial::Fl_Dial(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Dial` widget using the given position, size, and label string. The default type is `FL_NORMAL_DIAL`.

virtual Fl_Dial::~Fl_Dial()

Destroys the valuator.

short Fl_Dial::angle1() const;
void Fl_Dial::angle1(short);
short Fl_Dial::angle2() const;
void Fl_Dial::angle2(short);
void Fl_Dial::angles(short a, short b);

Sets the angles used for the minimum and maximum values. The default values are 45 and 315 (0 degrees is straight down and the angles progress clockwise). Normally angle1 is less than angle2, but if you reverse them the dial moves counter-clockwise.

type(uchar)

Sets the type of the dial to:

- `FL_NORMAL_DIAL` - Draws a normal dial with a knob.
- `FL_LINE_DIAL` - Draws a dial with a line.
- `FL_FILL_DIAL` - Draws a dial with a filled arc.

class Fl_Double_Window

Class Hierarchy

```
Fl_Window
   |
   +----Fl_Double_Window
```

Include Files

```
#include <FL/Fl_Double_Window.H>
```

Description

The `Fl_Double_Window` class provides a double-buffered window. If possible this will use the X double buffering extension (Xdbe). If not, it will draw the window data into an off-screen pixmap, and then copy it to the on-screen window.

It is highly recommended that you put the following code before the first `show()` of *any* window in your program:

```
Fl::visual(FL_DOUBLE|FL_INDEX)
```

This makes sure you can use Xdbe on servers where double buffering does not exist for every visual.

Methods

- Fl_Double_Window
- ~Fl_Double_Window
- pixmap

Fl_Double_Window::Fl_Double_Window(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Double_Window` widget using the given position, size, and label (title) string.

virtual Fl_Double_Window::~Fl_Double_Window()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code.

class Fl_End

Class Hierarchy

Fl_End

Include Files

```
#include <FL/Fl_Group.H>
```

Description

This is a dummy class that allows you to end a Fl_Group in a constructor list of a class:

```
class MyClass {
  Fl_Group group;
  Fl_Button button_in_group;
  Fl_End end;
  Fl_Button button_outside_group;
  MyClass();
};
MyClass::MyClass() :
  group(10,10,100,100),
  button_in_group(20,20,60,30),
  end(),
  button_outside_group(10,120,60,30)
{}
```

Methods

- Fl_End

Fl_End::Fl_End

The constructor does `Fl_Group::current()->end()`.

class Fl_Float_Input

Class Hierarchy

```
Fl_Input
   |
   +----Fl_Float_Input
```

Include Files

```
#include <FL/Fl_Input.H>
```

Description

The `Fl_Float_Input` class is a subclass of `Fl_Input` that only allows the user to type floating point numbers (sign, digits, decimal point, more digits, 'E' or 'e', sign, digits).

Methods

- Fl_Float_Input
- ~Fl_Float_Input

Fl_Float_Input::Fl_Float_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Float_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Float_Input::~Fl_Float_Input()

Destroys the widget and any value associated with it.

class Fl_Free

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Free
```

Include Files

```
#include <FL/Fl_Free.H>
```

Description

Emulation of the Forms "free" widget. This emulation allows the free demo to run, and appears to be useful for porting programs written in Forms which use the free widget or make subclasses of the Forms widgets.

There are five types of free, which determine when the handle function is called:

```
#define FL_NORMAL_FREE          1
#define FL_SLEEPING_FREE        2
#define FL_INPUT_FREE           3
#define FL_CONTINUOUS_FREE      4
#define FL_ALL_FREE             5
```

An FL_INPUT_FREE accepts FL_FOCUS events. A FL_CONTINUOUS_FREE sets a timeout callback 100 times a second and provides a FL_STEP event, this has obvious detrimental effects on machine performance. FL_ALL_FREE does both. FL_SLEEPING_FREE are deactivated.

Methods

- Fl_Free
- ~Fl_Free

Fl_Free(uchar type, int, int, int, int, const char*l, FL_HANDLEPTR hdl)

The constructor takes both the `type` and the `handle` function. The handle function should be declared as follows:

```
int
handle_function(Fl_Widget *w,
                int       event,
                float     event_x,
                float     event_y,
                char      key)
```

This function is called from the the `handle()` method in response to most events, and is called by the `draw()` method. The `event` argument contains the event type:

```
// old event names for compatability:
#define FL_MOUSE                FL_DRAG
#define FL_DRAW                 0
#define FL_STEP                 9
#define FL_FREEMEM              12
#define FL_FREEZE               FL_UNMAP
#define FL_THAW                 FL_MAP
```

virtual Fl_Free::~Fl_Free()

The destructor will call the handle function with the event `FL_FREE_MEM`.

class Fl_Gl_Window

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Gl_Window
           |
           +----Fl_Pack, Fl_Scroll, Fl_Tabs, Fl_Tile, Fl_Window
```

Include Files

```
#include <FL/Fl_Gl_Window.H>
```

Description

The `Fl_Gl_Window` widget sets things up so OpenGL works, and also keeps an OpenGL "context" for that window, so that changes to the lighting and projection may be reused between redraws. Fl_Gl_Window also flushes the OpenGL streams and swaps buffers after `draw()` returns.

OpenGL hardware typically provides some overlay bit planes, which are very useful for drawing UI controls atop your 3D graphics. If the overlay hardware is not provided, FLTK tries to simulate the overlay, This works pretty well if your graphics are double buffered, but not very well for single-buffered.

Methods

• Fl_Gl_Window	• draw	• hide	• make_overlay_current	• redraw_overlay
• ~Fl_Gl_Window	• draw_overlay	• invalidate	• mode	• swap_buffers
• can_do	• handle	• make_current	• ortho	• valid
• can_do_overlay				

Fl_Gl_Window::Fl_Gl_Window(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Gl_Window` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`. The default mode is `FL_RGB|FL_DOUBLE|FL_DEPTH`.

virtual Fl_Gl_Window::~Fl_Gl_Window()

The destructor removes the widget and destroys the OpenGL context associated with it.

virtual void Fl_Gl_Window::draw(void)

`Fl_Gl_Window::draw()` is a pure virtual method. You must subclass `Fl_Gl_Window` and provide an implementation for `draw()`. You may also provide an implementation of draw_overlay() if you want to draw into the overlay planes. You can avoid reinitializing the viewport and lights and other things by checking `valid()` at the start of `draw()` and only doing the initialization if it is false.

The `draw()` method can *only* use OpenGL calls. Do not attempt to call X, any of the functions in <FL/fl_draw.H>, or `glX` directly. Do not call `gl_start()` or `gl_finish()`.

If double-buffering is enabled in the window, the back and front buffers are swapped after this function is completed.

const int Fl_Gl_Window::mode() const
int Fl_Gl_Window::mode(int m)

Set or change the OpenGL capabilites of the window. The value can be any of the following OR'd together:

- `FL_RGB` - RGB color (not indexed)
- `FL_RGB8` - RGB color with at least 8 bits of each color
- `FL_INDEX` - Indexed mode
- `FL_SINGLE` - not double buffered
- `FL_DOUBLE` - double buffered
- `FL_ACCUM` - accumulation buffer
- `FL_ALPHA` - alpha channel in color
- `FL_DEPTH` - depth buffer
- `FL_STENCIL` - stencil buffer
- `FL_MULTISAMPLE` - multisample antialiasing

`FL_RGB` and `FL_SINGLE` have a value of zero, so they are "on" unless you give `FL_INDEX` or `FL_DOUBLE`.

If the desired combination cannot be done, FLTK will try turning off `FL_MULTISAMPLE`. If this also fails the `show()` will call `Fl::error()` and not show the window.

You can change the mode while the window is displayed. This is most useful for turning double-buffering on and off. Under X this will cause the old X window to be destroyed and a new one to be created. If this is a top-level window this will unfortunately also cause the window to blink, raise to the top, and be de-iconized, and the `xid()` will change, possibly breaking other code. It is best to make the GL window a child of another window if you wish to do this!

static int Fl_Gl_Window::can_do(int)
int Fl_Gl_Window::can_do() const

Returns non-zero if the hardware supports the given or current OpenGL mode.

char Fl_Gl_Window::valid() const
void Fl_Gl_Window::valid(char i)

`Fl_Gl_Window::valid()` is turned off when FLTK creates a new context for this window or when the window resizes, and is turned on *after*`draw()` is called. You can use this inside your `draw()` method to avoid unneccessarily initializing the OpenGL context. Just do this:

```
void mywindow::draw() {
  if (!valid()) {
    glViewport(0,0,w(),h());
    glFrustum(...);
    glLight(...);
    ...other initialization...
  }
  ... draw your geometry here ...
}
```

You can turn `valid()` on by calling `valid(1)`. You should only do this after fixing the transformation inside a `draw()` or after `make_current()`. This is done automatically after `draw()` returns.

void Fl_Gl_Window::invalidate()

The `invalidate()` method turns off `valid()` and is equivalent to calling `value(0)`.

void Fl_Gl_Window::ortho()

Set the projection so 0,0 is in the lower left of the window and each pixel is 1 unit wide/tall. If you are drawing 2D images, your `draw()` method may want to call this if `valid()` is false.

void Fl_Gl_Window::make_current()

The `make_current()` method selects the OpenGL context for the widget. It is called automatically prior to the `draw()` method being called and can also be used to implement feedback and/or selection within the `handle()` method.

void Fl_Gl_Window::make_overlay_current()

The `make_overlay_current()` method selects the OpenGL context for the widget's overlay. It is called automatically prior to the `draw_overlay()` method being called and can also be used to implement feedback and/or selection within the `handle()` method.

void Fl_Gl_Window::swap_buffers()

The `swap_buffers()` method swaps the back and front buffers. It is called automatically after the `draw()` method is called.

void Fl_Gl_Window::hide()

Hides the window and destroys the OpenGL context.

int Fl_Gl_Window::can_do_overlay()

Returns true if the hardware overlay is possible. If this is false, FLTK will try to simulate the overlay, with significant loss of update speed. Calling this will cause FLTK to open the display.

void Fl_Gl_Window::redraw_overlay()

This method causes `draw_overlay` to be called at a later time. Initially the overlay is clear, if you want the window to display something in the overlay when it first appears, you must call this immediately after you `show()` your window.

virtual void Fl_Gl_Window::draw_overlay()

You must implement this virtual function if you want to draw into the overlay. The overlay is cleared before this is called. You should draw anything that is not clear using OpenGL. You must use `gl_color(i)` to choose colors (it allocates them from the colormap using system-specific calls), and remember that you are in an indexed OpenGL mode and drawing anything other than flat-shaded will probably not work.

Both this function and `Fl_Gl_Window::draw()` should check `Fl_Gl_Window::valid()` and set the same transformation. If you don't your code may not work on other systems. Depending on the OS, and on whether overlays are real or simulated, the OpenGL context may be the same or different between the overlay and main window.

class Fl_Group

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Group
           |
           +----Fl_Pack, Fl_Scroll, Fl_Tabs, Fl_Tile, Fl_Window
```

Include Files

```
#include <FL/Fl_Group.H>
```

Description

The `Fl_Group` class is the FLTK container widget. It maintains an array of child widgets. These children can themselves be any widget including `Fl_Group`. The most important subclass of `Fl_Group` is `Fl_Window`, however groups can also be used to control radio buttons or to enforce resize behavior.

Methods

• Fl_Group	• add_resizable	• child	• end	• remove
• ~Fl_Group	• array	• children	• find	• resizable
• add	• begin	• current	• insert	

Fl_Group::Fl_Group(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Group` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Group::~Fl_Group()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Group` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Group`*first*, so that it is destroyed last.

void Fl_Group::add(Fl_Widget &w)
void Fl_Group::add(Fl_Widget *w)

The widget is removed from it's current group (if any) and then added to the end of this group.

void Fl_Group::insert(Fl_Widget &w, int n)

The widget is removed from it's current group (if any) and then inserted into this group. It is put at index `n` (or at the end if `n >= children()`. This can also be used to rearrange the windgets inside a group.

void Fl_Group::insert(Fl_Widget &w, Fl_Widget* beforethis)

This does `insert(w, find(beforethis))`. This will append the widget if `beforethis` is not in the group.

void Fl_Group::remove(Fl_Widget &w)

Removes a widget from the group. This does nothing if the widget is not currently a child of this group.

static Fl_Group *Fl_Group::current()
static void Fl_Group::current(Fl_Group *w)

`current()` returns the currently active group. The Fl_Widget constructor automatically does `current()->add(widget)` if this is not null. To prevent new widgets from being added to a group, call `Fl_Group::current(0)`.

void Fl_Group::begin()

`begin()` sets the current group so you can build the widget tree by just constructing the widgets. `begin()` is automatically called by the constructor for Fl_Group (and thus for Fl_Window as well). `begin()`*is exactly the same as*`current(this)`.

Don't forget to `end()` *the group or window!*

void Fl_Group::end()

`end()`*is exactly the same as*`current(this->parent())`. Any new widgets added to the widget tree will be added to the parent of the group.

const Fl_Widget **Fl_Group::array() const

Returns a pointer to the array of children. *This pointer is only valid until the next time a child is added or removed.*

Fl_Widget *Fl_Group::child(int n) const

Returns `array()[n]`. *No range checking is done!*

int Fl_Group::children() const

Returns how many child widgets the group has.

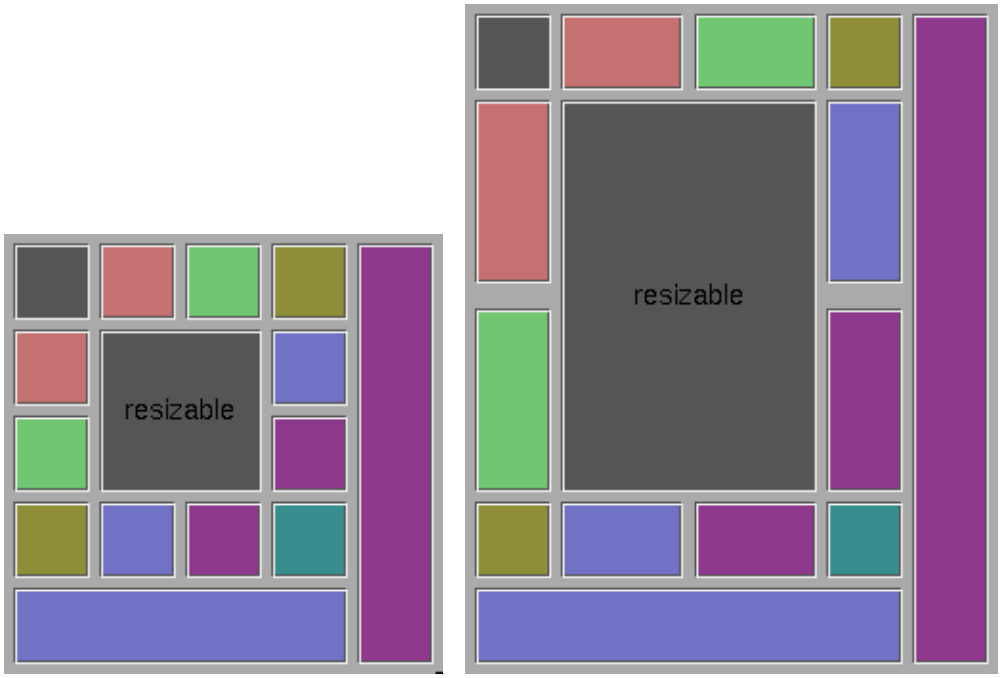
int Fl_Group::find(const Fl_Widget *w) const
int Fl_Group::find(const Fl_Widget &w) const

Searches the child array for the widget and returns the index. Returns `children()` if the widget is `NULL` or not found.

void Fl_Group::resizable(Fl_Widget *box)
void Fl_Group::resizable(Fl_Widget &box)
Fl_Widget *Fl_Group::resizable() const

The resizable widget defines the resizing box for the group. When the group is resized it calculates a new size and position for all of its children. Widgets that are horizontally or vertically inside the dimensions of the box are scaled to the new size. Widgets outside the box are moved.

In these examples the gray area is the resizable:



The resizable may be set to the group itself (this is the default value for an `Fl_Group`, although `NULL` is the default for an `Fl_Window`), in which case all the contents are resized. If the resizable is `NULL` then all widgets remain a fixed size and distance from the top-left corner.

It is possible to achieve any type of resize behavior by using an invisible `Fl_Box` as the resizable and/or by using a hierarchy of child `Fl_Group`'s.

Fl_Group &Fl_Group::add_resizable(Fl_Widget &box)

Adds a widget to the group and makes it the resizable widget.

class Fl_Hold_Browser

Class Hierarchy

```
Fl_Browser
   |
   +----Fl_Hold_Browser
```

Include Files

```
#include <FL/Fl_Hold_Browser.H>
```

Description

The `Fl_Hold_Browser` class is a subclass of `Fl_Browser` which lets the user select a single item, or no items by clicking on the empty space. As long as the mouse button is held down the item pointed to by it is highlighted, and this highlighting remains on when the mouse button is released. Normally the callback is done when the user releases the mouse, but you can change this with `when()`.

See `Fl_Browser` for methods to add and remove lines from the browser.

Methods

- Fl_Hold_Browser
- ~Fl_Hold_Browser
- deselect
- select
- value

Fl_Hold_Browser::Fl_Hold_Browser(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Hold_Browser` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Hold_Browser::~Fl_Hold_Browser()

The destructor *also deletes all the items in the list.*

int Fl_Browser::deselect()

Same as `value(0)`.

int Fl_Browser::select(int,int=1)
int Fl_Browser::selected(int) const

You can use these for compatibility with `Fl_Multi_Browser`. If you turn on the selection of more than one line the results are unpredictable.

int Fl_Browser::value() const
void Fl_Browser::value(int)

Set or get which line is selected. This returns zero if no line is selected, so be aware that this can happen in a callback.

class Fl_Input

Class Hierarchy

```
Fl_Input_
   |
   +----Fl_Input
           |
           +----Fl_Float_Input, Fl_Int_Input,
                Fl_Multiline_Input, Fl_Secret_Input
```

Include Files

```
#include <FL/Fl_Input.H>
```

Description

This is the FLTK text input widget. It displays a single line of text and lets the user edit it. Normally it is drawn with an inset box and a white background. The text may contain any characters (even 0), and will correctly display anything, using ^X notation for unprintable control characters and \nnn notation for unprintable characters with the high bit set. It assummes the font can draw any characters in the ISO8859-1 character set.

Mouse button 1	Moves the cursor to this point. Drag selects characters. Double click selects words. Triple click selects all text. Shift+click extends the selection.
Mouse button 2	Insert the current X selection at the cursor (unlike Motif this does not move the insertion point to the mouse). If the widget does not have the input focus (and thus no cursor) it puts the cursor where clicked and inserts the selection there.
Mouse button 3	Currently acts like button 1.
Backspace	Deletes one character to the left, or deletes the selected region.
Enter	May cause the callback, see when().
^A or Home	Go to start of line.
^B or Left	Move left
^C	Copy the selection to the clipboard
^D or Delete	Deletes one character to the right or deletes the selected region. *Due to silly historical X problems, the Delete key will act like Backspace until you type a "real" backspace.*
^E or End	Go to the end of line.
^F or Right	Move right
^K	Delete to the end of line (next \n character) or deletes a single

	\n character. These deletions are all concatenated into the clipboard.
^N or Down	Move down (for Fl_Multiline_Input only, otherwise it moves to the next input field).
^P or Up	Move up (for Fl_Multiline_Input only, otherwise it moves to the previous input field).
^Q or RightCtrl or Compose	Start a compose-character sequence. The next one or two keys typed define the character to insert. This also can be used to "quote" control characters.
^U	Delete everything.
^V or ^Y	Paste the clipboard
^X or ^W	Copy the region to the clipboard and delete it.
^Z or ^_	Undo. This is a single-level undo mechanism, but all adjacent deletions and insertions are concatenated into a single "undo". Often this will undo a lot more than you expected.
Shift+move	Move the cursor but also extend the selection.

Methods

- Fl_Input
- ~Fl_Input
- cursor_color
- index
- size
- static_value
- textcolor
- textfont
- textsize
- value
- when

Fl_Input::Fl_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Input::~Fl_Input()

Destroys the widget and any value associated with it.

const char *Fl_Input::value() const
int Fl_Input::value(const char*)
int Fl_Input::value(const char*, int)

The first form returns the current value, which is a pointer to the internal buffer and is valid only until the next event is handled.

The second two forms change the text and set the mark and the point to the end of it. The string is copied to the internal buffer. Passing `NULL` is the same as "". This returns non-zero if the new value is different than the current one. You can use the second version to directly set the length if you know it already or want to put nul's in the text.

int Fl_Input::static_value(const char*)
int Fl_Input::static_value(const char*, int)

Change the text and set the mark and the point to the end of it. The string is *not* copied. If the user edits the string it is copied to the internal buffer then. This can save a great deal of time and memory if your program is rapidly changing the values of text fields, but this will only work if the passed string remains unchanged until either the `Fl_Input` is destroyed or `value()` is called again.

int Fl_Input::size() const

Returns the number of characters in `value()`. This may be greater than `strlen(value())` if there are nul characters in it.

char Fl_Input::index(int) const

Same as `value()[n]`, but may be faster in plausible implementations. No bounds checking is done.

Fl_When Fl_Widget::when() const
void Fl_Widget::when(Fl_When)

Controls when callbacks are done. The following values are useful, the default value is `FL_WHEN_RELEASE`:

- `0`: The callback is not done, but `changed()` is turned on.
- `FL_WHEN_CHANGED`: The callback is done each time the text is changed by the user.
- `FL_WHEN_RELEASE`: The callback will be done when this widget loses the focus, including when the window is unmapped. This is a useful value for text fields in a panel where doing the callback on every change is wasteful. However the callback will also happen if the mouse is moved out of the window, which means it should not do anything visible (like pop up an error message). You might do better setting this to zero, and scanning all the items for `changed()` when the OK button on a panel is pressed.
- `FL_WHEN_ENTER_KEY`: If the user types the Enter key, the entire text is selected, and the callback is done if the text has changed. Normally the Enter key will navigate to the next field (or insert a newline for a `Fl_Mulitline_Input`), this changes the behavior.
- `FL_WHEN_ENTER_KEY|FL_WHEN_NOT_CHANGED`: The Enter key will do the callback even if the text has not changed. Useful for command fields.

Fl_Color Fl_Input::textcolor() const
void Fl_Input::textcolor(Fl_Color)

Gets or sets the color of the text in the input field.

Fl_Font Fl_Input::textfont() const
void Fl_Input::textfont(Fl_Font)

Gets or sets the font of the text in the input field.

uchar Fl_Input::textsize() const
void Fl_Input::textsize(uchar)

Gets or sets the size of the text in the input field.

Fl_Color Fl_Input::cursor_color() const
void Fl_Input::cursor_color(Fl_Color)

Get or set the color of the cursor. This is black by default.

class Fl_Input_

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Input_
           |
           +----Fl_Input
```

Include Files

```
#include <FL/Fl_Input_.H>
```

Description

This is a virtual base class below `Fl_Input`. It has all the same interfaces, but lacks the `handle()` and `draw()` method. You may want to subclass it if you are one of those people who likes to change how the editing keys work.

This can act like any of the subclasses of Fl_Input, by setting `type()` to one of the following values:

```
#define FL_NORMAL_INPUT         0
#define FL_FLOAT_INPUT          1
#define FL_INT_INPUT            2
#define FL_MULTILINE_INPUT      4
#define FL_SECRET_INPUT         5
```

Methods

- Fl_Input_
- ~Fl_Input_
- copy
- copy_cuts
- cut
- drawtext
- handletext
- insert
- lineboundary
- mark
- maybe_do_callback
- position
- replace
- undo
- up_down_position
- wordboundary

Fl_Input_::Fl_Input_(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Input_` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Input_::~Fl_Input_()

The destructor removes the widget and any value associated with it.

int Fl_Input_::wordboundary(int i) const

Returns true if position `i` is at the start or end of a word.

int Fl_Input_::lineboundary(int i) const

Returns true if position `i` is at the start or end of a line.

void Fl_Input_::drawtext(int,int,int,int)

Draw the text in the passed bounding box. If `damage() FL_DAMAGE_ALL` is true, this assummes the area has already been erased to `color()`. Otherwise it does minimal update and erases the area itself.

void Fl_Input_::handletext(int e,int,int,int,int)

Default handler for all event types. Your `handle()` method should call this for all events that it does not handle completely. You must pass it the same bounding box as passed to `draw()`. Handles `FL_PUSH`, `FL_DRAG`, `FL_RELEASE` to select text, handles `FL_FOCUS` and `FL_UNFOCUS` to show and hide the cursor.

int Fl_Input_::up_down_position(int i, int keepmark=0)

Do the correct thing for arrow keys. Sets the position (and mark if *keepmark* is zero) to somewhere in the same line as *i*, such that pressing the arrows repeatedly will cause the point to move up and down.

void Fl_Input_::maybe_do_callback()

Does the callback if `changed()` is true or if `when() FL_WHEN_NOT_CHANGED` is non-zero. You should call this at any point you think you should generate a callback.

int Fl_Input_::position() const
int Fl_Input_::position(int new_position, int new_mark)
int Fl_Input_::position(int new_position_and_new_mark)

The input widget maintains two pointers into the string. The "position" is where the cursor is. The "mark" is the other end of the selected text. If they are equal then there is no selection. Changing this does not affect the clipboard (use `copy()` to do that).

Changing these values causes a `redraw()`. The new values are bounds checked. The return value is non-zero if the new position is different than the old one. `position(n)` is the same as `position(n,n)`. `mark(n)` is the same as `position(position(),n)`.

int Fl_Input_::mark() const
int Fl_Input_::mark(int new_mark)

Gets or sets the current selection mark. `mark(n)` is the same as `position(position(),n)`.

int Fl_Input_::replace(int a, int b, const char *insert, int length=0)

This call does all editing of the text. It deletes the region between `a` and `b` (either one may be less or equal to the other), and then inserts the string `insert` at that point and leaves the `mark()` and `position()` after the insertion. Does the callback if `when() FL_WHEN_CHANGED` and there is a change.

Set `start` and `end` equal to not delete anything. Set `insert` to `NULL` to not insert anything.

`length` must be zero or `strlen(insert)`, this saves a tiny bit of time if you happen to already know the length of the insertion, or can be used to insert a portion of a string or a string containing nul's.

`a` and `b` are clamped to the 0..`size()` range, so it is safe to pass any values.

`cut()` and `insert()` are just inline functions that call `replace()`.

int Fl_Input_::cut()
int Fl_Input_::cut(int n)
int Fl_Input_::cut(int a, int b);

`Fl_Input_::cut()` deletes the current selection. `cut(n)` deletes `n` characters after the `position()`. `cut(-n)` deletes `n` characters before the `position()` . `cut(a,b)` deletes the characters between offsets `a` and `b`. `A`, `b`, and `n` are all clamped to the size of the string. The mark and point are left where the deleted text was.

If you want the data to go into the clipboard, do `Fl_Input_::copy()` before calling `Fl_Input_::cut()`, or do `Fl_Input_::copy_cuts()` afterwards.

int Fl_Input_::insert(const char *t,int l=0)

Insert the string `t` at the current position, and leave the mark and position after it. If `l` is not zero then it is assummed to be `strlen(t)`.

int Fl_Input_::copy()

Put the current selection between `mark()` and `position()` into the clipboard. Does not replace the old clipboard contents if `position()` and `mark()` are equal.

int Fl_Input_::undo()

Does undo of several previous calls to `replace()`. Returns non-zero if any change was made.

int Fl_Input_::copy_cuts()

Copy all the previous contiguous cuts from the undo information to the clipboard. This is used to make ^K work.

class Fl_Int_Input

Class Hierarchy

```
Fl_Input
   |
   +----Fl_Int_Input
```

Include Files

```
#include <FL/Fl_Input.H>
```

Description

The `Fl_Int_Input` class is a subclass of `Fl_Input` that only allows the user to type decimal digits (or hex numbers of the form `0xaef`).

Methods

- Fl_Int_Input
- ~Fl_Int_Input

Fl_Int_Input::Fl_Int_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Int_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Int_Input::~Fl_Int_Input()

Destroys the widget and any value associated with it.

class Fl_Light_Button

Class Hierarchy

```
Fl_Button
   |
   +----Fl_Light_Button
```

Include Files

```
#include <FL/Fl_Light_Button.H>
```

Description

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for `type()` and `when()`.

The `Fl_Light_Button` subclass display the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_DOWN_BOX. The color of the light when on is controlled with `selection_color(), which defaults to FL_YELLOW.`

Methods

- Fl_Light_Button
- ~Fl_Light_Button

Fl_Light_Button::Fl_Light_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Light_Button` widget using the given position, size, and label string.

Fl_Light_Button::~Fl_Light_Button()

The destructor deletes the check button.

class Fl_Menu_

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Menu_----Fl_Menu_Item
           |
           +----Fl_Choice, Fl_Menu_Bar, Fl_Menu_Button
```

Include Files

```
#include <FL/Fl_Menu_.H>
```

Description

All widgets that have a menu in FLTK are subclassed off of this class. Currently FLTK provides you with `Fl_Menu_Button`, `Fl_Menu_Bar`, and `Fl_Choice`.

The class contains a pointer to an array of structures of type `Fl_Menu_Item`. The array may either be supplied directly by the user program, or it may be "private": a dynamically allocated array managed by the Fl_Menu_.

Methods

• Fl_Menu_	• copy	• mode	• size	• textfont
• ~Fl_Menu_	• down_box	• remove	• test_shortcut	• textsize
• add	• global	• replace	• text	• value
• clear	• menu	• shortcut	• textcolor	

Fl_Menu_::Fl_Menu_(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Menu_` widget using the given position, size, and label string.`menu()` is initialized to null.

virtual Fl_Menu_::~Fl_Menu_()

If the menu array is private the memory it uses is freed.

const Fl_Menu_Item* Fl_Menu_::menu() const

Returns a pointer to the array of Fl_Menu_Items. This will either be the value passed to `menu(value)` or the private copy.

void Fl_Menu_::menu(const Fl_Menu_Item*)

Set the menu array pointer directly. If the old menu is private it is deleted. `NULL` is allowed and acts the same as a zero-length menu. If you try to modify the array (with add(), replace(), or delete()) a private copy is automatically done.

void Fl_Menu_::copy(const Fl_Menu_Item*)

The menu is set to a private copy of the passed Fl_Menu_Item array. This is useful if you want to modify the flags of the menu items.

void Fl_Menu_::clear()

Same as `menu(NULL)`, set the array pointer to null, indicating a zero-length menu.

int Fl_Menu_::size() const

This returns the number of Fl_Menu_Item structures that make up the menu, correctly counting submenus. This includes the "terminator" item at the end. To copy a menu array you need to copy `size()*sizeof(Fl_Menu_Item)` bytes. If the menu is `NULL` this returns zero (an empty menu will return 1).

int Fl_Menu_::add(const char* label, const char* shortcut, Fl_Callback*, void *user_data=0, int flags=0)

Adds a new menu item, with a `title` string, `shortcut` string, `callback`, argument to the callback, and flags. If the menu array was directly set with menu(x) then copy() is done to make a private array.

Text is a string of the form "foo/bar/baz", this example will result in a submenu called "foo" and one in that called "bar" and and entry called "baz". The text is copied to new memory and can be freed. The other arguments (including the shortcut) are copied into the menu item unchanged.

If an item exists already with that name then it is replaced with this new one. Otherwise this new one is added to the end of the correct menu or submenu. The return value is the offset into the array that the new entry was placed at.

The return value is the index into the array that the entry was put.

int Fl_Menu_::add(const char *)

The passed string is split at any '|' characters and then `add(s,0,0,0,0)` is done with each section. This is often useful if you are just using the value, and is compatable with Forms and other GL programs.

void Fl_Menu_::replace(int n, const char *)

Changes the text of item `n`. This is the only way to get slash into an add()'ed menu item. If the menu array was directly set with menu(x) then copy() is done to make a private array.

void Fl_Menu_::remove(int n)

Deletes item `n` from the menu. If the menu array was directly set with menu(x) then copy() is done to make a private array.

void Fl_Menu_::shortcut(int i, int n);

Changes the shortcut of item `i` to `n`.

void Fl_Menu_::mode(int i, int x);

Changes the flags of item `i`. For a list of the flags, see Fl_Menu_Item.

int Fl_Menu_::value() const
int Fl_Menu_::value(int)
const Fl_Menu_Item* mvalue() const
int Fl_Menu_::value(const Fl_Menu_Item*)

The value is the index into `menu()` of the last item chosen by the user. It is zero initially. You can set it as an integer, or set it with a pointer to a menu item. The set routines return non-zero if the new value is different than the old one.

const Fl_Menu_Item* Fl_Menu_::test_shortcut()

Only call this in response to `FL_SHORTCUT events`. If the event matches an entry in the menu that entry is selected and the callback will be done (or `changed()` will be set). This allows shortcuts directed at one window to call menus in another.

void Fl_Menu_::global()

Make the shortcuts for this menu work no matter what window has the focus when you type it. This is done by using `Fl::add_handler()`. This `Fl_Menu_` widget does not have to be visible (ie the window it is in can be hidden, or it does not have to be put in a window at all).

Currently there can be only one `global()` menu. Setting a new one will replace the old one. There is no way to remove the `global()` setting (so don't destroy the widget!)

const char* Fl_Menu_::text() const
const char* Fl_Menu_::text(int i) const

Returns the title of the last item chosen, or of item `i`.

Fl_Color Fl_Menu_::textcolor() const
void Fl_Menu_::textcolor(Fl_Color)

Get or set the current color of menu item labels.

Fl_Font Fl_Menu_::textfont() const
void Fl_Menu_::textfont(Fl_Font)

Get or set the current font of menu item labels.

uchar Fl_Menu_::textsize() const
void Fl_Menu_::textsize(uchar)

Get or set the font size of menu item labels.

Fl_Boxtype Fl_Menu_::down_box() const
void Fl_Menu_::down_box(Fl_Boxtype)

This box type is used to surround the currently-selected items in the menus. If this is `FL_NO_BOX` then it acts like `FL_THIN_UP_BOX` and `selection_color()` acts like `FL_WHITE`, for back compatability.

class Fl_Menu_Bar

Class Hierarchy

```
Fl_Menu_
   |
   +----Fl_Menu_Bar
```

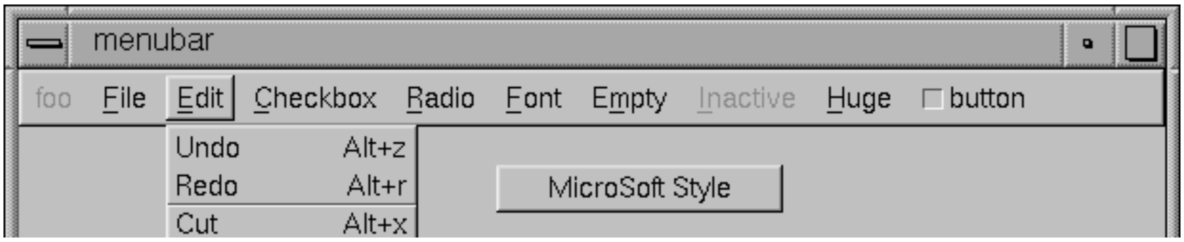
Include Files

```
#include <FL/Fl_Menu_Bar.H>
```

Description

This widget provides a standard menubar interface. Usually you will put this widget along the top edge of your window. The height of the widget should be 30 for the menu titles to draw correctly with the default font.

The items on the bar and the menus they bring up are defined by a single `Fl_Menu_Item` array. Because a `Fl_Menu_Item` array defines a hierarchy, the top level menu defines the items in the menubar, while the submenus define the pull-down menus. Sub-sub menus and lower pop up to the right of the submenus.



If there is an item in the top menu that is not a title of a submenu, then it acts like a "button" in the menubar. Clicking on it will pick it.

When the user picks an item off the menu, the item's callback is done with the menubar as the `Fl_Widget*` argument. If the item does not have a callback the menubar's callback is done instead.

Submenus will also pop up in response to shortcuts indicated by putting a '&' character in the name field of the menu item. If you put a '&' character in a top-level "button" then the shortcut picks it. The '&' character in submenus is ignored until the menu is popped up.

Typing the `shortcut()` of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse.

Methods

- Fl_Menu_Bar
- ~Fl_Menu_Bar

Fl_Menu_Bar::Fl_Menu_Bar(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Menu_Bar` widget using the given position, size, and label string. The default boxtype is `FL_UP_BOX`.

The constructor sets `menu()` to `NULL`. See `Fl_Menu_` for the methods to set or change the menu.

`labelsize()`, `labelfont()`, and `labelcolor()` are used to control how the menubar items are drawn. They are initialized from the `Fl_Menu` static variables, but you can change them if desired.

`label()` is ignored unless you change `align()` to put it outside the menubar.

virtual Fl_Menu_Bar::~Fl_Menu_Bar()

The destructor removes the `Fl_Menu_Bar` widget and all of its menu items.

class Fl_Menu_Button

Class Hierarchy

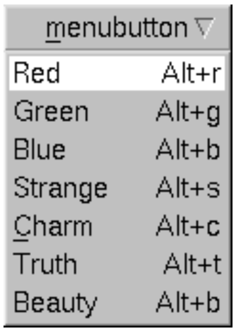
```
Fl_Menu_
   |
   +----Fl_Menu_Button
```

Include Files

```
#include <FL/Fl_Menu_Button.H>
```

Description

This is a button that when pushed pops up a menu (or hierarchy of menus) defined by an array of `Fl_Menu_Item` objects.



Normally any mouse button will pop up a menu and it is lined up below the button as shown in the picture. However an `Fl_Menu_Button` may also control a pop-up menu. This is done by setting the `type()` , see below.

The menu will also pop up in response to shortcuts indicated by putting a '&' character in the `label()`.

Typing the `shortcut()` of any of the menu items will cause callbacks exactly the same as when you pick the item with the mouse. The '&' character in menu item names are only looked at when the menu is popped up, however.

When the user picks an item off the menu, the item's callback is done with the menu_button as the `Fl_Widget*` argument. If the item does not have a callback the menu_button's callback is done instead.

Methods

- Fl_Menu_Button
- ~Fl_Menu_Button
- popup
- type

Fl_Menu_Button::Fl_Menu_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Menu_Button` widget using the given position, size, and label string. The default boxtype is `FL_UP_BOX`.

The constructor sets `menu()` to `NULL`. See `Fl_Menu_` for the methods to set or change the menu.

virtual Fl_Menu_Button::~Fl_Menu_Button()

The destructor removes the `Fl_Menu_Button` widget and all of its menu items.

const Fl_Menu* Fl_Menu_Button::popup()

Act exactly as though the user clicked the button or typed the shortcut key. The menu appears, it waits for the user to pick an item, and if they pick one it sets `value()` and does the callback or sets `changed()` as described above. The menu item is returned or `NULL` if the user dismisses the menu.

void Fl_Menu_Button::type(uchar)

If `type()` is zero a normal menu button is produced. If it is nonzero then this is a pop-up menu. The bits in `type()` indicate what mouse buttons pop up the menu. For convienece the constants `Fl_Menu_Button::POPUP1, POPUP2, POPUP3, POPUP12, POPUP13, POPUP23`, and `POPUP123` are defined. `Fl_Menu_Button::POPUP3` is usually what you want.

A popup menu button is invisible and does not interfere with any events other than the mouse button specified (and any shortcuts). The widget can be stretched to cover all your other widgets by putting it last in the hierarchy so it is "on top". You can also make several widgets covering different areas for context-sensitive popup menus.

The popup menus appear with the cursor pointing at the previously selected item. This is a *feature*. If you don't like it, do `value(0)` after the menu items are picked to forget the current item.

struct Fl_Menu_Item

Class Hierarchy

```
struct Fl_Menu_Item
```

Include Files

```
#include <FL/Fl_Menu_Item.H>
```

Description

The `Fl_Menu_Item` structure defines a single menu item that is used by the Fl_Menu_ class. This structure is defined in `<FL/Fl_Menu_Item.H>`

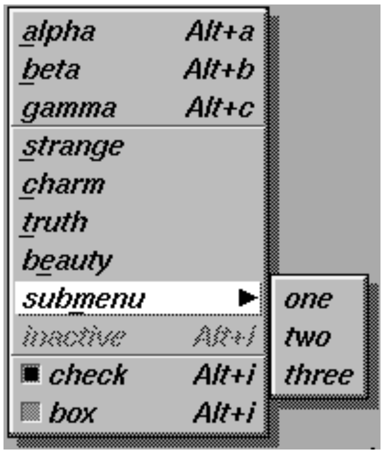
```
struct Fl_Menu_Item {
  const char*   text; // label()
  ulong         shortcut_;
  Fl_Callback*  callback_;
  void*         user_data_;
  int           flags;
  uchar         labeltype_;
  uchar         labelfont_;
  uchar         labelsize_;
  uchar         labelcolor_;
};

enum { // values for flags:
  FL_MENU_INACTIVE      = 1,
  FL_MENU_TOGGLE        = 2,
  FL_MENU_VALUE         = 4,
  FL_MENU_RADIO         = 8,
  FL_MENU_INVISIBLE     = 0x10,
  FL_SUBMENU_POINTER    = 0x20,
  FL_SUBMENU            = 0x40,
  FL_MENU_DIVIDER       = 0x80,
  FL_MENU_HORIZONTAL    = 0x100
};
```

Typically menu items are statically defined; for example:

```
Fl_Menu_Item popup[] = {
  {"&alpha",   FL_ALT+'a', the_cb, (void*)1},
  {"&beta",    FL_ALT+'b', the_cb, (void*)2},
  {"gamma",    FL_ALT+'c', the_cb, (void*)3, FL_MENU_DIVIDER},
  {"&strange",  0,   strange_cb},
  {"&charm",    0,   charm_cb},
  {"&truth",    0,   truth_cb},
  {"b&eauty",   0,   beauty_cb},
  {"sub&menu",  0,   0, 0, FL_SUBMENU},
    {"one"},
    {"two"},
    {"three"},
  {0},
```

```
  {"inactive", FL_ALT+'i', 0, 0, FL_MENU_INACTIVE|FL_MENU_DIVIDER},
  {"invisible",FL_ALT+'i', 0, 0, FL_MENU_INVISIBLE},
  {"check",    FL_ALT+'i', 0, 0, FL_MENU_TOGGLE|FL_MENU_VALUE},
  {"box",      FL_ALT+'i', 0, 0, FL_MENU_TOGGLE},
{0}};
```

produces:



A submenu title is identified by the bit `FL_SUBMENU` in the `flags` field, and ends with a `label()` that is `NULL`. You can nest menus to any depth. A pointer to the first item in the submenu can be treated as an `Fl_Menu` array itself. It is also possible to make seperate submenu arrays with `FL_SUBMENU_POINTER` flags.

You should use the method functions to access structure members and not access them directly to avoid compatibility problems with future releases of FLTK.

Methods

• label	• user_data	• radio	• show	• popup
• labeltype	• argument	• value	• hide	• pulldown
• labelcolor	• do_callback	• set	• active	• test_shortcut
• labelfont	• shortcut	• setonly	• activate	• size
• labelsize	• submenu	• clear	• deactivate	• next
• callback	• checkbox	• visible		

const char* Fl_Menu_Item::label() const
void Fl_Menu_Item::label(const char*)
void Fl_Menu_Item::label(Fl_Labeltype, const char*)

This is the title of the item. A `NULL` here indicates the end of the menu (or of a submenu). A '&' in the item will print an underscore under the next letter, and if the menu is popped up that letter will be a "shortcut" to pick that item. To get a real '&' put two in a row.

Fl_Labeltype Fl_Menu_Item::labeltype() const
void Fl_Menu_Item::labeltype(Fl_Labeltype)

A `labeltype` identifies a routine that draws the label of the widget. This can be used for special effects such as emboss, or to use the `label()` pointer as another form of data such as a bitmap. The value `FL_NORMAL_LABEL` prints the label as text.

Fl_Color Fl_Menu_Item::labelcolor() const
void Fl_Menu_Item::labelcolor(Fl_Color)

This color is passed to the labeltype routine, and is typically the color of the label text. This defaults to `FL_BLACK`. If this color is not black fltk will *not* use overlay bitplanes to draw the menu - this is so that images put in the menu draw correctly.

Fl_Font Fl_Menu_Item::labelfont() const
void Fl_Menu_Item::labelfont(Fl_Font)

Fonts are identified by small 8-bit indexes into a table. See the enumeration list for predefined fonts. The default value is a Helvetica font. The function `Fl::set_font()` can define new fonts.

uchar Fl_Menu_Item::labelsize() const
void Fl_Menu_Item::labelsize(uchar)

Gets or sets the label font pixel size/height.

typedef void (Fl_Callback)(Fl_Widget*, void*)
Fl_Callback* Fl_Menu_Item::callback() const
void Fl_Menu_Item::callback(Fl_Callback*, void* = 0)
void Fl_Menu_Item::callback(void (*)(Fl_Widget*))

Each item has space for a callback function and an argument for that function. Due to back compatability, the `Fl_Menu_Item` itself is not passed to the callback, instead you have to get it by calling `((Fl_Menu_*)w)->mvalue()` where `w` is the widget argument.

void* Fl_Menu_Item::user_data() const
void Fl_Menu_Item::user_data(void*)

Get or set the `user_data` argument that is sent to the callback function.

void Fl_Menu_Item::callback(void (*)(Fl_Widget*, long), long = 0)
long Fl_Menu_Item::argument() const
void Fl_Menu_Item::argument(long)

For convenience you can also define the callback as taking a `long` argument. This is implemented by casting this to a `Fl_Callback` and casting the `long` to a `void*` and may not be portable to some machines.

void Fl_Menu_Item::do_callback(Fl_Widget*)
void Fl_Menu_Item::do_callback(Fl_Widget*, void*)
void Fl_Menu_Item::do_callback(Fl_Widget*, long)

Call the `Fl_Menu_Item` item's callback, and provide the `Fl_Widget` argument (and optionally override the user_data() argument). You must first check that `callback()` is non-zero before calling this.

ulong Fl_Menu_Item::shortcut() const
void Fl_Menu_Item::shortcut(ulong)

Sets exactly what key combination will trigger the menu item. The value is a logical 'or' of a key and a set of shift flags, for instance `FL_ALT+'a'` or `FL_ALT+FL_F+10` or just 'a'. A value of zero disables the shortcut.

The key can be any value returned by `Fl::event_key()`, but will usually be an ASCII letter. Use a lower-case letter unless you require the shift key to be held down.

The shift flags can be any set of values accepted by `Fl::event_state()`. If the bit is on that shift key must be pushed. Meta, Alt, Ctrl, and Shift must be off if they are not in the shift flags (zero for the other bits indicates a "don't care" setting).

int Fl_Menu_Item::submenu() const

Returns true if either `FL_SUBMENU` or `FL_SUBMENU_POINTER` is on in the flags. `FL_SUBMENU` indicates an embedded submenu that goes from the next item through the next one with a `NULLlabel()`. `FL_SUBMENU_POINTER` indicates that `user_data()` is a pointer to another menu array.

int Fl_Menu_Item::checkbox() const

Returns true if a checkbox will be drawn next to this item. This is true if `FL_MENU_TOGGLE` or `FL_MENU_RADIO` is set in the flags.

int Fl_Menu_Item::radio() const

Returns true if this item is a radio item. When a radio button is selected all "adjacent" radio buttons are turned off. A set of radio items is delimited by an item that has `radio()` false, or by an item with `FL_MENU_DIVIDER` turned on.

int Fl_Menu_Item::value() const

Returns the current value of the check or radio item.

void Fl_Menu_Item::set()

Turns the check or radio item "on" for the menu item. Note that this does not turn off any adjacent radio items like `set_only()` does.

void Fl_Menu_Item::setonly()

Turns the radio item "on" for the menu item and turns off adjacent radio item.

void Fl_Menu_Item::clear()

Turns the check or radio item "off" for the menu item.

int Fl_Menu_Item::visible() const

Gets the visibility of an item.

void Fl_Menu_Item::show()

Makes an item visible in the menu.

void Fl_Menu_Item::hide()

Hides an item in the menu.

int Fl_Menu_Item::active() const

Get whether or not the item can be picked.

void Fl_Menu_Item::activate()

Allows a menu item to be picked.

void Fl_Menu_Item::deactivate()

Prevents a menu item from being picked. Note that this will also cause the menu item to appear grayed-out.

const Fl_Menu_Item *Fl_Menu_Item::popup(int X, int Y, const char* title = 0, const Fl_Menu_Item* picked = 0, const Fl_Menu_* button = 0) const

This method is called by widgets that want to display menus. The menu stays up until the user picks an item or dismisses it. The selected item (or `NULL` if none) is returned. *This does not do the callbacks or change the state of check or radio items.*

`X,Y` is the position of the mouse cursor, relative to the window that got the most recent event (usually you can pass `Fl::event_x()` and `Fl::event_y()` unchanged here).

`title` is a character string title for the menu. If non-zero a small box appears above the menu with the title in it.

The menu is positioned so the cursor is centered over the item `picked`. This will work even if `picked` is in a submenu. If `picked` is zero or not in the menu item table the menu is positioned with the cursor in the top-left corner.

`button` is a pointer to an `Fl_Menu_` from which the color and boxtypes for the menu are pulled. If `NULL` then defaults are used.

const Fl_Menu_Item *Fl_Menu_Item::pulldown(int X, int Y, int W, int H, const Fl_Menu_Item* picked = 0, const Fl_Menu_* button = 0, const Fl_Menu_Item* title = 0, int menubar=0) const

`pulldown()` is similar to `popup()`, but a rectangle is provided to position the menu. The menu is made at least `W` wide, and the `picked` item is centered over the rectangle (like `Fl_Choice` uses). If `picked` is zero or not found, the menu is aligned just below the rectangle (like a pulldown menu).

The `title` and `menubar` arguments are used internally by the `Fl_Menu_Bar` widget.

const Fl_Menu_Item* Fl_Menu_Item::test_shortcut() const

This is designed to be called by a widgets `handle()` method in response to a `FL_SHORTCUT` event. If the current event matches one of the items shortcut, that item is returned. If the keystroke does not match any shortcuts then `NULL` is returned. This only matches the `shortcut()` fields, not the letters in the title preceeded by '

int Fl_Menu_Item::size()

Returns the number of Fl_Menu_Item structures that make up this menu, correctly counting submenus. This includes the "terminator" item at the end. So to copy a menu you need to copy `size()*sizeof(Fl_Menu_Item)` bytes.

const Fl_Menu_Item* Fl_Menu_Item::next(int n=1) const
Fl_Menu_Item* Fl_Menu_Item::next(int n=1);

Advance a pointer by `n` items through a menu array, skipping the contents of submenus and invisible items. There are two calls so that you can advance through const and non-const data.

class Fl_Menu_Window

Class Hierarchy

```
Fl_Single_Window
   |
   +----Fl_Menu_Window
```

Include Files

```
#include <FL/Fl_Menu_Window.H>
```

Description

The `Fl_Menu_Window` widget is a window type used for menus. By default the window is drawn in the hardware overlay planes if they are available so that the menu don't force the rest of the window to redraw.

Methods

- Fl_Menu_Window
- ~Fl_Menu_Window
- clear_overlay
- set_overlay

Fl_Menu_Window::Fl_Menu_Window(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Menu_Window` widget using the given position, size, and label string.

virtual Fl_Menu_Window::~Fl_Menu_Window()

Destroys the window and all of its children.

Fl_Menu_Window::clear_overlay();

Tells FLTK to use normal drawing planes instead of overlay planes. This is usually necessary if your menu contains multi-color pixmaps.

Fl_Menu_Window::set_overlay()

Tells FLTK to use hardware overlay planes if they are available.

class Fl_Multi_Browser

Class Hierarchy

```
Fl_Browser
   |
   +----Fl_Multi_Browser
```

Include Files

```
#include <FL/Fl_Multi_Browser.H>
```

Description

The `Fl_Multi_Browser` class is a subclass of `Fl_Browser` which lets the user select any set of the lines. The user interface is Macintosh style: clicking an item turns off all the others and selects that one, dragging selects all the items the mouse moves over, and shift + click toggles the items. This is different then how forms did it. Normally the callback is done when the user releases the mouse, but you can change this with `when()`.

See `Fl_Browser` for methods to add and remove lines from the browser.

Methods

- Fl_Multi_Browser
- ~Fl_Multi_Browser
- deselect
- select
- value

Fl_Multi_Browser::Fl_Multi_Browser(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Multi_Browser` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Multi_Browser::~Fl_Multi_Browser()

The destructor *also deletes all the items in the list*.

int Fl_Browser::deselect()

Deselects all lines.

int Fl_Browser::select(int,int=1)
int Fl_Browser::selected(int) const

Selects one or more lines or gets the current selection state of a line.

int Fl_Browser::value() const
void Fl_Browser::value(int)

Selects a single line or gets the last toggled line. This returns zero if no line has been toggled, so be aware that this can happen in a callback.

class Fl_Multiline_Input

Class Hierarchy

```
Fl_Input
   |
   +----Fl_Multiline_Input
```

Include Files

```
#include <FL/Fl_Input.H>
```

Description

This input field displays '\n' characters as new lines rather than ^J, and accepts the Return, Tab, and up and down arrow keys. This is for editing multiline text.

This is far from the nirvana of text editors, and is probably only good for small bits of text, 10 lines at most. I think FLTK can be used to write a powerful text editor, but it is not going to be a built-in feature. Powerful text editors in a toolkit are a big source of bloat.

Methods

- Fl_Multiline_Input
- ~Fl_Multiline_Input

Fl_Multiline_Input::Fl_Multiline_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Multiline_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Multiline_Input::~Fl_Multiline_Input()

Destroys the widget and any value associated with it.

class Fl_Multiline_Output

Class Hierarchy

```
Fl_Output
   |
   +----Fl_Multiline_Output
```

Include Files

```
#include <FL/Fl_Multiline_Output.H>
```

Description

This widget is a subclass of `Fl_Output` that displays multiple lines of text. It also displays tab characters as whitespace to the next column.

Methods

- Fl_Multiline_Output
- ~Fl_Multiline_Output

Fl_Multiline_Output::Fl_Multiline_Output(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Multiline_Output` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Multiline_Output::~Fl_Multiline_Output()

Destroys the widget and any value associated with it.

class Fl_Output

Class Hierarchy

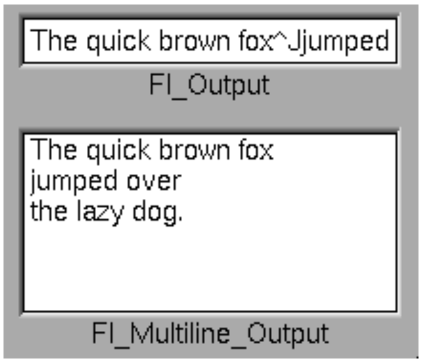
```
Fl_Input_
   |
   +----Fl_Output
           |
           +----Fl_Multiline_Output
```

Include Files

```
#include <FL/Fl_Output.H>
```

Description

This widget displays a piece of text. When you set the `value()` , `Fl_Output` does a `strcpy()` to it's own storage, which is useful for program-generated values. The user may select portions of the text using the mouse and paste the contents into other fields or programs.



There is a single subclass, Fl_Multiline_Output, which allows you to display multiple lines of text.

The text may contain any characters except \0, and will correctly display anything, using ^X notation for unprintable control characters and \nnn notation for unprintable characters with the high bit set. It assummes the font can draw any characters in the ISO-Latin1 character set.

Methods

- Fl_Output
- ~Fl_Output
- cursor_color
- index
- size
- textcolor
- textfont
- textsize

- value

Fl_Output::Fl_Output(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Output` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Output::~Fl_Output()

Destroys the widget and any value associated with it.

const char *Fl_Output::value() const
int Fl_Output::value(const char*)
int Fl_Output::value(const char*, int)

The first form returns the current value, which is a pointer to the internal buffer and is valid only until the value is changed.

The second two forms change the text and set the mark and the point to the end of it. The string is copied to the internal buffer. Passing `NULL` is the same as "". This returns non-zero if the new value is different than the current one. You can use the second version to directly set the length if you know it already or want to put nul's in the text.

int Fl_Output::size() const

Returns the number of characters in `value()`. This may be greater than `strlen(value())` if there are nul characters in it.

char Fl_Output::index(int) const

Same as `value()[n]`, but may be faster in plausible implementations. No bounds checking is done.

Fl_Color Fl_Output::textcolor() const
void Fl_Output::textcolor(Fl_Color)

Gets or sets the color of the text in the input field.

Fl_Font Fl_Output::textfont() const
void Fl_Output::textfont(Fl_Font)

Gets or sets the font of the text in the input field.

uchar Fl_Output::textsize() const
void Fl_Output::textsize(uchar)

Gets or sets the size of the text in the input field.

class Fl_Overlay_Window

Class Hierarchy

```
Fl_Double_Window
   |
   +----Fl_Overlay_Window
```

Include Files

```
#include <FL/Fl_Overlay_Window.H>
```

Description

This window provides double buffering and also the ability to draw the "overlay" which is another picture placed on top of the main image. The overlay is designed to be a rapidly-changing but simple graphic such as a mouse selection box. `Fl_Overlay_Window` uses the overlay planes provided by your graphics hardware if they are available.

If no hardware support is found the overlay is simulated by drawing directly into the on-screen copy of the double-buffered window, and "erased" by copying the backbuffer over it again. This means the overlay will blink if you change the image in the window.

Methods

- Fl_Overlay_Window
- ~Fl_Overlay_Window
- draw_overlay
- redraw_overlay

Fl_Overlay_Window::Fl_Overlay_Window(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Overlay_Window` widget using the given position, size, and label (title) string.

virtual Fl_Overlay_Window::~Fl_Overlay_Window()

Destroys the window and all child widgets.

virtual void Fl_Overlay_Window::draw_overlay() = 0

You must subclass `Fl_Overlay_Window` and provide this method. It is just like a `draw()` method, except it draws the overlay. The overlay will have already been "cleared" when this is called. You can use any of the routines described in <FL/fl_draw.H>.

void Fl_Overlay_Window::redraw_overlay()

Call this to indicate that the overlay data has changed and needs to be redrawn. The overlay will be clear until the first time this is called, so if you want an initial display you must call this after calling `show()`.

class Fl_Pack

Class Hierarchy

```
Fl_Group
   |
   +----Fl_Pack
```

Include Files

```
#include <FL/Fl_Pack.H>
```

Description

This widget was designed to add the functionality of compressing and aligning widgets.

If `type()` is `FL_HORIZONTAL` all the children are resized to the height of the `Fl_Pack`, and are moved next to each other horizontally. If `type()` is not `FL_HORIZONTAL` then the children are resized to the width and are stacked below each other. Then the `Fl_Pack` resizes itself to surround the child widgets.

This widget is needed for the `Fl_Tab`. In addition you may want to put the `Fl_Pack` inside an `Fl_Scroll`.

Methods

• Fl_Pack	• add_resizeable	• child	• end	• remove
• ~Fl_Pack	• array	• children	• find	• resizeable
• add	• begin	• current	• insert	

Fl_Pack::Fl_Pack(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Pack` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Pack::~Fl_Pack()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Pack` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Pack`*first*, so that it is destroyed last.

int Fl_Pack::spacing() const
void Fl_Pack::spacing(int)

Gets or sets the number of extra pixels of blank space that are added between the children.

class Fl_Positioner

Class Hierarchy

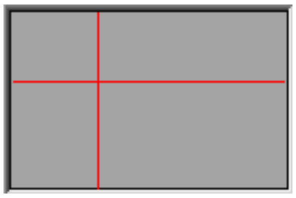
```
Fl_Widget
   |
   +----Fl_Positioner
```

Include Files

```
#include <FL/Fl_Positioner.H>
```

Description

This class is provided for Forms compatibility. It provides 2D input. It would be useful if this could be put atop another widget so that the crosshairs are on top, but this is not implemented. The color of the crosshairs is `selection_color()`.



Methods

- Fl_Positioner
- ~Fl_Positioner
- value
- xbounds
- xstep
- xvalue
- ybounds
- ystep
- yvalue

Fl_Positioner::Fl_Positioner(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Positioner` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Positioner::~Fl_Positioner()

Deletes the widget.

void Fl_Positioner::value(float *x, float *y) const

Returns the current position in `x` and `y`.

void xbounds(float *xmin, float *xmax)
void xbounds(float xmin, float xmax)

Gets or sets the X axis bounds.

void xstep(float x)

Sets the stepping value for the X axis.

float Fl_Positioner::xvalue(void) const
void Fl_Positioner::xvalue(float x)

Gets or sets the X axis coordinate.

void ybounds(float *ymin, float *ymay)
void ybounds(float ymin, float ymay)

Gets or sets the Y axis bounds.

void ystep(float y)

Sets the stepping value for the Y axis.

float Fl_Positioner::yvalue(void) const
void Fl_Positioner::yvalue(float y)

Gets or sets the Y axis coordinate.

class Fl_Repeat_Button

Class Hierarchy

```
Fl_Button
   |
   +----Fl_Repeat_Button
```

Include Files

```
#include <FL/Fl_Repeat_Button.H>
```

Description

The `Fl_Repeat_Button` is a subclass of `Fl_Button` that generates a callback when it is pressed and then repeatedly generates callbacks as long as it is held down. The speed of the repeat is fixed and depends on the implementation.

Methods

- Fl_Repeat_Button
- ~Fl_Repeat_Button

Fl_Repeat_Button::Fl_Repeat_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Repeat_Button` widget using the given position, size, and label string. The default boxtype is `FL_UP_BOX` .

virtual Fl_Repeat_Button::~Fl_Repeat_Button()

Deletes the button.

class Fl_Return_Button

Class Hierarchy

```
Fl_Button
   |
   +----Fl_Return_Button
```

Include Files

```
#include <FL/Fl_Return_Button.H>
```

Description

The `Fl_Return_Button` is a subclass of `Fl_Button` that generates a callback when it is pressed or when the user presses the Enter key. A carriage-return symbol is drawn next to the button label.

Methods

- Fl_Return_Button
- ~Fl_Return_Button

Fl_Return_Button::Fl_Return_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Return_Button` widget using the given position, size, and label string. The default boxtype is `FL_UP_BOX` .

virtual Fl_Return_Button::~Fl_Return_Button()

Deletes the button.

class Fl_Roller

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Roller
```

Include Files

```
#include <FL/Fl_Roller.H>
```

Description

The `Fl_Roller` widget is a "dolly" control commonly used to move 3D objects.

Methods

- Fl_Roller
- ~Fl_Roller

Fl_Roller::Fl_Roller(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Roller` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Roller::~Fl_Roller()

Destroys the valuator.

class Fl_Round_Button

Class Hierarchy

```
Fl_Button
   |
   +----Fl_Round_Button
```

Include Files

```
#include <FL/Fl_Round_Button.H>
```

Description

Buttons generate callbacks when they are clicked by the user. You control exactly when and how by changing the values for `type()` and `when()`.

The `Fl_Round_Button` subclass display the "on" state by turning on a light, rather than drawing pushed in. The shape of the "light" is initially set to FL_ROUND_DOWN_BOX. The color of the light when on is controlled with `selection_color(), which defaults to FL_RED.`

Methods

- Fl_Round_Button
- ~Fl_Round_Button

Fl_Round_Button::Fl_Round_Button(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Round_Button` widget using the given position, size, and label string.

Fl_Round_Button::~Fl_Round_Button()

The destructor deletes the check button.

class Fl_Scroll

Class Hierarchy

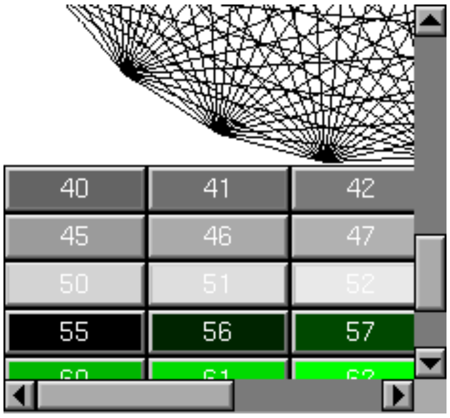
```
Fl_Group
   |
   +----Fl_Scroll
```

Include Files

```
#include <FL/Fl_Scroll.H>
```

Description

This container widget lets you maneuver around a set of widgets much larger than your window. If the child widgets are larger than the size of this object then scrollbars will appear so that you can scroll over to them:



If all of the child widgets are packed together into a solid rectangle then you want to set `box()` to `FL_NO_BOX` or one of the `_FRAME` types. This will result in the best output. However, if the child widgets are a sparse arrangment you must set `box()` to a real `_BOX` type. This can result in some blinking during redrawing, but that can be solved by using a `Fl_Double_Window`.

This widget can also be used to pan around a single child widget "canvas". This child widget should be of your own class, with a `draw()` method that draws the contents. The scrolling is done by changing the `x()` and `y()` of the widget, so this child must use the `x()` and `y()` to position it's drawing. To speed up drawing it should test `fl_clip()` .

Another very useful child is a single `Fl_Pack`, which is itself a group that packs it's children together and changes size to surround them. Filling the `Fl_Pack` with `Fl_Tab` groups (and then putting normal widgets inside those) gives you a very powerful scrolling list of individually-openable panels.

Fluid lets you create these, but you can only lay out objects that fit inside the `Fl_Scroll` without scrolling. Be sure to leave space for the scrollbars, as Fluid won't show these either.

You cannot use `Fl_Window` as a child of this since the clipping is not conveyed to it when drawn, and it will draw over the scrollbars and neighboring objects.

Methods

- Fl_Scroll
- ~Fl_Scroll
- align
- position
- type
- xposition
- yposition

Fl_Scroll::Fl_Scroll(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Scroll` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Scroll::~Fl_Scroll()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Scroll` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Scroll`*first*, so that it is destroyed last.

void Fl_Widget::type(int)

By default you can scroll in both directions, and the scrollbars disappear if the data will fit in the area of the scroll. `type()` can change this:

- `0` - No scrollbars
- `Fl_Scroll::HORIZONTAL` - Only a horizontal scrollbar.
- `Fl_Scroll::VERTICAL` - Only a vertical scrollbar.
- `Fl_Scroll::BOTH` - The default is both scrollbars.
- `Fl_Scroll::HORIZONTAL_ALWAYS` - Horizontal scrollbar always on, vertical always off.
- `Fl_Scroll::VERTICAL_ALWAYS` - Vertical scrollbar always on, horizontal always off.
- `Fl_Scroll::BOTH_ALWAYS` - Both always on.

void Fl_Scroll::scrollbar.align(int)
void Fl_Scroll::hscrollbar.align(int)

This is used to change what side the scrollbars are drawn on. If the `FL_ALIGN_LEFT` bit is on, the vertical scrollbar is on the left. If the `FL_ALIGN_TOP` bit is on, the horizontal scrollbar is on the top.

int Fl_Scroll::xposition() const

Gets the current horizontal scrolling position.

int Fl_Scroll::yposition() const

Gets the current vertical scrolling position.

void Fl_Scroll::position(int w, int h)

Sets the upper-lefthand corner of the scrolling region.

class Fl_Scrollbar

Class Hierarchy

```
Fl_Slider
   |
   +----Fl_Scrollbar
```

Include Files

```
#include <FL/Fl_Scrollbar.H>
```

Description

The `Fl_Scrollbar` widget displays a slider with arrow buttons at the ends of the scrollbar. Clicking on the arrows move up/left and down/right by `linesize()`. Scrollbars also accept `FL_SHORTCUT` events: the arrows move by `linesize()`, and vertical scrollbars take Page Up/Down (they move by the page size minus `linesize()`) and Home/End (they jump to the top or bottom).

Scrollbars have `step(1)` preset (they always return integers). If desired you can set the `step()` to non-integer values. You will then have to use casts to get at the floating-point versions of `value()` from `Fl_Slider`.

Methods

- Fl_Scrollbar
- ~Fl_Scrollbar
- linesize
- value

Fl_Scrollbar::Fl_Scrollbar(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Scrollbar` widget using the given position, size, and label string. You need to do `type(FL_HORIZONTAL)` if you want a horizontal scrollbar.

virtual Fl_Scrollbar::~Fl_Scrollbar()

Destroys the valuator.

int Fl_Scrollbar::linesize() const
void Fl_Scrollbar::linesize(int i)

This number controls how big the steps are that the arrow keys do. In addition page up/down move by the size last sent to `value()` minus one `linesize()`. The default is 16.

int Fl_Scrollbar::value()
int Fl_Scrollbar::value(int position, int size, int top, int total)

The first form returns the integer value of the scrollbar. You can get the floating point value with `Fl_Slider::value()`. The second form sets `value()`, `range()`, and `slider_size()` to make a variable-sized scrollbar. You should call this every time your window changes size, your data changes size, or your scroll position changes (even if in response to a callback from this scrollbar). All necessary calls to `redraw()` are done.

class Fl_Secret_Input

Class Hierarchy

```
Fl_Input
   |
   +----Fl_Secret_Input
```

Include Files

```
#include <FL/Fl_Input.H>
```

Description

The `Fl_Secret_Input` class is a subclass of `Fl_Input` that displays its input as a string of asterisks. This subclass is usually used to recieve passwords and other "secret" information.

Methods

- Fl_Secret_Input
- ~Fl_Secret_Input

Fl_Secret_Input::Fl_Secret_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Secret_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Secret_Input::~Fl_Secret_Input()

Destroys the widget and any value associated with it.

class Fl_Select_Browser

Class Hierarchy

```
Fl_Browser
   |
   +----Fl_Select_Browser
```

Include Files

```
#include <FL/Fl_Select_Browser.H>
```

Description

The `Fl_Select_Browser` class is a subclass of `Fl_Browser` which lets the user select a single item, or no items by clicking on the empty space. As long as the mouse button is held down the item pointed to by it is highlighted. Normally the callback is done when the user presses the mouse, but you can change this with `when()`.

See `Fl_Browser` for methods to add and remove lines from the browser.

Methods

- Fl_Select_Browser
- ~Fl_Select_Browser
- deselect
- select
- value

Fl_Select_Browser::Fl_Select_Browser(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Select_Browser` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Select_Browser::~Fl_Select_Browser()

The destructor *also deletes all the items in the list*.

int Fl_Browser::deselect()

Same as `value(0)`.

int Fl_Browser::select(int,int=1)
int Fl_Browser::selected(int) const

You can use these for compatibility with `Fl_Multi_Browser`. If you turn on the selection of more than one line the results are unpredictable.

int Fl_Browser::value() const

Returns the number of the highlighted item, or zero if none. Notice that this is going to be zero except *during* a callback!

class Fl_Single_Window

Class Hierarchy

```
Fl_Window
   |
   +----Fl_Single_Window
```

Include Files

```
#include <FL/Fl_Single_Window.H>
```

Description

This is the same as `Fl_Window`. However, it is possible that some implementations will provide double-buffered windows by default. This subclass can be used to force single-buffering. This may be useful for modifying existing programs that use incremental update, or for some types of image data, such as a movie flipbook.

Methods

- Fl_Single_Window
- ~Fl_Single_Window

Fl_Single_Window::Fl_Single_Window(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Single_Window` widget using the given position, size, and label (title) string.

virtual Fl_Single_Window::~Fl_Single_Window()

Destroys the window and all child widgets.

class Fl_Slider

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Slider
           |
           +----Fl_Scrollbar, Fl_Value_Slider
```

Include Files

```
#include <FL/Fl_Slider.H>
```

Description

The `Fl_Slider` widget contains a sliding knob inside a box. It if often used as a scrollbar. Moving the box all the way to the top/left sets it to the `minimum()`, and to the bottom/right to the `maximum()`. The `minimum()` may be greater than the `maximum()` to reverse the slider direction.

Methods

- Fl_Slider
- ~Fl_Slider
- scrollvalue
- slider
- slider_size
- type

Fl_Slider::Fl_Slider(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Slider` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Slider::~Fl_Slider()

Destroys the valuator.

int Fl_Slider::scrollvalue(int windowtop, int windowsize, int first, int totalsize)

Returns `Fl_Scrollbar::value()`.

Fl_Boxtype Fl_Slider::slider() const
void Fl_Slider::slider(Fl_Boxtype)

Set the type of box to draw for the moving part of the slider. The color of the moving part (or of the notch in it for the nice sliders) is controlled by `selection_color()`. The default value of zero causes the slider to figure out what to draw from `box()`.

float Fl_Slider::slider_size() const
void Fl_Slider::slider_size(float)

Get or set the dimensions of the moving piece of slider. This is the fraction of the size of the entire widget. If you set this to 1 then the slider cannot move. The default value is .08.

For the "fill" sliders this is the size of the area around the end that causes a drag effect rather than causing the slider to jump to the mouse.

uchar Fl_Widget::type() const
void Fl_Widget::type(uchar t)

Setting this changes how the slider is drawn, which can be one of the following:

- `FL_VERTICAL` - Draws a vertical slider (this is the default).
- `FL_HORIZONTAL` - Draws a horizontal slider.
- `FL_VERT_FILL_SLIDER` - Draws a filled vertical slider, useful as a progress or value meter.
- `FL_HOR_FILL_SLIDER` - Draws a filled horizontal slider, useful as a progress or value meter.
- `FL_VERT_NICE_SLIDER` - Draws a vertical slider with a nice looking control knob.
- `FL_HOR_NICE_SLIDER` - Draws a horizontal slider with a nice looking control knob.

class Fl_Tabs

Class Hierarchy

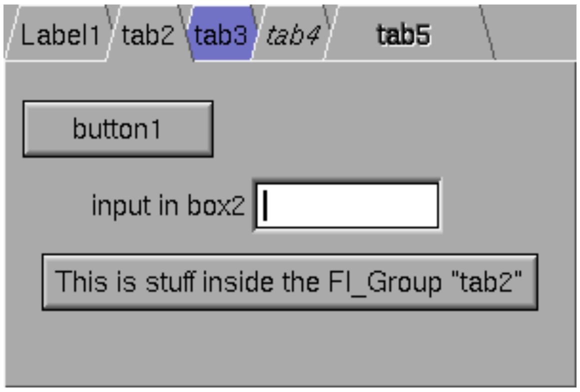
```
Fl_Group
   |
   +----Fl_Tabs
```

Include Files

```
#include <FL/Fl_Tabs.H>
```

Description

The `Fl_Tabs` widget is the "file card tabs" interface that allows you to put lots and lots of buttons and switches in a panel, as popularized by many toolkits.



Clicking the tab makes a child `visible()` (by calling `show()` on it) and all other children are invisible (by calling `hide()` on them). Usually the children are `Fl_Group` widgets containing several widgets themselves.

Each child makes a card, and it's `label()` is printed on the card tab (including the label font and style). The color of that child is used to color the card as well. Currently this only draws nicely if you set `box()` to the default `FL_THIN_UP_BOX` or to `FL_FLAT_BOX`, which gets rid of the edges drawn on the sides and bottom.

The size of the tabs is controlled by the bounding box of the children (there should be some space between the children and the edge of the `Fl_Tabs`), and the tabs may be placed "inverted" on the bottom, this is determined by which gap is larger. It is easiest to lay this out in fluid, using the fluid browser to select each child group and resize them until the tabs look the way you want them to.

Methods

- Fl_Tab
- ~Fl_Tab
- value

Fl_Tab::Fl_Tab(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Tab` widget using the given position, size, and label string. The default boxtype is `FL_THIN_UP_BOX`.

Use `add(Fl_Widget *)` to add each child (which is probably itself a `Fl_Group`). The children should be sized to stay away from the top or bottom edge of the `Fl_Tabs`, which is where the tabs are drawn.

virtual Fl_Tab::~Fl_Tab()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Tab` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Tab`*first*, so that it is destroyed last.

Fl_Widget* Fl_Tabs::value() const
int Fl_Tabs::value(Fl_Widget*)

Gets or sets the currently visible widget/tab.

class Fl_Tile

Class Hierarchy

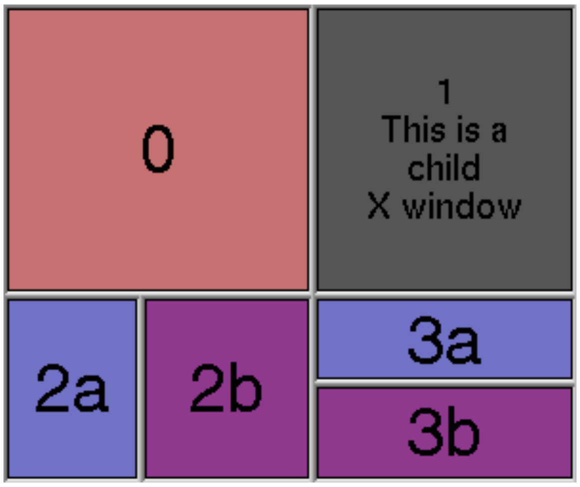
```
Fl_Widget
   |
   +----Fl_Tile
           |
           +----Fl_Pack, Fl_Scroll, Fl_Tabs, Fl_Tile, Fl_Window
```

Include Files

```
#include <FL/Fl_Tile.H>
```

Description

The `Fl_Tile` class lets you resize the children by dragging the border between them:



`Fl_Tile` allows objects to be resized to zero dimensions. To prevent this you can use the `resizable()` to limit where corners can be dragged to.

Even though objects can be resized to zero sizes, they must initially have non-zero sizes so the `Fl_Tile` can figure out their layout. If desired, call `position()` after creating the children but before displaying the window to set the borders where you want.

The "borders" are part of the children, an `Fl_Tile` does not draw any graphics of it's own. In the above example all the final children have `FL_DOWN_BOX` types, and the "ridges" you see are two adjacent `FL_DOWN_BOX`'s drawn next to each other.

Methods

- Fl_Tile
- ~Fl_Tile
- position

- resizeable

Fl_Tile::Fl_Tile(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Tile` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Tile::~Fl_Tile()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Tile` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Tile`*first*, so that it is destroyed last.

void Fl_Tile::position(from_x, from_y, to_x, to_y)

Drag the intersection at `from_x,from_y` to `to_x,to_y`. This redraws all the necessary children.

void Fl_Tile::resizable(Fl_Widget &w)
void Fl_Tile::resizable(Fl_Widget *w)

The "resizable" child widget (which should be invisible) limits where the border can be dragged to. If you don't set it, it will be possible to drag the borders right to the edge, and thus resize objects on the edge to zero width or height. The `resizable()` widget is not resized by dragging any borders.

class Fl_Timer

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Timer
```

Include Files

```
#include <FL/Fl_Timer.H>
```

Description

This is provided only to emulate the Forms Timer widget. It works by making a timeout callback every 1/5 second. This is wasteful and inaccurate if you just want something to happen a fixed time in the future. You should directly call `Fl::add_timeout()` instead.

Methods

- Fl_Timer
- ~Fl_Timer
- direction
- suspended
- value

Fl_Timer::Fl_Timer(uchar type, int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Timer` widget using the given type, position, size, and label string. The `type` parameter can be any of the following symbolic constants:

- `FL_NORMAL_TIMER` - The timer just does the callback and displays the string "Timer" in the widget.
- `FL_VALUE_TIMER` - The timer does the callback and displays the current timer value in the widget.
- `FL_HIDDEN_TIMER` - The timer just does the callback and does not display anything.

virtual Fl_Timer::~Fl_Timer()

Destroys the timer and removes the timeout.

char direction() const
void direction(char d)

Gets or sets the direction of the timer. If the direction is zero then the timer will count up, otherwise it will count down from the initial `value()`.

char suspended() const
void suspended(char d)

Gets or sets whether the timer is suspended.

float value() const
void value(float)

Gets or sets the current timer value.

class Fl_Valuator

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Valuator
           |
           +----Fl_Adjuster, Fl_Counter, Fl_Dial, Fl_Roller,
                Fl_Slider, Fl_Value_Input, Fl_Value_Output,
```

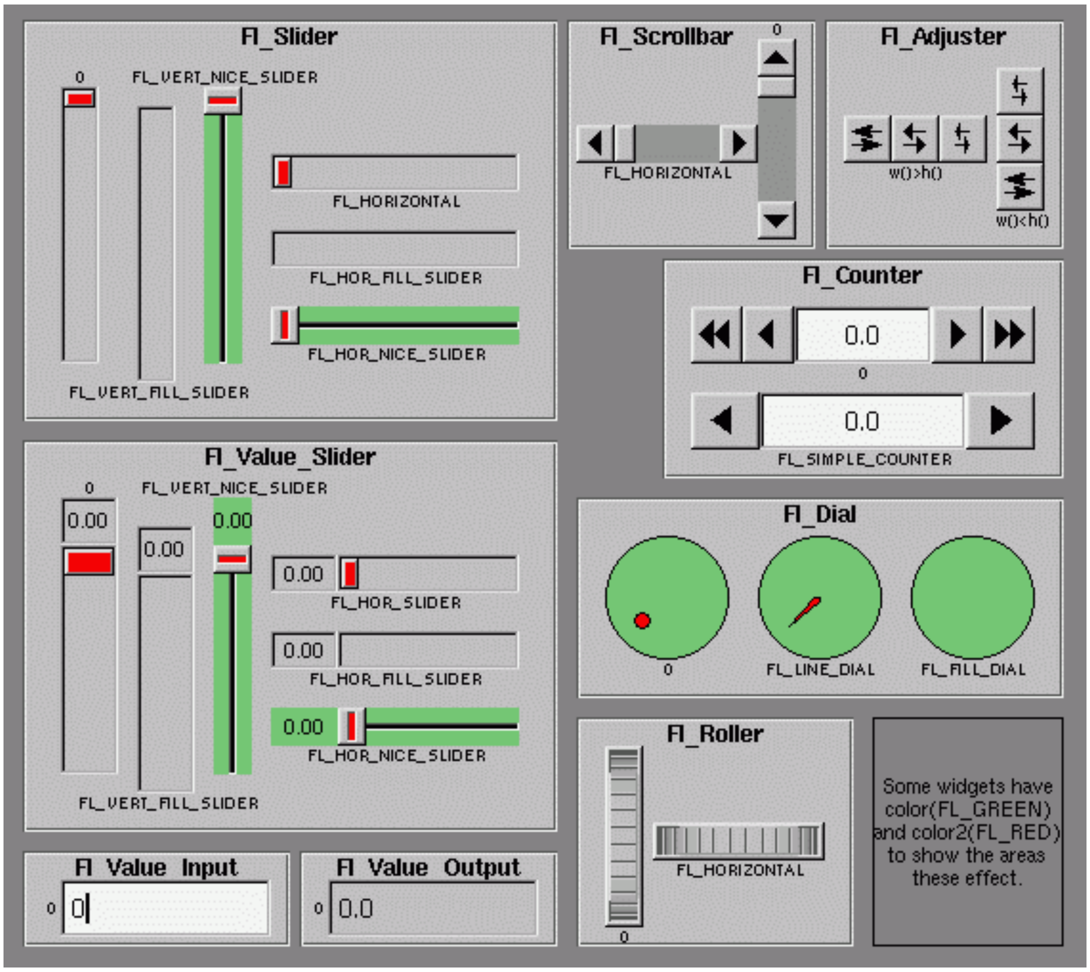
Include Files

```
#include <FL/Fl_Valuator.H>
```

Description

The `Fl_Valuator` class controls a single floating-point value and provides a consistent interface to set the value, range, and step, and insures that callbacks are done the same for every object.

There are probably more of these classes in FLTK than any others:



In the above diagram each box surrounds an actual subclass. These are further differentiated by setting the `type()` of the widget to the symbolic value labeling the widget. The ones labelled "0" are the default versions with a `type(0)`. For consistency the symbol `FL_VERTICAL` is defined as zero.

Methods

• Fl_Valuator	• clamp	• increment	• range	• step
• ~Fl_Valuator	• clear_changed	• maximum	• round	• value
• changed	• format	• minimum	• set_changed	

Fl_Valuator::Fl_Valuator(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Valuator` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX`.

virtual Fl_Valuator::~Fl_Valuator()

Destroys the valuator.

double Fl_Valuator::value() const
int Fl_Valuator::value(double)

Get or set the current value. The new value is *not* clamped or otherwise changed before storing it. Use `clamp()` or `round()` to modify the value before calling this if you want. If the new value is different than the current one the object is redrawn. The initial value is zero.

double Fl_Valuator::minimum() const
void Fl_Valuator::minimum(double)

Gets or sets the minimum value for the valuator.

double Fl_Valuator::maximum() const
void Fl_Valuator::maximum(double)

Gets or sets the maximum value for the valuator.

void Fl_Valuator::range(double min, double max);

Sets the minimum and maximum values for the valuator. When the user manipulates the widget, the value is limited to this range. This clamping is done *after* rounding to the step value (this makes a difference if the range is not a multiple of the step).

The minimum may be greater than the maximum. This has the effect of "reversing" the object so the larger values are in the opposite direction. This also switches which end of the filled sliders is filled.

Some widgets consider this a "soft" range. This means they will stop at the range, but if the user releases and grabs the control again and tries to move it further, it is allowed.

The range may affect the display. You must `redraw()` the widget after changing the range.

double Fl_Valuator::step() const
void Fl_Valuator::step(double)
void Fl_Valuator::step(int A, int B)

Get or set the step value. As the user moves the mouse the value is rounded to the nearest multiple of the step value. This is done *before* clamping it to the range. For most objects the default step is zero.

For precision the step is stored as the ratio of two integers, A/B. You can set these integers directly. Currently setting a floating point value sets the nearest A/1 or 1/B value possible.

int Fl_Valuator::format(char *)

Format the passed value to show enough digits so that for the current step value. If the step has been set to zero then it does a `%g` format. The characters are written into the passed buffer.

double Fl_Valuator::round(double)

Round the passed value to the nearest step increment. Does nothing if step is zero.

double Fl_Valuator::clamp(double)

Clamp the passed value to the valuator range.

double Fl_Valuator::increment(double,int n)

Adds `n` times the step value to the passed value. If step was set to zero it uses `fabs(maximum() - minimum()) / 100`.

int Fl_Widget::changed() const

This value is true if the user has moved the slider. It is turned off by `value(x)` and just before doing a callback (the callback can turn it back on if desired).

void Fl_Widget::set_changed()

Sets the `changed()` flag.

void Fl_Widget::clear_changed()

Clears the `changed()` flag.

class Fl_Value_Input

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Value_Input
```

Include Files

```
#include <FL/Fl_Value_Input.H>
```

Description

The `Fl_Value_Input` widget displays a floating point value. The user can click in the text field and edit it (there is in fact a hidden `Fl_Input` widget with `type(FL_FLOAT_INPUT)` in there), and when they hit return or tab the value updates to what they typed and the callback is done.

If `step()` is not zero, the user can also drag the mouse across the object and thus slide the value. The left button moves one `step()` per pixel, the middle by `10 * step()`, and the left button by `100 * step()`. It is then impossible to select text by dragging across it, although clicking can still move the insertion cursor.

Methods

- Fl_Value_Input
- ~Fl_Value_Input
- cursor_color
- soft
- textcolor
- textfont
- textsize

Fl_Value_Input::Fl_Value_Input(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Value_Input` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX`.

virtual Fl_Value_Input::~Fl_Value_Input()

Destroys the valuator.

Fl_Color Fl_Value_Input::cursor_color() const
void Fl_Value_Input::cursor_color(Fl_Color)

Get or set the color of the cursor. This is black by default.

uchar Fl_Value_Input::soft() const
void Fl_Value_Input::soft(uchar)

If "soft" is turned on, the user is allowed to drag the value outside the range. If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is true.

Fl_Color Fl_Value_Input::textcolor() const
void Fl_Value_Input::textcolor(Fl_Color)

Gets or sets the color of the text in the value box.

Fl_Font Fl_Value_Input::textfont() const
void Fl_Value_Input::textfont(Fl_Font)

Gets or sets the typeface of the text in the value box.

uchar Fl_Value_Input::textsize() const
void Fl_Value_Input::textsize(uchar)

Gets or sets the size of the text in the value box.

class Fl_Value_Output

Class Hierarchy

```
Fl_Valuator
   |
   +----Fl_Value_Output
```

Include Files

```
#include <FL/Fl_Value_Output.H>
```

Description

The `Fl_Value_Output` widget displays a floating point value. If `step()` is not zero, the user can adjust the value by dragging the mouse left and right. The left button moves one `step()` per pixel, the middle by `10 * step()`, and the right button by `100 * step()`.

This is much lighter-weight than `Fl_Value_Input` because it contains no text editing code or character buffer.

Methods

- Fl_Value_Output
- ~Fl_Value_Output
- soft
- textcolor
- textfont
- textsize

Fl_Value_Output::Fl_Value_Output(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Value_Output` widget using the given position, size, and label string. The default boxtype is `FL_NO_BOX` .

virtual Fl_Value_Output::~Fl_Value_Output()

Destroys the valuator.

uchar Fl_Value_Output::soft() const
void Fl_Value_Output::soft(uchar)

If "soft" is turned on, the user is allowed to drag the value outside the range. If they drag the value to one of the ends, let go, then grab again and continue to drag, they can get to any value. Default is one.

Fl_Color Fl_Value_Output::textcolor() const
void Fl_Value_Output::textcolor(Fl_Color)

Gets or sets the color of the text in the value box.

Fl_Font Fl_Value_Output::textfont() const
void Fl_Value_Output::textfont(Fl_Font)

Gets or sets the typeface of the text in the value box.

uchar Fl_Value_Output::textsize() const
void Fl_Value_Output::textsize(uchar)

Gets or sets the size of the text in the value box.

class Fl_Value_Slider

Class Hierarchy

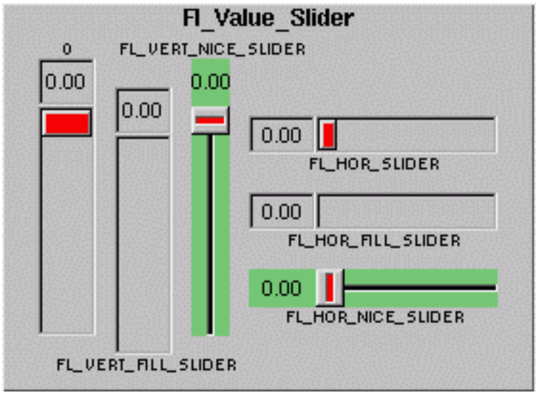
```
Fl_Slider
   |
   +----Fl_Value_Slider
```

Include Files

```
#include <FL/Fl_Value_Slider.H>
```

Description

The `Fl_Value_Slider` widget is a `Fl_Slider` widget with a box displaying the current value.



Methods

- Fl_Value_Slider
- ~Fl_Value_Slider
- textcolor
- textfont
- textsize

Fl_Value_Slider::Fl_Value_Slider(int x, int y, int w, int h, const char *label = 0)

Creates a new `Fl_Value_Slider` widget using the given position, size, and label string. The default boxtype is `FL_DOWN_BOX` .

virtual Fl_Value_Slider::~Fl_Value_Slider()

Destroys the valuator.

Fl_Color Fl_Value_Slider::textcolor() const
void Fl_Value_Slider::textcolor(Fl_Color)

Gets or sets the color of the text in the value box.

Fl_Font Fl_Value_Slider::textfont() const
void Fl_Value_Slider::textfont(Fl_Font)

Gets or sets the typeface of the text in the value box.

uchar Fl_Value_Slider::textsize() const
void Fl_Value_Slider::textsize(uchar)

Gets or sets the size of the text in the value box.

class Fl_Widget

Class Hierarchy

```
Fl_Widget
   |
   +----Fl_Box, Fl_Browser_, Fl_Button, Fl_Chart, Fl_Clock,
        Fl_Free, Fl_Group, Fl_Input_, Fl_Menu_, Fl_Positioner,
        Fl_Timer, Fl_Valuator
```

Include Files

```
#include <FL/Fl_Widget.H>
```

Description

`Fl_Widget` is the base class for all widgets in FLTK. You can't create one of these because the constructor is not public. However you can subclass it.

All "property" accessing methods, such as `color()`, `parent()`, or `argument()` are implemented as trivial inline functions and thus are as fast and small as accessing fields in a structure. Unless otherwise noted, the property setting methods such as `color(n)` or `label(s)` are also trivial inline functions, even if they change the widget's appearance. It is up to the user code to call `redraw()` after these.

Methods

• Fl_Widget	• changed	• hide	• position	• type
• ~Fl_Widget	• clear_changed	• inside	• redraw	• user_data
• activate	• color	• label	• resize	• visible
• active	• contains	• labelcolor	• selection_color	• visible_r
• active_r	• damage	• labelfont	• set_changed	• w
• align	• deactivate	• labelsize	• show	• when
• argument	• default_callback	• labeltype	• size	• window
• box	• do_callback	• output	• take_focus	• x
• callback	• h	• parent	• takesevents	• y

Fl_Widget::Fl_Widget(int x, int y, int w, int h, const char* label=0);

This is the protected constructor for an Fl_Widget, but all derived widgets have a matching public constructor. It takes a value for x(), y(), w(), h(), and an optional value for label().

virtual Fl_Widget::~Fl_Widget();

Destroying single widgets is not very common. It is your responsibility to either remove() them from any enclosing group, or to destroy that group *immediately* after destroying the children.

uchar Fl_Widget::type() const;

This value is used for Forms compatability and to simulate RTTI.

short Fl_Widget::x() const
short Fl_Widget::y() const
short Fl_Widget::w() const
short Fl_Widget::h() const

The position of the upper-left corner of the widget in its enclosing Fl_Window (*not* its parent if that is not an Fl_Window), and its width and height.

virtual void Fl_Widget::resize(int x, int y, int w, int h)
void Fl_Widget::position(short x, short y)
void Fl_Widget::size(short w, short h)

Change the size or position of the widget. This is a virtual function so the widget may implement its own handling of resizing. The default version does *not* do redraw(), that is the parent widget's responsibility (this is because the parent may know a faster way to update the display, such as scrolling from the old position).

`position(x,y)` is a shortcut for `resize(x,y,w(),h())` , and `size(w,h)` is a shortcut for `resize(x(),y(),w,h)` .

Fl_Window* Fl_Widget::window() const;

Return a pointer to the `Fl_Window` that this widget is in (it will skip any and all parent widgets between this and the window). Returns `NULL` if none. Note: for an `Fl_Window`, this returns its *parent* window (if any), not *this* window.

Fl_Boxtype Fl_Widget::box() const
void Fl_Widget::box(Fl_Boxtype)

The `box()` identifies a routine that draws the background of the widget. See Box Types for the available types. The default depends on the widget, but is usually `FL_NO_BOX` or `FL_UP_BOX`.

Fl_Color Fl_Widget::color() const
void Fl_Widget::color(Fl_Color)

This color is passed to the box routine. Color is an index into an internal table of rgb colors. For most widgets this defaults to `FL_GRAY`. See the enumeration list for predefined colors. Use `Fl::set_color()` to redefine colors.

Fl_Color Fl_Widget::selection_color() const
void Fl_Widget::selection_color(Fl_Color)
void Fl_Widget::color(Fl_Color, Fl_Color)

For Forms compatibility a second color is defined. This is usually used to color the widget when it is selected, although some widgets use this color for other purposes. You can set both colors at once with `color(a,b)`.

const char* Fl_Widget::label() const
void Fl_Widget::label(const char*)

The label is printed somewhere on the widget or next to it. The string is *not* copied, the passed pointer is stored unchanged in the widget.

void Fl_Widget::label(Fl_Labeltype, const char*)
uchar Fl_Widget::labeltype() const
void Fl_Widget::labeltype(Fl_Labeltype)

A `labeltype` identifies a routine that draws the label of the widget. This can be used for special effects such as emboss, or to use the `label()` pointer as another form of data such as a bitmap. The value `FL_NORMAL_LABEL` prints the label as text.

Fl_Align Fl_Widget::align() const
void Fl_Widget::align(Fl_Align)

How the label is printed next to or inside the widget. The default value is `FL_ALIGN_CENTER`, which centers the label. The value can be any of these constants or'd together:

- `FL_ALIGN_CENTER`
- `FL_ALIGN_TOP`
- `FL_ALIGN_BOTTOM`
- `FL_ALIGN_LEFT`
- `FL_ALIGN_RIGHT`
- `FL_ALIGN_INSIDE`
- `FL_ALIGN_CLIP`
- `FL_ALIGN_WRAP`

Fl_Color Fl_Widget::labelcolor() const
void Fl_Widget::labelcolor(Fl_Color)

This color is passed to the labeltype routine, and is typically the color of the label text. This defaults to `FL_BLACK`.

Fl_Font Fl_Widget::labelfont() const
void Fl_Widget::labelfont(Fl_Font)

Fonts are identified by small 8-bit indexes into a table. See the enumeration list for predefined typefaces. The default value uses a Helvetica typeface (Arial for Microsoft® Windows®). The function `Fl::set_font()` can define new typefaces.

uchar Fl_Widget::labelsize() const
void Fl_Widget::labelsize(uchar)

Fonts are further identified by a point size. The default is 14.

typedef void (Fl_Callback)(Fl_Widget*, void*)
Fl_Callback* Fl_Widget::callback() const
void Fl_Widget::callback(Fl_Callback*, void* = 0)

Each widget has a single callback. You can set it or examine it with these methods.

void* Fl_Widget::user_data() const
void Fl_Widget::user_data(void*)

You can also just change the `void *` second argument to the callback with the `user_data` methods.

void Fl_Widget::callback(void (*)(Fl_Widget*, long), long = 0)
long Fl_Widget::argument() const
void Fl_Widget::argument(long)

For convenience you can also define the callback as taking a long argument. This is implemented by casting this to a `Fl_Callback` and casting the `long` to a `void *` and may not be portable to some machines.

void Fl_Widget::callback(void (*)(Fl_Widget*))

For convenience you can also define the callback as taking only one argument. This is implemented by casting this to a `Fl_Callback` and may not be portable to some machines.

void Fl_Widget::do_callback()
void Fl_Widget::do_callback(Fl_Widget*, void* = 0)
void Fl_Widget::do_callback(Fl_Widget*, long)

You can cause a widget to do its callback at any time, and even pass arbitrary arguments.

int Fl_Widget::changed() const
void Fl_Widget::set_changed()
void Fl_Widget::clear_changed()

`Fl_Widget::changed()` is a flag that is turned on when the user changes the value stored in the widget. This is only used by subclasses of `Fl_Widget` that store values, but is in the base class so it is easier to scan all the widgets in a panel and `do_callback()` on the changed ones in response to an "OK" button.

Most widgets turn this flag off when they do the callback, and when the program sets the stored value.

Fl_When Fl_Widget::when() const
void Fl_Widget::when(Fl_When)

`Fl_Widget::when()` is a set of bitflags used by subclasses of `Fl_Widget` to decide when to do the callback. If the value is zero then the callback is never done. Other values are described in the individual widgets. This field is in the base class so that you can scan a panel and `do_callback()` on all the ones that don't do their own callbacks in response to an "OK" button.

static void Fl_Widget::default_callback(Fl_Widget*, void*)

The default callback, which puts a pointer to the widget on the queue returned by `Fl::readqueue()`. You may want to call this from your own callback.

int Fl_Widget::visible() const
int Fl_Widget::visible_r() const
void Fl_Widget::show()
void Fl_Widget::hide()

An invisible widget never gets redrawn and does not get events. The `visible()` method returns true if the widget is set to be visible.The `visible_r()` method returns true if the widget and all of its parents are visible. A widget is only visible if `visible()` is true on it *and all of its parents*.

Changing it will send `FL_SHOW` or `FL_HIDE` events to the widget. *Do not change it if the parent is not visible, as this will send false `FL_SHOW` or `FL_HIDE` events to the widget*. `redraw()` is called if necessary on this or the parent.

int Fl_Widget::active() const
int Fl_Widget::active_r() constvoid Fl_Widget::activate()void Fl_Widget::deactivate()

`Fl_Widget::active()` returns whether the widget is active. `Fl_Widget::active_r()` returns whether the widget and all of its parents are active. An inactive widget does not get any events, but it does get redrawn. A widget is only active if `active()` is true on it *and all of its parents*.

Changing this value will send `FL_ACTIVATE` or `FL_DEACTIVATE` to the widget if `active_r()` is true.

Currently you cannot deactivate `Fl_Window` widgets.

int Fl_Widget::output() const
void Fl_Widget::set_output()void Fl_Widget::clear_output()

`output()` means the same as `!active()` except it does not change how the widget is drawn. The widget will not recieve any events. This is useful for making scrollbars or buttons that work as displays rather than input devices.

int Fl_Widget::takesevents() const

This is the same as `(active() && visible() && !output())` but is faster.

void Fl_Widget::redraw()

Mark the widget as needing its `draw()` routine called.

uchar Fl_Widget::damage() const

Non-zero if `draw()` needs to be called. Actually this is a bit field that the widget subclass can use to figure out what parts to draw.

Fl_Widget *Fl_Widget::parent() const

Returns a pointer to the parent widget. Usually this is a `Fl_Group` or `Fl_Window`. Returns `NULL` if none.

int Fl_Widget::contains(Fl_Widget* b) const

Returns true if `b` is a child of this widget, or is equal to this widget. Returns false if `b` is `NULL`.

int Fl_Widget::inside(const Fl_Widget* a) const

Returns true if this is a child of `a`, or is equal to `a` . Returns false if `a` is `NULL`.

int Fl_Widget::take_focus()

Tries to make this widget be the `Fl::focus()` widget, by first sending it an `FL_FOCUS` event, and if it returns non-zero, setting `Fl::focus()` to this widget. You should use this method to assign the focus to an widget. Returns true if the widget accepted the focus.

class Fl_Window

Class Hierarchy

```
Fl_Group
   |
   +----Fl_Window
           |
           +----Fl_Double_Window, Fl_Gl_Window,
                Fl_Overlay_Window, Fl_Single_Window
```

Include Files

```
#include <FL/Fl_Window.H>
```

Description

This widget produces an actual window. This can either be a main window, with a border and title and all the window management controls, or a "subwindow" inside a window. This is controlled by whether or not the window has a `parent()`.

Once you create a window, you usually add children `Fl_Widget` 's to it by using `window->add(child)` for each new widget. See `Fl_Group` for more information on how to add and remove children.

There are several subclasses of `Fl_Window` that provide double-buffering, overlay, menu, and OpenGL support.

The window's callback is done if the user tries to close a window using the window manager and `Fl::modal()` is zero or equal to the window. `Fl_Window` has a default callback that calls `Fl_Window::hide()` and calls `exit(0)` if this is the last top-level window.

Methods

• Fl_Window	• first_window	• hotspot	• modal	• set_non_modal
• ~Fl_Window	• free_position	• iconize	• next_window	• show
• border	• fullscreen	• iconlabel	• non_modal	• shown
• clear_border	• fullscreen_off	• label	• resize	• size_range
• current	• hide	• make_current	• set_modal	• xclass
• cursor				

Fl_Window::Fl_Window(int w, int h, const char *title = 0)

The first form of the constructor should be used for a "top-level" window (that is, one that is not inside another window). It correctly sets `visible()` to false and `parent()` to `NULL`. By not specifying the position of the window, the window system will pick a place to show the window or allow the user to pick a location. If you want to force a position you should call `position(x,y)` or `hotspot()` before calling `show()`.

`Fl_Widget::box()` is set to `FL_FLAT_BOX`. If you plan to completely fill the window with children widgets you should change this to `FL_NO_BOX`. If you turn the window border off you may want to change this to `FL_UP_BOX`.

Fl_Window::Fl_Window(int x, int y, int w, int h, const char *title = 0)

The second form of the constructor is for creating child windows. It leaves `visible()` set to true.

virtual Fl_Window::~Fl_Window()

The destructor *also deletes all the children*. This allows a whole tree to be deleted at once, without having to keep a pointer to all the children in the user code. A kludge has been done so the `Fl_Window` and all of it's children can be automatic (local) variables, but you must declare the `Fl_Window`*first* so that it is destroyed last.

void Fl_Window::size_range(int minw, int minh, int maxw=0, int maxh=0, int dw=0, int dh=0, int aspect=0)

Set the allowable range the user can resize this window to. This only works for top-level windows.

- `minw` and `minh` are the smallest the window can be.
- `maxw` and `maxh` are the largest the window can be. If either is *equal* to the minimum then you cannot resize in that direction. If either is zero then FLTK picks a maximum size in that direction such that the window will fill the screen.
- `dw` and `dh` are size increments. The window will be constrained to widths of `minw + N * dw`, where `N` is any non-negative integer. If these are less or equal to 1 they are ignored. (this is ignored on WIN32)
- `aspect` is a flag that indicates that the window should preserve it's aspect ratio. This only works if both the maximum and minimum have the same aspect ratio. (ignored on WIN32 and by many X window managers)

If this function is not called, FLTK tries to figure out the range from the setting of `resizeable()` :

- If `resizeable()` is `NULL` (this is the default) then the window cannot be resized and the resize border and max-size control will not be displayed for the window.
- If either dimension of `resizeable()` is less than 100, then that is considered the minimum size. Otherwise the `resizeable()` has a minimum size of 100.
- If either dimension of `resizeable()` is zero, then that is also the maximum size (so the window cannot resize in that direction).

It is undefined what happens if the current size does not fit in the constraints passed to `size_range()`.

virtual void Fl_Window::show()
int Fl_Window::show(int argc, char **argv, int i)
void Fl_Window::show(int argc, char **argv)

Put the window on the screen. Usually this has the side effect of opening the display. The second two forms are used for top-level windows and allow standard arguments to be parsed from the command-line.

If the window is already shown then it is restored and raised to the top. This is really convenient because your program can call `show()` at any time, even if the window is already up. It also means that `show()` serves the purpose of `raise()` in other toolkits.

virtual void Fl_Window::hide()

Remove the window from the screen. If the window is already hidden or has not been shown then this does nothing and is harmless.

int Fl_Window::shown() const

Returns non-zero if `show()` has been called (but not `hide()`). You can tell if a window is iconified with `(w->shown() &!w->visible())`.

void Fl_Window::iconize()

Iconifies the window. If you call this when `shown()` is false it will `show()` it as an icon. If the window is already iconified this does nothing.

Call `show()` to restore the window.

When a window is iconified/restored (either by these calls or by the user) the `handle()` method is called with `FL_HIDE` and `FL_SHOW` events and `visible()` is turned on and off.

There is no way to control what is drawn in the icon except with the string passed to `Fl_Window::xclass()`. You should not rely on window managers displaying the icons.

Fl_Window *Fl::first_window()

Returns the first `shown()` window in the widget hierarchy. If no windows are displayed `first_window` returns `NULL`.

Fl_Window *Fl::next_window(const Fl_Window*)

Returns the next `shown()` window in the hierarchy. You can use this call to iterate through all the windows that are shown().

void Fl_Window::resize(int,int,int,int)

Change the size and position of the window. If `shown()` is true, these changes are communicated to the window server (which may refuse that size and cause a further resize). If `shown()` is false, the size and position are used when `show()` is called. See `Fl_Group` for the effect of resizing on the child widgets.

You can also call the `Fl_Widget` methods `size(x,y)` and `position(w,h)`, which are inline wrappers for this virtual function.

void Fl_Window::free_position()

Undoes the effect of a previous `resize()` or `show()` so that the next time `show()` is called the window manager is free to position the window.

void Fl_Window::hotspot(int x, int y, int offscreen = 0)
void Fl_Window::hotspot(const Fl_Widget*, int offscreen = 0)
void Fl_Window::hotspot(const Fl_Widgetp, int offscreen = 0)

`position()` the window so that the mouse is pointing at the given position, or at the center of the given widget, which may be the window itself. If the optional `offscreen` parameter is non-zero, then the window is allowed to extend off the screen (this does not work with some X window managers).

void Fl_Window::fullscreen()

Makes the window completely fill the screen, without any window manager border visible. You must use `fullscreen_off()` to undo this. This may not work with all window managers.

int Fl_Window::fullscreen_off(int x, int y, int w, int h)

Turns off any side effects of `fullscreen()` and does `resize(x,y,w,h)`.

int Fl_Window::border(int)
uchar Fl_Window::border() const

Gets or sets whether or not the window manager border is around the window. The default value is true. `border(n)` can be used to turn the border on and off, and returns non-zero if the value has been changed. *Under most X window managers this does not work after `show()` has been called, although SGI's 4DWM does work.*

void Fl_Window::clear_border()

`clear_border()` is a fast inline function to turn the border off. It only works before `show()` is called.

void Fl_Window::set_modal()

A "modal" window, when `shown()`, will prevent any events from being delivered to other windows in the same program, and will also remain on top of the other windows (if the X window manager supports the "transient for" property). Several modal windows may be shown at once, in which case only the last one shown gets events. You can see which window (if any) is modal by calling `Fl::modal()`.

uchar Fl_Window::modal() const

Returns true if this window is modal.

void Fl_Window::set_non_modal()

A "non-modal" window (terminology borrowed from Microsoft Windows) acts like a `modal()` one in that it remains on top, but it has no effect on event delivery. There are *three* states for a window: modal, non-modal, and normal.

uchar Fl_Window::non_modal() const

Returns true if this window is modal or non-modal.

void Fl_Window::label(const char*)
const char* Fl_Window::label() const

Gets or sets the window title bar label.

void Fl_Window::iconlabel(const char*)
const char* Fl_Window::iconlabel() const

Gets or sets the icon label.

void Fl_Window::xclass(const char*)
const char* Fl_Window::xclass() const

A string used to tell the system what type of window this is. Mostly this identifies the picture to draw in the icon. *Under X, this is turned into a `XA_WM_CLASS` pair by truncating at the first non-alphanumeric character and capitalizing the first character, and the second one if the first is 'x'. Thus "foo" turns into "foo, Foo", and "xprog.1" turns into "xprog, XProg".* This only works if called *before* calling `show()`.

Under Microsoft Windows this string is used as the name of the WNDCLASS structure, though it is not clear if this can have any visible effect.

void Fl_Window::make_current()

`make_current()` sets things up so that the drawing functions in `<FL/fl_draw.H>` will go into this window. This is useful for incremental update of windows, such as in an idle callback, which will make your program behave much better if it draws a slow graphic. **Danger: incremental update is very hard to debug and maintain!**

This method only works for the `Fl_Window` and `Fl_Gl_Window` classes.

static Fl_Window* Fl_Window::current()

Returns the last window that was made current.

void Fl_Window::cursor(Fl_Cursor, Fl_Color = FL_WHITE, Fl_Color = FL_BLACK)

Change the cursor for this window. This always calls the system, if you are changing the cursor a lot you may want to keep track of how you set it in a static varaible and call this only if the new cursor is different.

The type `Fl_Cursor` is an enumeration defined in `<Enumerations.H>`. (Under X you can get any XC_cursor value by passing `Fl_Cursor((XC_foo/2)+1)`). The colors only work on X, they are not implemented on WIN32.

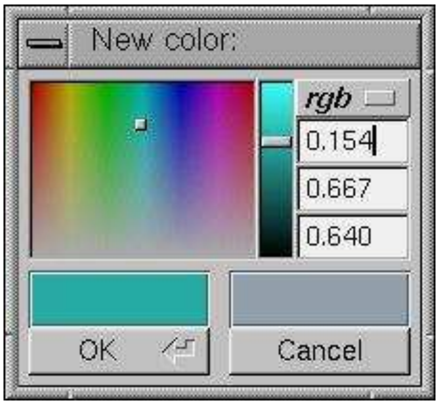
B - Function Reference

This appendix describes all of the `fl_` functions and `Fl::` methods. For a description of the FLTK widgets, see Appendix A.

Functions

int fl_color_chooser(const char*, double &r, double &g, double &b)
int fl_color_chooser(const char *, uchar &r, uchar &g, uchar &b)

The double version takes RGB values in the range 0.0 to 1.0. The uchar version takes RGB values in the range 0 to 255.



`fl_color_chooser()` pops up a window to let the user pick an arbitrary RGB color. They can pick the hue and saturation in the "hue box" on the left (hold down CTRL to just change the saturation), and the brighness using the vertical slider. Or they can type the 8-bit numbers into the RGB `Fl_Value_Input` fields, or drag the mouse across them to adjust them. The pull-down menu lets the user set the input fields to show RGB, HSV, or 8-bit RGB (0 to 255).

This returns non-zero if the user picks ok, and updates the RGB values. If the user picks cancel or closes the window this returns zero and leaves RGB unchanged.

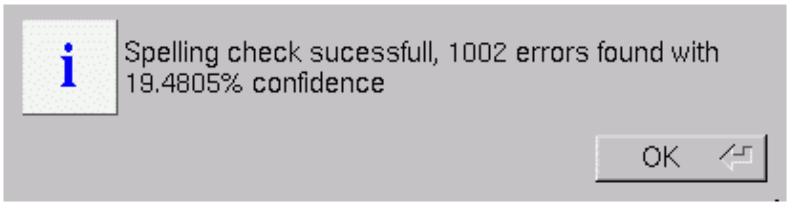
If you use the color chooser on an 8-bit screen, it will allocate all the available colors, leaving you no space to exactly represent the color the user picks! You can however use `fl_rectf()` to fill a region with a simulated color using dithering.

int fl_show_colormap(int oldcol)

`fl_show_colormap()` pops up a panel of the 256 colors you can access with `fl_color()` and lets the user pick one of them. It returns the new color index, or the old one if the user types ESC or clicks outside the window.

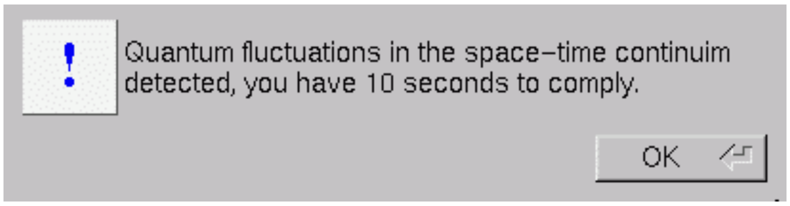
void fl_message(const char *, ...)

Displays a printf-style message in a pop-up box with an "OK" button, waits for the user to hit the button. The message will wrap to fit the window, or may be many lines by putting `\n` characters into it. The enter key is a shortcut for the OK button.



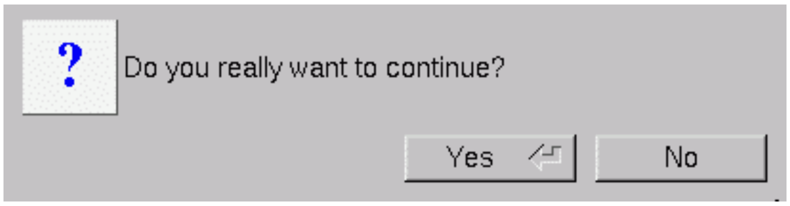
void fl_alert(const char *, ...)

Same as `fl_message()` except for the "!" symbol.



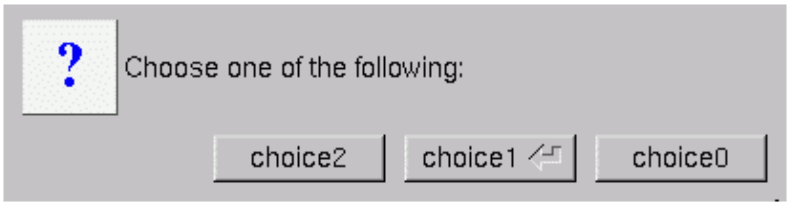
int fl_ask(const char *, ...)

Displays a printf-style message in a pop-up box with an "Yes" and "No" button and waits for the user to hit a button. The return value is 1 if the user hits Yes, 0 if they pick No. The enter key is a shortcut for Yes and ESC is a shortcut for No.



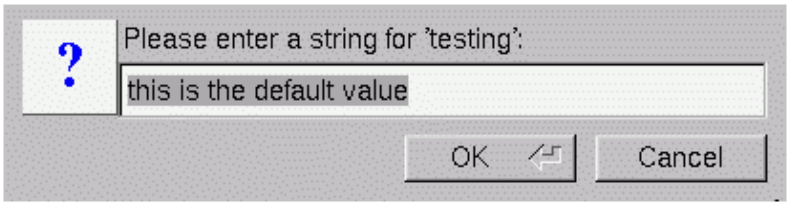
int fl_choice(const char *q, const char *b0, const char *b1, const char *b2, ...)

Shows the message with three buttons below it marked with the strings `b0`, `b1`, and `b2`. Returns 0, 1, or 2 depending on which button is hit. ESC is a shortcut for button 0 and the enter key is a shortcut for button 1. Notice the buttons are positioned "backwards" You can hide buttons by passing `NULL` as their labels.



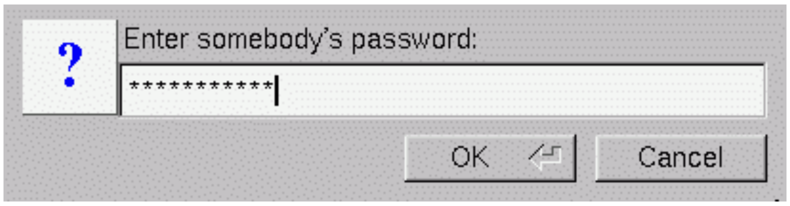
const char *fl_input(const char *label, const char *deflt = 0, ...)

Pops up a window displaying a string, lets the user edit it, and return the new value. The cancel button returns `NULL`. *The returned pointer is only valid until the next time `fl_input()` is called.* Due to back-compatability, the arguments to any printf commands in the label are after the default value.



const char *fl_password(const char *label, const char *deflt = 0, ...)

Same as `fl_input()` except an `Fl_Secret_Input` field is used.



void fl_message_font(Fl_Font fontid, uchar size)

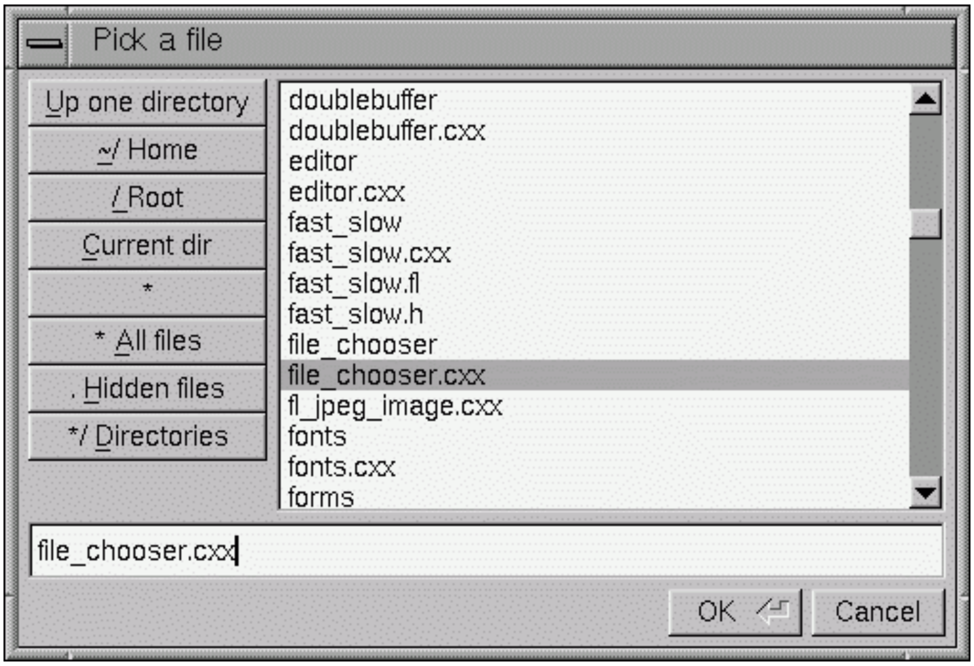
Change the font and font size used for the messages in all the popups.

Fl_Widget *fl_message_icon()

Returns a pointer to the box at the left edge of all the popups. You can alter the font, color, or label (including making it a Pixmap), before calling the functions.

char *fl_file_chooser(const char * message, const char *pattern, const char *fname)

FLTK provides a "tab completion" file chooser that makes it easy to choose files from large directories. This file chooser has several unique features, the major one being that the Tab key completes filenames like it does in Emacs or tcsh, and the list always shows all possible completions.



`fl_file_chooser()` pops up the file chooser, waits for the user to pick a file or Cancel, and then returns a pointer to that filename or `NULL` if Cancel is chosen.

`message` is a string used to title the window.

`pattern` is used to limit the files listed in a directory to those matching the pattern. This matching is done by `filename_match()`. Use `NULL` to show all files.

`fname` is a default filename to fill in the chooser with. If this is `NULL` then the last filename that was choosen is used (unless that had a different pattern, in which case just the last directory with no name is used). The first time the file chooser is called this defaults to a blank string.

The returned value points at a static buffer that is only good until the next time `fl_file_chooser()` is called.

void fl_file_chooser_callback(void (*cb)(const char *))

Set a function that is called every time the user clicks a file in the currently popped-up file chooser. This could be used to preview the contents of the file. It has to be reasonably fast, and cannot create FLTK windows.

int filename_list(const char *d, dirent ***list)

This is a portable and const-correct wrapper for the `fl_scandir` function. `d` is the name of a directory (it does not matter if it has a trailing slash or not). For each file in that directory a "dirent" structure is created. The only portable thing about a dirent is that dirent.d_name is the nul-terminated file name. An array of pointers to these dirents is created and a pointer to the array is returned in `*list`. The number of entries is given as a return value. If there is an error reading the directory a number less than zero is returned, and `errno` has the reason (`errno` does not work under WIN32). The files are sorted in "alphanumeric" order, where an attempt is made to put unpadded numbers in consecutive order.

You can free the returned list of files with the following code:

```
for (int i = return_value; i > 0;) free((void*)(list[--i]));
free((void*)list);
```

int filename_isdir(const char *f)

Returns non-zero if the file exists and is a directory.

const char *filename_name(const char *f)

Returns a pointer to the character after the last slash, or to the start of the filename if there is none.

const char *filename_ext(const char *f)

Returns a pointer to the last period in `filename_name(f)`, or a pointer to the trailing nul if none.

char *filename_setext(char *f, const char *ext)

Does `strcpy(filename_ext(f), ext ? ext : "")`. Returns a pointer to `f`.

int filename_expand(char *out, const char *in)

Splits `in` at each slash character. Replaces any occurrance of `$X` with `getenv("X")` (leaving it as `$X` if the environment variable does not exist). Replaces any occurances of `~X` with user `X`'s home directory (leaving it as `~X` if the user does not exist). Any resulting double slashes cause everything before the second slash to be deleted. Copies the result to `out` (`in` and `out` may be the same buffer). Returns non-zero if any changes were made. *In true retro programming style, it is up to you to provide a buffer big enough for the result. 1024 characters should be enough.*

int filename_absolute(char *out, const char *in)

If `in` does not start with a slash, this prepends the current working directory to `in` and then deletes any occurances of **.** and x/.. from the result, which it copies to `out` (`in` and `out` may be the same buffer). Returns non-zero if any changes were made. *In true retro programming style, it is up to you to provide a buffer big enough for the result. 1024 characters should be enough.*

int filename_match(const char *f, const char *pattern)

Returns true if `f` matches `pattern`. The following syntax is used by `pattern`:

- `*` matches any sequence of 0 or more characters.
- `?` matches any single character.
- `[set]` matches any character in the set. Set can contain any single characters, or a-z to represent a range. To match] or - they must be the first characters. To match ^ or ! they must not be the first characters.
- `[^set] or` **`[!set]`** matches any character not in the set.
- `{X|Y|Z} or` **`{X,Y,Z}`** matches any one of the subexpressions literally.

- `\x` quotes the character x so it has no special meaning.
- `x` all other characters must be matched exactly.

Fl:: Methods

static void Fl::add_fd(int fd, void (*cb)(int, void *), void * = 0)
static void Fl::add_fd(int fd, int when, void (*cb)(int, void *), void * = 0)
static void Fl::remove_fd(int)

Add file descriptor `fd` to listen to. When the `fd` becomes ready for reading the callback is done. The callback is passed the `fd` and the arbitrary `void *` argument. `Fl::wait()` will return immediately after calling the callback.

The second version takes a `when` bitfield, with the bits `FL_READ`, `FL_WRITE`, and `FL_EXCEPT` defined, to indicate when the callback should be done.

There can only be one callback of each type for a file descriptor. `Fl::remove_fd()` gets rid of *all* the callbacks for a given file descriptor.

Under UNIX *any* file descriptor can be monitored (files, devices, pipes, sockets, etc.) Due to limitations in Microsoft Windows, WIN32 applications can only monitor sockets.

static void Fl::add_handler(int (*f)(int))

Install a function to parse unrecognized events. If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non zero then the event is ignored. Events that cause this to be called are:

- `FL_SHORTCUT` events that are not recognized by any widget. This lets you provide global shortcut keys.
- System events that FLTK does not recognize. See `fl_xevent`.
- *Some* other events when the widget FLTK selected returns zero from its `handle()` method. Exactly which ones may change in future versions, however.

static Fl::add_idle(void (*cb)(void *), void *)

Adds a callback function that is called by `Fl::wait()` when there is nothing to do. This can be used for background processing.

Warning: this can absorb all your machine's time!

You can have multiple idle callbacks. To remove an idle callback use `Fl::remove_idle()`.

`Fl::wait()` and `Fl::check()` call idle callbacks, but `Fl::ready()` does not.

The idle callback can call any FLTK functions. However if you call something that calls `Fl::wait()` or `Fl::check()` (such as a message pop-up) you should first remove the idle callback so that it does not recurse.

static void Fl::add_timeout(float t, void (*cb)(void *),void *v=0)

Add a one-shot timeout callback. The timeout will happen as soon as possible after `t` seconds after the last time `wait()` was called. The optional `void *` argument is passed to the callback.

This code will print "TICK" each second on stdout, no matter what else the user or program does:

```
void callback(void *) {
  printf("TICK\n");
  Fl::add_timeout(1.0,callback);
}

main() {
  Fl::add_timeout(1.0,callback);
  Fl::run();
}
```

static int Fl::arg(int argc, char **argv, int &i)

Consume a single switch from `argv`, starting at word i. Returns the number of words eaten (1 or 2, or 0 if it is not recognized) and adds the same value to `i`. You can use this function if you prefer to control the incrementing through the arguments yourself.

static int Fl::args(int argc, char **argv, int &i, int (*callback)(int, char**,int &)=0)
void Fl::args(int argc, char **argv)

FLTK provides an *entirely optional* command-line switch parser. You don't have to call it if you don't like them! Everything it can do can be done with other calls to FLTK.

To use the switch parser, call `Fl::args(...)` near the start of your program. This does *not* open the display, instead switches that need the display open are stashed into static variables. Then you *must* display your first window by calling window->show(argc,argv), which will do anything stored in the static variables.

`callback` lets you define your own switches. It is called with the same `argc` and `argv`, and with `i` the index of each word. The callback should return zero if the switch is unrecognized, and not change `i`. It should return non-zero if the switch is recognized, and add at least 1 to `i` (it can add more to consume words after the switch). This function is called *before* any other tests, so *you can override any FLTK switch* (this is why fltk can use very short switches instead of the long ones all other toolkits force you to use).

On return `i` is set to the index of the first non-switch. This is either:

- The first word that does not start with '-'.
- The word '-' (used by many programs to name stdin as a file)
- The first unrecognized switch (return value is 0).
- `argc`

The return value is `i` unless an unrecognized switch is found, in which case it is zero. If your program takes no arguments other than switches you should produce an error if the return value is less than `argc`.

All switches except -bg2 may be abbreviated one letter and case is ignored:

- `-display host:n.n` The X display to use (ignored under WIN32).
- `-geometry WxH+X+Y` The window position and size will be modified according the the standard X geometry string.
- `-name string` Fl_Window::xclass(string) will be done to the window, possibly changing its icon.
- `-title string` Fl_Window::label(string) will be done to the window, changing both its title and the icontitle.
- `-iconic` Fl_Window::iconize() will be done to the window.
- `-bg color` XParseColor is used to lookup the passed color and then Fl::background() is done. Under WIN32 only color names of the form "#xxxxxx" are understood.
- `-bg2 color` XParseColor is used to lookup the passed color and then Fl::background2() is done.
- `-fg color` XParseColor is used to lookup the passed color and then Fl::foreground() is done.

The second form of `Fl::args()` is useful if your program does not have command line switches of its own. It parses all the switches, and if any are not recognized it calls `Fl::abort(Fl::help)`.

static void Fl::background(uchar, uchar, uchar)

Changes `fl_color(FL_GRAY)` to the given color, and changes the gray ramp from 32 to 56 to black to white. These are the colors used as backgrounds by almost all widgets and used to draw the edges of all the boxtypes.

static void Fl::background2(uchar, uchar, uchar)

Changes `fl_color(FL_WHITE)` and the same colors as `Fl::foreground()`. This color is used as a background by `Fl_Input` and other text widgets.

static Fl_Widget *Fl::belowmouse() const
static void Fl::belowmouse(Fl_Widget *)

Get or set the widget that is below the mouse. This is for highlighting buttons. It is not used to send `FL_PUSH` or `FL_MOVE` directly, for several obscure reasons, but those events typically go to this widget. This is also the first widget tried for `FL_SHORTCUT` events.

If you change the belowmouse widget, the previous one and all parents (that don't contain the new widget) are sent `FL_LEAVE` events. Changing this does *not* send `FL_ENTER` to this or any widget, because sending `FL_ENTER` is supposed to *test* if the widget wants the mouse (by it returning non-zero from `handle()`).

static int Fl::box_dh(Fl_Boxtype)

Returns the height offset for the given boxtype.

static int Fl::box_dw(Fl_Boxtype)

Returns the width offset for the given boxtype.

static int Fl::box_dx(Fl_Boxtype)

Returns the X offset for the given boxtype.

static int Fl::box_dy(Fl_Boxtype)

Returns the Y offset for the given boxtype.

static int Fl::check()

This does the same thing as `Fl::wait(0)`, except because it does not have to return the elapsed time value it can be implemented faster on certain systems. Use this to interrupt a big calculation:

```
while (!calculation_done()) {
  calculate();
  Fl::check();
  if (user_hit_abort_button()) break;
}
```

This returns non-zero if any windows are displayed, and 0 if no windows are displayed.

static int Fl::damage()

If true then `flush()` will do something.

static void Fl::display(const char *)

Sets the X display to use for all windows. Actually this just sets the environment variable $DISPLAY to the passed string, so this only works before you show() the first window or otherwise open the display, and does nothing useful under WIN32.

static void Fl::enable_symbols()

Enables the symbol drawing code.

static int Fl::event_button()

Returns which mouse button was pressed. This returns garbage if the most recent event was not a `FL_PUSH` or `FL_RELEASE` event.

int Fl::event_clicks()
void Fl::event_clicks(int)

The first form returns non-zero if the most recent `FL_PUSH` or `FL_KEYBOARD` was a "double click". Returns N-1 for N clicks. A double click is counted if the same button is pressed again while `event_is_click()` is true.

The second form directly sets the number returned by `Fl::event_clicks()`. This can be used to set it to zero so that later code does not think an item was double-clicked.

int Fl::event_inside(const Fl_Widget *) const
int Fl::event_inside(int x, int y, int w, int h)

Returns non-zero if the current `event_x` and `event_y` put it inside the widget or inside an arbitrary bounding box. You should always call this rather than doing your own comparison so you are consistent about edge effects.

int Fl::event_is_click()
void Fl::event_is_click(0)

The first form returns non-zero if the mouse has not moved far enough and not enough time has passed since the last `FL_PUSH` or `FL_KEYBOARD` event for it to be considered a "drag" rather than a "click". You can test this on `FL_DRAG`, `FL_RELEASE`, and `FL_MOVE` events. The second form clears the value returned by `Fl::event_is_click()`. Useful to prevent the *next* click from being counted as a double-click or to make a popup menu pick an item with a single click. Don't pass non-zero to this.

int Fl::event_key()
int Fl::event_key(int)
int Fl::get_key(int)

`Fl::event_key()` returns which key on the keyboard was last pushed.

`Fl::event_key(int)` returns true if the given key was held down (or pressed) *during* the last event. This is constant until the next event is read from the server.

`Fl::get_key(int)` returns true if the given key is held down *now*. Under X this requires a round-trip to the server and is *much* slower than `Fl::event_key(int)`.

Keys are identified by the *unshifted* values. FLTK defines a set of symbols that should work on most modern machines for every key on the keyboard:

- All keys on the main keyboard producing a printable ASCII character use the value of that ASCII character (as though shift, ctrl, and caps lock were not on). The space bar is 32.
- All keys on the numeric keypad producing a printable ASCII character use the value of that ASCII character plus `FL_KP`. The highest possible value is `FL_KP_Last` so you can range-check to see if something is on the keypad.
- All numbered function keys use the number on the function key plus `FL_F`. The highest possible number is `FL_F_Last`, so you can range-check a value.

- Buttons on the mouse are considered keys, and use the button number (where the left button is 1) plus `FL_Button`.
- All other keys on the keypad have a symbol: `FL_Escape, FL_BackSpace, FL_Tab, FL_Enter, FL_Print, FL_Scroll_Lock, FL_Pause, FL_Insert, FL_Home, FL_Page_Up, FL_Delete, FL_End, FL_Page_Down, FL_Left, FL_Up, FL_Right, FL_Down, FL_Shift_L, FL_Shift_R, FL_Control_L, FL_Control_R, FL_Caps_Lock, FL_Alt_L, FL_Alt_R, FL_Meta_L, FL_Meta_R, FL_Menu, FL_Num_Lock, FL_KP_Enter`. Be careful not to confuse these with the very similar, but all-caps, symbols used by `Fl::event_state()`.

On X `Fl::get_key(FL_Button+n)` does not work.

On WIN32 `Fl::get_key(FL_KP_Enter)` and `Fl::event_key(FL_KP_Enter)` do not work.

char *Fl::event_length()

Returns the length of the text in `Fl::event_text()`. There will always be a nul at this position in the text. However there may be a nul before that if the keystroke translates to a nul character or you paste a nul character.

ulong Fl::event_state()
unsigned int Fl::event_state(ulong)

This is a bitfield of what shift states were on and what mouse buttons were held down during the most recent event. The second version returns non-zero if any of the passed bits are turned on. The legal bits are:

- `FL_SHIFT`
- `FL_CAPS_LOCK`
- `FL_CTRL`
- `FL_ALT`
- `FL_NUM_LOCK`
- `FL_META`
- `FL_SCROLL_LOCK`
- `FL_BUTTON1`
- `FL_BUTTON2`
- `FL_BUTTON3`

X servers do not agree on shift states, and FL_NUM_LOCK, FL_META, and FL_SCROLL_LOCK may not work. The values were selected to match the XFree86 server on Linux. In addition there is a bug in the way X works so that the shift state is not correctly reported until the first event *after* the shift key is pressed or released.

char *Fl::event_text()

Returns the ASCII text (in the future this may be UTF-8) produced by the last `FL_KEYBOARD` or `FL_PASTEM` or possibly other event. A zero-length string is returned for any keyboard function keys that do not produce text. This pointer points at a static buffer and is only valid until the next event is processed.

Under X this is the result of calling `XLookupString()`.

static int Fl::event_x()
static int Fl::event_y()

Returns the mouse position of the event relative to the `Fl_Window` it was passed to.

static int Fl::event_x_root()
static int Fl::event_y_root()

Returns the mouse position on the screen of the event. To find the absolute position of an `Fl_Window` on the screen, use the difference between `event_x_root(),event_y_root()` and `event_x(),event_y()`.

static Fl_Window *Fl::first_window()

Returns the first top-level window in the widget hierarchy.

static void Fl::flush()

Causes all the windows that need it to be redrawn and graphics forced out through the pipes. This is what `wait()` does before looking for events.

static Fl_Widget *Fl::focus() const
static void Fl::focus(Fl_Widget *)

Get or set the widget that will receive `FL_KEYBOARD` events.

If you change `Fl::focus()`, the previous widget and all parents (that don't contain the new widget) are sent `FL_UNFOCUS` events. Changing the focus does *not* send `FL_FOCUS` to this or any widget, because sending `FL_FOCUS` is supposed to *test* if the widget wants the focus (by it returning non-zero from `handle()`).

static void Fl::foreground(uchar, uchar, uchar)

Changes `fl_color(FL_BLACK)`. Also changes `FL_INACTIVE_COLOR` and `FL_SELECTION_COLOR` to be a ramp between this and `FL_WHITE`.

static void Fl::free_color(Fl_Color, int overlay = 0)

Frees the specified color from the colormap, if applicable. If `overlay` is non-zero then the color is freed from the overlay colormap.

static unsigned Fl::get_color(Fl_Color)
static void Fl::get_color(Fl_Color, uchar &r, uchar &g, uchar &b)

Returns the color index or RGB value for the given FLTK color index.

static const char *Fl::get_font(int face)

Get the string for this face. This string is different for each face. Under X this value is passed to XListFonts to get all the sizes of this face.

static const char *Fl::get_font_name(int face, int *attributes = 0)

Get a human-readable string describing the family of this face. This is useful if you are presenting a choice to the user. There is no guarantee that each face has a different name. The return value points to a static buffer that is overwritten each call.

The integer pointed to by `attributes` (if the pointer is not zero) is set to zero, `FL_BOLD` or `FL_ITALIC` or `FL_BOLD | FL_ITALIC`. To locate a "family" of fonts, search forward and back for a set with non-zero attributes, these faces along with the face with a zero attribute before them constitute a family.

int get_font_sizes(int face, int *&sizep)

Return an array of sizes in `sizep`. The return value is the length of this array. The sizes are sorted from smallest to largest and indicate what sizes can be given to `fl_font()` that will be matched exactly (`fl_font()` will pick the closest size for other sizes). A zero in the first location of the array indicates a scalable font, where any size works, although the array may list sizes that work "better" than others. Warning: the returned array points at a static buffer that is overwritten each call. Under X this will open the display.

static void Fl::get_mouse(int &x, int &y)

Return where the mouse is on the screen by doing a round-trip query to the server. You should use `Fl::event_x_root()` and `Fl::event_y_root()` if possible, but this is necessary if you are not sure if a mouse event has been processed recently (such as to position your first window). If the display is not open, this will open it.

static void Fl::get_system_colors()

Read the user preference colors from the system and use them to call `Fl::foreground()`, `Fl::background()`, and `Fl::background2()`. This is done by `Fl_Window::show(argc,argv)` before applying the -fg and -bg switches.

Currently this only does something on WIN32. In future versions for X it may read the window manager (KDE, Gnome, etc.) setup as well.

static int Fl::gl_visual(int)

This does the same thing as `Fl::visual(int)` but also requires OpenGL drawing to work. This *must* be done if you want to draw in normal windows with OpenGL with `gl_start()` and `gl_end()`. It may be useful to call this so your X windows use the same visual as an `Fl_Gl_Window`, which on some servers will reduce colormap flashing.

See `Fl_Gl_Window` for a list of additional values for the argument.

static void Fl::grab(Fl_Window*)
static Fl_Window* Fl::grab()

This is used when pop-up menu systems are active. Send all events to the passed window no matter where the pointer or focus is (including in other programs). The window *does not have to be `shown()`*, this lets the `handle()` method of a "dummy" window override all event handling and allows you to map and unmap a complex set of windows (under both X and WIN32 *some* window must be mapped because the system interface needs a window id).

If `grab()` is on it will also affect show() of windows by doing system-specific operations (on X it turns on override-redirect). These are designed to make menus popup reliably and faster on the system.

To turn off grabbing do `Fl::grab(0)`.

Be careful that your program does not enter an infinite loop while `grab()` is on. On X this will lock up your screen!

static int Fl::h()

Returns the height of the screen in pixels.

static int Fl::handle(int, Fl_Window *)

Sends the event to a window for processing. Returns non-zero if any widget uses the event.

static const char *Fl::help

This is the usage string that is displayed if `Fl::args()` detects an invalid argument on the command-line.

static Fl_Window *Fl::modal()

Returns the top-most `modal()` window currently shown. This is the most recently `shown()` window with `modal()` true, or `NULL` if there are no `modal()` windows `shown()`. The `modal()` window has its `handle()` method called for all events, and no other windows will have `handle()` called (`grab()` overrides this).

static Fl_Window *Fl::next_window(Fl_Window *)

Returns the next top-level window in the widget hierarchy.

static void Fl::own_colormap()

Makes FLTK use its own colormap. This may make FLTK display better and will reduce conflicts with other programs that want lots of colors. However the colors may flash as you move the cursor between windows.

This does nothing if the current visual is not colormapped.

static void Fl::paste(Fl_Widget *receiver)

Set things up so the receiver widget will be called with an `FL_PASTE` event some time in the future. The reciever should be prepared to be called *directly* by this, or for it to happen *later*, or possibly *not at all*. This allows the window system to take as long as necessary to retrieve the paste buffer (or even to screw up completely) without complex and error-prone synchronization code in FLTK.

static Fl_Widget *Fl::pushed() const
static void Fl::pushed(Fl_Widget *)

Get or set the widget that is being pushed. `FL_DRAG` or `FL_RELEASE` (and any more `FL_PUSH`) events will be sent to this widget.

If you change the pushed widget, the previous one and all parents (that don't contain the new widget) are sent `FL_RELEASE` events. Changing this does *not* send `FL_PUSH` to this or any widget, because sending `FL_PUSH` is supposed to *test* if the widget wants the mouse (by it returning non-zero from `handle()`).

static Fl_Widget *Fl::readqueue()

All `Fl_Widgets` that don't have a callback defined use a default callback that puts a pointer to the widget in this queue, and this method reads the oldest widget out of this queue.

static int Fl::ready()

Returns non-zero if there are pending timeouts or events or file descriptors. This does *not* call `Fl::flush()` or any callbacks, which is useful if your program is in a state where such callbacks are illegal:

```
while (!calculation_done()) {
  calculate();
  if (Fl::ready()) {
    do_expensive_cleanup();
    Fl::check();
    if (user_hit_abort_button()) break;
  }
}
```

static void Fl::redraw()

Redraws all widgets.

static void Fl::remove_idle(void (*cb)(void *), void *= 0)

Removes the specified idle callback.

static void Fl::remove_timeout(void (*cb)(void *), void *= 0)

Removes a timeout callback. It is harmless to remove a timeout callback that no longer exists.

static Fl::run()

Runs FLTK until there are no windows displayed, and then returns a zero. `Fl::run()` is *exactly equivalent to:*

```
while (Fl::wait());
return 0;
```

static void Fl::selection(Fl_Widget *owner, const char *stuff, int len)
static const char* Fl::selection()
static int Fl::selection_length()

The first form changes the current selection. The block of text is copied to an internal buffer by FLTK (be careful if doing this in response to an `FL_PASTE` as this *may* be the same buffer returned by `event_text()`). The `selection_owner()` widget is set to the passed owner (possibly sending `FL_SELECTIONCLEAR` to the previous owner). The second form looks at the buffer containing the current selection. The contents of this buffer are undefined if this program does not own the current selection.

static Fl_Widget *Fl::selection_owner() const
static void Fl::selection_owner(Fl_Widget *)

The single-argument `selection_owner(x)` call can be used to move the selection to another widget or to set the owner to `NULL` , without changing the actual text of the selection. `FL_SELECTIONCLEAR` is sent to the previous selection owner, if any.

Copying the buffer every time the selection is changed is obviously wasteful, especially for large selections. An interface will probably be added in a future version to allow the selection to be made by a callback function. The current interface will be emulated on top of this.

static void Fl::set_boxtype(Fl_Boxtype, Fl_Box_Draw_F *, uchar, uchar, uchar, uchar)
static void Fl::set_boxtype(Fl_Boxtype, Fl_Boxtype from)

The first form sets the function to call to draw a specific boxtype.

The second form copies the `from` boxtype.

static void Fl::set_color(Fl_Color, uchar r, uchar g, uchar b)

Sets an entry in the `fl_color` index table. You can set it to any 8-bit RGB color. The color is not allocated until `fl_color(i)` is used.

static int Fl::set_font(int face, const char *)
static int Fl::set_font(int face, int from)

The first form changes a face. The string pointer is simply stored, the string is not copied, so the string must be in static memory.

The second form copies one face to another.

int Fl::set_fonts(const char * = 0)

FLTK will open the display, and add every font on the server to the face table. It will attempt to put "families" of faces together, so that the normal one is first, followed by bold, italic, and bold italic.

The optional argument is a string to describe the set of fonts to add. Passing `NULL` will select only fonts that have the ISO8859-1 character set (and are thus usable by normal text). Passing "-*" will select all fonts with any encoding as long as they have normal X font names with dashes in them. Passing "*" will list every font that exists (on X this may produce some strange output). Other values may be useful but are system dependent. With WIN32 `NULL` selects fonts with ISO8859-1 encoding and non-`NULL` selects all fonts.

The return value is how many faces are in the table after this is done.

static void Fl::set_labeltype(Fl_Labeltype, Fl_Label_Draw_F *, Fl_Label_Measure_F *)
static void Fl:set_labeltype(Fl_Labeltype, Fl_Labeltype from)

The first form sets the functions to call to draw and measure a specific labeltype.

The second form copies the `from` labeltype.

int Fl::test_shortcut(ulong) const

Test the current event, which must be an `FL_KEYBOARD` or `FL_SHORTCUT`, against a shortcut value (described in `Fl_Button`). Returns non-zero if there is a match. Not to be confused with `Fl_Widget::test_shortcut()`.

static int Fl::visual(int)

Selects a visual so that your graphics are drawn correctly. This does nothing if the default visual satisfies the capabilities, or if no visual satisfies the capabilities, or on systems that don't have such brain-dead notions.

Only the following combinations do anything useful:

- `Fl::visual(FL_RGB)`
  Full/true color (if there are several depths FLTK chooses the largest). Do this if you use `fl_draw_image` for much better (non-dithered) output.

- `Fl::visual(FL_RGB8)`
  Full color with at least 24 bits of color. `FL_RGB` will always pick this if available, but if not it will

happily return a less-than-24 bit deep visual. This call fails if 24 bits are not available.

- `Fl::visual(FL_DOUBLE|FL_INDEX)`
  Hardware double buffering. Call this if you are going to use `Fl_Double_Window`.

- `Fl::visual(FL_DOUBLE|FL_RGB)`
- `Fl::visual(FL_DOUBLE|FL_RGB8)`
  Hardware double buffering and full color.

This returns true if the system has the capabilities by default or FLTK suceeded in turing them on. Your program will still work even if this returns false (it just won't look as good).

static int Fl::w()

Returns the width of the screen in pixels.

static int wait()
static double wait(double time)

Calls the idle function if any, then calls any pending timeout functions, then calls `Fl::flush()`. If there are any windows displayed it then waits some time for events (zero if there is an idle(), the shortest timeout if there are any timeouts, or forever) and calls the handle() function on those events, and then returns non-zero.

Your program can check its global state and update things after each call to `Fl::wait()`, which can be very useful in complex programs.

If there are no windows (this is checked after the idle and timeouts are called) then `Fl::wait()` returns zero without waiting for any events. Your program can either exit at this point, or call `show()` on some window so the GUI can continue to operate. The second form of `Fl::wait()` waits only a certain amount of time for anything to happen. This does the same as `wait()` except if the given time (in seconds) passes it returns. The return value is how much time remains. If the return value is zero or negative then the entire time period elapsed.

If you do several `wait(time)` calls in a row, the subsequent ones are measured from when the first one is called, even if you do time-consuming calculations after they return. This allows you to accurately make something happen at regular intervals. This code will accurately call `A()` once per second (as long as it takes less than a second to execute):

```
for (;;) {
  for (float time = 1.0; time > 0; ) time = Fl::wait(time);
  A();
}
```

static void (*Fl::warning)(const char *, ...)
static void (*Fl::error)(const char *, ...)
static void (*Fl::fatal)(const char *, ...)

FLTK will call these to print messages when unexpected conditions occur. By default they `fprintf` to `stderr`, and `Fl::error` and `Fl::fatal` call `exit(1)`. You can override the behavior by setting the function pointers to your own routines.

`Fl::warning` means that there was a recoverable problem, the display may be messed up but the user can probably keep working (all X protocol errors call this). `Fl::error` means there is a recoverable error, but the display is so messed up it is unlikely the user can continue (very little calls this now). `Fl::fatal` must not return, as FLTK is in an unusable state, however your version may be able to use `longjmp` or an exception to continue, as long as it does not call FLTK again.

C - FLTK Enumerations

This appendix lists the enumerations provided in the `<FL/Enumerations.H>` header file, organized by section. Constants whose value is zero are marked with "(0)", this is often useful to know when programming.

Version Numbers

The FLTK version number is stored in a number of compile-time constants:

- `FL_MAJOR_VERSION` - The major release number, currently 1.
- `FL_MINOR_VERSION` - The minor release number, currently 0.
- `FL_PATCH_VERSION` - The patch release number, currently 0.
- `FL_VERSION` - A combined floating-point version number for the major and minor release numbers, currently 1.0.

Events

Events are identified by an `Fl_Event` enumeration value. The following events are currently defined:

- `FL_NO_EVENT` - No event (or an event fltk does not understand) occurred (0).
- `FL_PUSH` - A mouse button was pushed.
- `FL_RELEASE` - A mouse button was released.
- `FL_ENTER` - The mouse pointer entered a widget.
- `FL_LEAVE` - The mouse pointer left a widget.

- FL_DRAG - The mouse pointer was moved with a button pressed.
- FL_FOCUS - A widget should receive keyboard focus.
- FL_UNFOCUS - A widget loses keyboard focus.
- FL_KEYBOARD - A key was pressed.
- FL_CLOSE - A window was closed.
- FL_MOVE - The mouse pointer was moved with no buttons pressed.
- FL_SHORTCUT - The user pressed a shortcut key.
- FL_DEACTIVATE - The widget has been deactivated.
- FL_ACTIVATE - The widget has been activated.
- FL_HIDE - The widget has been hidden.
- FL_SHOW - The widget has been shown.
- FL_PASTE - The widget should paste the contents of the clipboard.
- FL_SELECTIONCLEAR - The widget should clear any selections made for the clipboard.

Callback "When" Conditions

The following constants determine when a callback is performed:

- FL_WHEN_NEVER - Never call the callback (0).
- FL_WHEN_CHANGED - Do the callback only when the widget value changes.
- FL_WHEN_NOT_CHANGED - Do the callback whenever the user interacts with the widget.
- FL_WHEN_RELEASE - Do the callback when the button or key is released and the value changes.
- FL_WHEN_ENTER_KEY - Do the callback when the user presses the ENTER key and the value changes.
- FL_WHEN_RELEASE_ALWAYS - Do the callback when the button or key is released, even if the value doesn't change.
- FL_WHEN_ENTER_KEY_ALWAYS - Do the callback when the user presses the ENTER key, even if the value doesn't change.

Fl::event_key() Values

The following constants define the non-ASCII keys on the keyboard for FL_KEYBOARD and FL_SHORTCUT events:

- FL_Button - A mouse button; use Fl_Button + n for mouse button n.
- FL_BackSpace - The backspace key.
- FL_Tab - The tab key.
- FL_Enter - The enter key.
- FL_Pause - The pause key.
- FL_Scroll_Lock - The scroll lock key.
- FL_Escape - The escape key.
- FL_Home - The home key.
- FL_Left - The left arrow key.
- FL_Up - The up arrow key.
- FL_Right - The right arrow key.
- FL_Down - The down arrow key.
- FL_Page_Up - The page-up key.
- FL_Page_Down - The page-down key.
- FL_End - The end key.

- FL_Print - The print (or print-screen) key.
- FL_Insert - The insert key.
- FL_Menu - The menu key.
- FL_Num_Lock - The num lock key.
- FL_KP - One of the keypad numbers; use FL_KP + n for number n.
- FL_KP_Enter - The enter key on the keypad.
- FL_F - One of the function keys; use FL_F + n for function key n.
- FL_Shift_L - The lefthand shift key.
- FL_Shift_R - The righthand shift key.
- FL_Control_L - The lefthand control key.
- FL_Control_R - The righthand control key.
- FL_Caps_Lock - The caps lock key.
- FL_Meta_L - The left meta/Windows key.
- FL_Meta_R - The right meta/Windows key.
- FL_Alt_L - The left alt key.
- FL_Alt_R - The right alt key.
- FL_Delete - The delete key.

Fl::event_state() Values

The following constants define bits in the Fl::event_state() value:

- FL_SHIFT - One of the shift keys is down.
- FL_CAPS_LOCK - The caps lock is on.
- FL_CTRL - One of the ctrl keys is down.
- FL_ALT - One of the alt keys is down.
- FL_NUM_LOCK - The num lock is on.
- FL_META - One of the meta/Windows keys is down.
- FL_SCROLL_LOCK - The scroll lock is on.
- FL_BUTTON1 - Mouse button 1 is pushed.
- FL_BUTTON2 - Mouse button 2 is pushed.
- FL_BUTTON3 - Mouse button 3 is pushed.

Alignment Values

The following constants define bits that can be used with Fl_Widget::align() to control the positioning of the label:

- FL_ALIGN_CENTER - The label is centered (0).
- FL_ALIGN_TOP - The label is top-aligned.
- FL_ALIGN_BOTTOM - The label is bottom-aligned.
- FL_ALIGN_LEFT - The label is left-aligned.
- FL_ALIGN_RIGHT - The label is right-aligned.
- FL_ALIGN_CLIP - The label is clipped to the widget.
- FL_ALIGN_WRAP - The label text is wrapped as needed.
- FL_ALIGN_TOP_LEFT
- FL_ALIGN_TOP_RIGHT
- FL_ALIGN_BOTTOM_LEFT
- FL_ALIGN_BOTTOM_RIGHT

- FL_ALIGN_LEFT_TOP
- FL_ALIGN_RIGHT_TOP
- FL_ALIGN_LEFT_BOTTOM
- FL_ALIGN_RIGHT_BOTTOM
- FL_ALIGN_INSIDE - 'or' this with other values to put label inside the widget.

Fonts

The following constants define the standard FLTK fonts:

- FL_HELVETICA - Helvetica (or Arial) normal (0).
- FL_HELVETICA_BOLD - Helvetica (or Arial) bold.
- FL_HELVETICA_ITALIC - Helvetica (or Arial) oblique.
- FL_HELVETICA_BOLD_ITALIC - Helvetica (or Arial) bold-oblique.
- FL_COURIER - Courier normal.
- FL_COURIER_BOLD - Courier bold.
- FL_COURIER_ITALIC - Courier italic.
- FL_COURIER_BOLD_ITALIC - Courier bold-italic.
- FL_TIMES - Times roman.
- FL_TIMES_BOLD - Times bold.
- FL_TIMES_ITALIC - Times italic.
- FL_TIMES_BOLD_ITALIC - Times bold-italic.
- FL_SYMBOL - Standard symbol font.
- FL_SCREEN - Default monospaced screen font.
- FL_SCREEN_BOLD - Default monospaced bold screen font.
- FL_ZAPF_DINGBATS - Zapf-dingbats font.

Colors

The following color constants can be used to access the colors in the FLTK standard color palette:

- FL_BLACK - the default label color (0)
- FL_RED
- FL_GREEN
- FL_YELLOW
- FL_BLUE
- FL_MAGENTA
- FL_CYAN
- FL_WHITE - the default background for text
- FL_SELECTION_COLOR - change to dark blue for Windows style
- FL_GRAY - the default color.

In addition there are two inline functions to allow you to select grays or colors from the FLTK colormap:

Fl_Color fl_gray_ramp(int i)
Returs a gray color. Returns black for zero, returns white for `FL_NUM_GRAY` (which is 24) minus 1. To get the closest to an 8-bit gray value 'I' use `fl_gray_ramp(I*FL_NUM_GRAY/256)`

Fl_Color fl_color_cube(int r, int g, int b)

Returns a color out of the color cube. `r` must be in the range 0 to FL_NUM_RED (5) minus 1. `g` must be in the range 0 to FL_NUM_GREEN (8) minus 1. `b` must be in the range 0 to FL_NUM_BLUE (5) minus 1. To get the closest color to a 8-bit set of R,G,B values use `fl_color_cube(R*FL_NUM_RED/256, G*FL_NUM_GREEN/256, B*FL_NUM_BLUE/256);`

Cursors

The following constants define the mouse cursors that are available in FLTK. The double-headed arrows are bitmaps provided by FLTK on X, the others are provided by system-defined cursors.

- `FL_CURSOR_DEFAULT` - the default cursor, usually an arrow (0)
- `FL_CURSOR_ARROW` - an arrow pointer
- `FL_CURSOR_CROSS` - crosshair
- `FL_CURSOR_WAIT` - watch or hourglass
- `FL_CURSOR_INSERT` - I-beam
- `FL_CURSOR_HAND` - hand (uparrow on MSWindows)
- `FL_CURSOR_HELP` - question mark
- `FL_CURSOR_MOVE` - 4-pointed arrow
- `FL_CURSOR_NS` - up/down arrow
- `FL_CURSOR_WE` - left/right arrow
- `FL_CURSOR_NWSE` - diagonal arrow
- `FL_CURSOR_NESW` - diagonal arrow
- `FL_CURSOR_NONE` - invisible

FD "When" Conditions

- `FL_READ` - Call the callback when there is data to be read.
- `FL_WRITE` - Call the callback when data can be written without blocking.
- `FL_EXCEPT` - Call the callback if an exception occurs on the file.

Damage Masks

The following damage mask bits are used by the standard FLTK widgets:

- `FL_DAMAGE_CHILD` - A child needs to be redrawn.
- `FL_DAMAGE_EXPOSE` - The window was exposed.
- `FL_DAMAGE_SCROLL` - The `Fl_Scroll` widget was scrolled.
- `FL_DAMAGE_OVERLAY` - The overlay planes need to be redrawn.
- `FL_DAMAGE_ALL` - Everything needs to be redrawn.

D - GLUT Compatibility

This appendix describes the GLUT compatibility header file supplied with FLTK.

Using the GLUT Compatibility Header File

You should be able to compile existing GLUT source code by including `<FL/glut.H>` instead of `<GL/glut.h>`. This can be done by editing the source, by changing the `-I` switches to the compiler, or by providing a symbolic link from `GL/glut.h` to `FL/glut.H`.

All files calling GLUT procedures must be compiled with C++. You may have to alter them slightly to get them to compile without warnings, and you may have to rename them to get make to use the C++ compiler.

You must link with the FLTK library. If you call any GLUT drawing functions that FLTK does not emulate (`glutExtensionsSupported()` , `glutWire*()`, `glutSolid*()`, and `glutStroke*()`), you will also have to link with the GLUT library (*after* the FLTK library!)

Most of `FL/glut.H` is inline functions. You should take a look at it (and maybe at `test/glpuzzle.cxx` in the FLTK source) if you are having trouble porting your GLUT program.

This has been tested with most of the demo programs that come with the GLUT 3.3 distribution.

Known Problems

The following functions and/or arguments to functions are missing, and you will have to replace them or comment them out for your code to compile:

- `glutLayerGet(GLUT_LAYER_IN_USE)`
- `glutLayerGet(GLUT_HAS_OVERLAY)`
- `glutSetColor(), glutGetColor(), glutCopyColormap()`
- `glutInitDisplayMode(GLUT_STEREO)`
- `glutInitDisplayMode(GLUT_LUMINANCE)`
- `glutPushWindow()`
- `glutWarpPointer()`
- Spaceball, buttonbox, dials, tablet functions, `glutDeviceGet()`
- `glutWindowStatusFunc()`
- `glutGet(GLUT_WINDOW_NUM_CHILDREN)`
- `glutGet(GLUT_SCREEN_WIDTH_MM)`
- `glutGet(GLUT_SCREEN_HEIGHT_MM)`
- `glutGet(GLUT_ELAPSED_TIME)`
- `glutVideoResize()` missing.

Most of the symbols/enumerations have different values than GLUT uses. This will break code that relies on the actual values. The only symbols guaranteed to have the same values are true/false pairs like `GLUT_DOWN` and `GLUT_UP`, mouse buttons `GLUT_LEFT_BUTTON, GLUT_MIDDLE_BUTTON, GLUT_RIGHT_BUTTON`, and `GLUT_KEY_F1` thru `F12`.

The strings passed as menu labels are not copied.

`glutPostRedisplay()` does not work if called from inside a display function. You must use `glutIdleFunc()` if you want your display to update continuously.

`glutSwapBuffers()` does not work from inside a display function. This is on purpose, because FLTK swaps the buffers for you.

`glutUseLayer()` does not work well, and should only be used to initialize transformations inside a resize callback. You should redraw overlays by using `glutOverlayDisplayFunc()`.

Overlays are cleared before the overlay display function is called. `glutLayerGet(GLUT_OVERLAY_DAMAGED)` always returns true for compatibility with some GLUT overlay programs. You must rewrite your code so that `gl_color()` is used to choose colors in an overlay, or you will get random overlay colors.

`glutSetCursor(GLUT_CURSOR_FULL_CROSSHAIR)` just results in a small crosshair.

The fonts used by `glutBitmapCharacter() and glutBitmapWidth()` may be different.

`glutInit(argc,argv)` will consume different switches than GLUT does. It accepts the switches recognized by `Fl::args()`, and will accept any abbreviation of these switches (such as "-di" for "-display").

Mixing GLUT and FLTK Code

You can make your GLUT window a child of a `Fl_Window` with the following scheme. The biggest trick is that GLUT insists on `show()` 'ing the window at the point it is created, which means the `Fl_Window` parent window must already be shown.

- Don't call `glutInit()`.
- Create your `Fl_Window`, and any FLTK widgets. Leave a blank area in the window for your GLUT window.
- `show()` the `Fl_Window`. Perhaps call `show(argc,argv)`.
- Call `window->begin()` so that the GLUT window will be automatically added to it.
- Use `glutInitWindowSize()` and `glutInitWindowPosition()` to set the location in the parent window to put the GLUT window.
- Put your GLUT code next. It probably does not need many changes. Call `window->end()` immediately after the `glutCreateWindow()`!
- You can call either `glutMainLoop()`, `Fl::run()`, or loop calling `Fl::wait()` to run the program.

class Fl_Glut_Window

Class Hierarchy

```
Fl_Gl_Window
   |
   +----Fl_Glut_Window
```

Include Files

```
#include <FL/glut.H>
```

Description

Each GLUT window is an instance of this class. You may find it useful to manipulate instances directly rather than use GLUT window id's. These may be created without opening the display, and thus can fit better into FLTK's method of creating windows.

The current GLUT window is available in the global variable `glut_window`.

`new Fl_Glut_Window(...)` is the same as `glutCreateWindow()` except it does not `show()` the window or make the window current.

`window->make_current()` is the same as `glutSetWindow(number)`. If the window has not had `show()` called on it yet, some functions that assumme an OpenGL context will not work. If you do `show()` the window, call `make_current()` again to set the context.

`~Fl_Glut_Window()` is the same as `glutDestroyWindow()` .

Methods

- Fl_Glut_Window
- ~Fl_Glut_Window

Fl_Glut_Window::Fl_Glut_Window(int x, int y, int w, int h, const char *title = 0)
Fl_Glut_Window::Fl_Glut_Window(int w, int h, const char *title = 0)

The first constructor takes 4 int arguments to create the window with a preset position and size. The second constructor with 2 arguments will create the window with a preset size, but the window manager will choose the position according to it's own whims.

virtual Fl_Glut_Window::~Fl_Glut_Window()

Destroys the GLUT window.

E - Forms Compatibility

This appendix describes the Forms compatibility included with FLTK.

Importing Forms Layout Files

FLUID can read the .fd files put out by all versions of Forms and XForms fdesign. However, it will mangle them a bit, but it prints a warning message about anything it does not understand. FLUID cannot write fdesign files, so you should save to a new name so you don't write over the old one.

You will need to edit your main code considerably to get it to link with the output from FLUID. If you are not interested in this you may have more immediate luck with the forms compatibility header, `<FL/forms.H>`.

Using the Compatibility Header File

You should be able to compile existing Forms or XForms source code by changing the include directory switch to your compiler so that the `forms.h` file supplied with FLTK is included. Take a look at `forms.h` to see how it works, but the basic trick is lots of inline functions. Most of the XForms demo programs work without changes.

You will also have to compile your Forms or XForms program using a C++ compiler. The FLTK library does not provide C bindings or header files.

Although FLTK was designed to be compatable with the GL Forms library (version 0.3 or so), XForms has bloated severely and it's interface is X-specific. Therefore, XForms compatibility is no longer a goal of FLTK. Compatibility was limited to things that were free, or that would add code that would not be linked in if the feature is unused, or that was not X-specific.

To use any new features of FLTK, you should rewrite your code to not use the inline functions and instead use "pure" FLTK. This will make it a lot cleaner and make it easier to figure out how to call the FLTK functions. Unfortunately this conversion is harder than expected and even Digital Domain's inhouse code still uses `forms.H` a lot.

Problems You Will Encounter

Many parts of XForms use X-specific structures like `XEvent` in their interface. I did not emulate these! Unfortunately these features (such as the "canvas" widget) are needed by most large programs. You will need to rewrite these to use FLTK subclasses.

`Fl_Free` widgets emulate the *old* Forms "free" widget. It may be useful for porting programs that change the `handle()` function on widgets, but you will still need to rewrite things.

`Fl_Timer` widgets are provided to emulate the XForms timer. These work, but are quite inefficient and inaccurate compared to using `Fl::add_timeout()`.

All instance variables are hidden. If you directly refer to the x, y, w, h, label, or other fields of your Forms widgets you will have to add empty parenthesis after each reference. The easiest way to do this is to globally replace "->x" with "->x()", etc. Replace "boxtype" with "box()".

`const char *` arguments to most FLTK methods are simply stored, while Forms would `strdup()` the passed string. This is most noticable with the label of widgets. Your program must always pass static data such as a string constant or malloc'd buffer to `label()`. If you are using labels to display program output you may want to try the `Fl_Output` widget.

The default fonts and sizes are matched to the older GL version of Forms, so all labels will draw somewhat larger than an XForms program does.

fdesign outputs a setting of a "fdui" instance variable to the main window. I did not emulate this because I wanted all instance variables to be hidden. You can store the same information in the `user_data()` field of a window. To do this, search through the fdesign output for all occurances of "->fdui" and edit to use "->user_data()" instead. This will require casts and is not trivial.

The prototype for the functions passed to `fl_add_timeout()` and `fl_set_idle_callback()` callback are different.

All the following XForms calls are missing:

- `FL_REVISION`, `fl_library_version()`
- `FL_RETURN_DBLCLICK` (use `Fl::event_clicks()`)
- `fl_add_signal_callback()`
- `fl_set_form_atactivate()` `fl_set_form_atdeactivate()`
- `fl_set_form_property()`

- fl_set_app_mainform(), fl_get_app_mainform()
- fl_set_form_minsize(), fl_set_form_maxsize()
- fl_set_form_event_cmask(), fl_get_form_event_cmask()
- fl_set_form_dblbuffer(), fl_set_object_dblbuffer() (use an Fl_Double_Window instead)
- fl_adjust_form_size()
- fl_register_raw_callback()
- fl_set_object_bw(), fl_set_border_width()
- fl_set_object_resize(), fl_set_object_gravity()
- fl_set_object_shortcutkey()
- fl_set_object_automatic()
- fl_get_object_bbox() (maybe FLTK should do this)
- fl_set_object_prehandler(), fl_set_object_posthandler()
- fl_enumerate_fonts()
- Most drawing functions
- fl_set_coordunit() (FLTK uses pixels all the time)
- fl_ringbell()
- fl_gettime()
- fl_win*() (all these functions)
- fl_initialize(argc,argv,x,y,z) ignores last 3 arguments
- fl_read_bitmapfile(), fl_read_pixmapfile()
- fl_addto_browser_chars()
- FL_MENU_BUTTON just draws normally
- fl_set_bitmapbutton_file(), fl_set_pixmapbutton_file()
- FL_CANVAS objects
- FL_DIGITAL_CLOCK (comes out analog)
- fl_create_bitmap_cursor(), fl_set_cursor_color()
- fl_set_dial_angles()
- fl_show_oneliner()
- fl_set_choice_shortcut(a,b,c)
- command log
- Only some of file selector is emulated
- FL_DATE_INPUT
- fl_pup*() (all these functions)
- textbox object (should be easy but I had no sample programs)
- xyplot object

Additional Notes

These notes were written for porting programs written with the older IRISGL version of Forms. Most of these problems are the same ones encountered when going from old Forms to XForms:

Does Not Run In Background

The IRISGL library always forked when you created the first window, unless "foreground()" was called. FLTK acts like "foreground()" is called all the time. If you really want the fork behavior do "if (fork()) exit(0)" right at the start of your program.

You Cannot Use IRISGL Windows or fl_queue

If a Forms (not XForms) program if you wanted your own window for displaying things you would create a IRISGL window and draw in it, periodically calling Forms to check if the user hit buttons on the panels. If the user did things to the IRISGL window, you would find this out by having the value FL_EVENT returned from the call to Forms.

None of this works with FLTK. Nor will it compile, the necessary calls are not in the interface.

You have to make a subclass of `Fl_Gl_Window` and write a `draw()` method and `handle()` method. This may require anywhere from a trivial to a major rewrite.

If you draw into the overlay planes you will have to also write a `draw_overlay()` method and call `redraw_overlay()` on the OpenGL window.

One easy way to hack your program so it works is to make the `draw()` and `handle()` methods on your window set some static variables, storing what event happened. Then in the main loop of your program, call `Fl::wait()` and then check these variables, acting on them as though they are events read from `fl_queue`.

You Must Use OpenGL to Draw Everything

The file `<FL/gl.h>` defines replacements for a lot of IRISGL calls, translating them to OpenGL. There are much better translators available that you might want to investigate.

You Cannot Make Forms Subclasses

Programs that call `fl_make_object` or directly setting the handle routine will not compile. You have to rewrite them to use a subclass of `Fl_Widget`. It is important to note that the `handle()` method is not exactly the same as the `handle()` function of Forms. Where a Forms `handle()` returned non-zero, your `handle()` must call `do_callback()`. And your `handle()` must return non-zero if it "understood" the event.

An attempt has been made to emulate the "free" widget. This appears to work quite well. It may be quicker to modify your subclass into a "free" widget, since the "handle" functions match.

If your subclass draws into the overlay you are in trouble and will have to rewrite things a lot.

You Cannot Use <device.h>

If you have written your own "free" widgets you will probably get a lot of errors about "getvaluator". You should substitute:

Forms	FLTK
MOUSE_X	Fl::event_x_root()
MOUSE_Y	Fl::event_y_root()

LEFTSHIFTKEY,RIGHTSHIFTKEY	Fl::event_shift()
CAPSLOCKKEY	Fl::event_capslock()
LEFTCTRLKEY,RIGHTCTRLKEY	Fl::event_ctrl()
LEFTALTKEY,RIGHTALTKEY	Fl::event_alt()
MOUSE1,RIGHTMOUSE	Fl::event_state()
MOUSE2,MIDDLEMOUSE	Fl::event_state()
MOUSE3,LEFTMOUSE	Fl::event_state()

Anything else in `getvaluator` and you are on your own...

Font Numbers Are Different

The "style" numbers have been changed because I wanted to insert bold-italic versions of the normal fonts. If you use Times, Courier, or Bookman to display any text you will get a different font out of FLTK. If you are really desperate to fix this use the following code:

```
fl_font_name(3,"*courier-medium-r-no*");
fl_font_name(4,"*courier-bold-r-no*");
fl_font_name(5,"*courier-medium-o-no*");
fl_font_name(6,"*times-medium-r-no*");
fl_font_name(7,"*times-bold-r-no*");
fl_font_name(8,"*times-medium-i-no*");
fl_font_name(9,"*bookman-light-r-no*");
fl_font_name(10,"*bookman-demi-r-no*");
fl_font_name(11,"*bookman-light-i-no*");
```

F - Operating System Issues

This appendix describes the X and WIN32 specific interfaces in FLTK.

X-Specific Interface

```
#include <FL/x.H>
```

On X you can include this file to access FLTK's X-specific functions. Be warned that some of the structures and calls in it are subject to change in future version of FLTK. Try to avoid doing this so your code is portable.

Handling Other X Events

void Fl::add_handler(int (*f)(int))

Installs a function to parse unrecognized events. If FLTK cannot figure out what to do with an event, it calls each of these functions (most recent first) until one of them returns non-zero. If none of them returns non-zero then the event is ignored.

FLTK calls this for any X events it does not recognize, or X events with a window id that FLTK does not recognize. You can look at the X event with the `fl_xevent` variable.

The argument is zero for unrecognized X events. These handlers are also called for global shortcuts and some other events that the widget they were passed to did not handle. In this case the argument is non-zero (for example `FL_SHORTCUT`).

extern XEvent *fl_xvent

The most recent X event.

extern ulong fl_event_time

This is the time stamp from the most recent X event that reported it (not all do). Many X calls (like cut and paste) need this value.

Window fl_xid(const Fl_Window *)

Returns the XID for a window, or zero if not `shown()`.

Fl_Window *fl_find(ulong xid)

Returns the `Fl_Window` that corresponds to the given XID, or `NULL` if not found. This uses a cache so it is slightly faster than iterating through the windows yourself.

int fl_handle(const XEvent &)

This call allows you to supply the X events to FLTK, which may allow FLTK to cooperate with another toolkit or library. The return value is true if FLTK understood the event (if the window does not belong to FLTK and the `add_handler()` functions all ignore it this returns false).

Besides feeding events your code should call `Fl::flush()` periodically so that FLTK redraws its windows.

This function will call the callback functions. It will not return until they complete. In particular if a callback pops up a modal window (by calling `fl_ask()`, for instance) it will not return until the modal function returns.

Drawing using Xlib

The following global variables are set before `Fl_Widget::draw()` is called, or by `Fl_Window::make_current()`:

```
extern Display *fl_display;
extern Window fl_window;
extern GC fl_gc;
extern int fl_screen;
extern XVisualInfo *fl_visual;
extern Colormap fl_colormap;
```

You must use them to produce Xlib calls. Don't attempt to change them. A typical X drawing call is written like this:

```
XDrawSomething(fl_display, fl_window, fl_gc, ...);
```

Other information such as the position or size of the X window can be found by looking at

`Fl_Window::current()`, which returns a pointer to the `Fl_Window` being drawn.

unsigned long fl_xpixel(Fl_Color i)
unsigned long fl_xpixel(uchar r, uchar g, uchar b)

Returns the X pixel number used to draw the given FLTK color index or RGB color. This is the X pixel that `fl_color()` would use.

extern XFontStruct *fl_xfont

Points at the font selected by the most recent `fl_font()`. This is not necessarily the current font of `fl_gc`, which is not set until `fl_draw()` is called.

Changing the Display, Screen, or X Visual

FLTK uses only a single display, screen, X visual, and X colormap. This greatly simplifies its internal structure and makes it much smaller and faster. You can change which it uses by setting global variables *before the first `Fl_Window::show()` is called*. You may also want to call Fl::visual() , which is a portable interface to get a full color and/or double buffered visual.

int Fl::display(const char *)

Set which X display to use. This actually does `putenv("DISPLAY=...")` so that child programs will display on the same screen if called with `exec()`. This must be done before the display is opened. This call is provided under WIN32 but it has no effect.

extern Display *fl_display

The open X display. This is needed as an argument to most Xlib calls. Don't attempt to change it! This is `NULL` before the display is opened.

void fl_open_display()

Opens the display. Does nothing if it is already open. This will make sure `fl_display` is non-zero. You should call this if you wish to do X calls and there is a chance that your code will be called before the first `show()` of a window.

This may call `Fl::abort()` if there is an error opening the display.

void fl_close_display()

This closes the X connection. You do *not* need to call this to exit, and in fact it is faster to not do so! It may be useful to call this if you want your program to continue without the X connection. You cannot open the display again, and probably cannot call any FLTK functions.

extern int fl_screen

Which screen number to use. This is set by `fl_open_display()` to the default screen. You can change it by setting this to a different value immediately afterwards. It can also be set by changing the last number in the `Fl::display()` string to "host:0,#".

extern XVisualInfo *fl_visual
extern Colormap fl_colormap

The visual and colormap that FLTK will use for all windows. These are set by `fl_open_display()` to the default visual and colormap. You can change them before calling `show()` on the first window. Typical code for changing the default visual is:

```
Fl::args(argc, argv); // do this first so $DISPLAY is set
fl_open_display();
fl_visual = find_a_good_visual(fl_display, fl_screen);
if (!fl_visual) Fl::abort("No good visual");
fl_colormap = make_a_colormap(fl_display, fl_visual->visual, fl_visual->depth);
// it is now ok to show() windows:
window->show(argc, argv);
```

Using a Subclass of Fl_Window for Special X Stuff

FLTK can manage an X window on a different screen, visual and/or colormap, you just can't use FLTK's drawing routines to draw into it. But you can write your own `draw()` method that uses Xlib (and/or OpenGL) calls only.

FLTK can also manage XID's provided by other libraries or programs, and call those libraries when the window needs to be redrawn.

To do this, you need to make a subclass of `Fl_Window` and override some of these virtual functions:

virtual void Fl_Window::show()

If the window is already `shown()` this must cause it to be raised, this can usually be done by calling `Fl_Window::show()`. If not `shown()` your implementation must call either `Fl_X::set_xid()` or `Fl_X::make_xid()`.

An example:

```
void MyWindow::show() {
  if (shown()) {Fl_Window::show(); return;}  // you must do this!
  fl_open_display();    // necessary if this is first window
  // we only calcualte the necessary visual colormap once:
  static XVisualInfo *visual;
  static Colormap colormap;
  if (!visual) {
    visual = figure_out_visual();
    colormap = XCreateColormap(fl_display, RootWindow(fl_display,fl_screen),
                               vis->visual, AllocNone);
  }
  Fl_X::make_xid(this, visual, colormap);
}
```

Fl_X *Fl_X::set_xid(Fl_Window *, Window xid)

Allocate a hidden structure called an `Fl_X`, put the XID into it, and set a pointer to it from the `Fl_Window`. This causes `Fl_Window::shown()` to return true.

void Fl_X::make_xid(Fl_Window *, XVisualInfo *= fl_visual, Colormap = fl_colormap)

This static method does the most onerous parts of creating an X window, including setting the label, resize limitations, etc. It then does `Fl_X::set_xid()` with this new window and maps the window.

virtual void Fl_Window::flush()

This virtual function is called by `Fl::flush()` to update the window. For FLTK's own windows it does this by setting the global variables `fl_window` and `fl_gc` and then calling the `draw()` method. For your own windows you might just want to put all the drawing code in here.

The X region that is a combination of all `damage()` calls done so far is in `Fl_X::i(this)->region`. If `NULL` then you should redraw the entire window. The undocumented function `fl_clip_region(XRegion)` will initialize the FLTK clip stack with a region or `NULL` for no clipping. You must set region to `NULL` afterwards as `fl_clip_region()` now owns it and will delete it when done.

If `damage() FL_DAMAGE_EXPOSE` then only X expose events have happened. This may be useful if you have an undamaged image (such as a backing buffer) around.

Here is a sample where an undamaged image is kept somewhere:

```
void MyWindow::flush() {
  fl_clip_region(Fl_X::i(this)->region);
  Fl_X::i(this)->region = 0;
  if (damage() != 2) {... draw things into backing store ...}
  ... copy backing store to window ...
}
```

virtual void Fl_Window::hide()

Destroy the window server copy of the window. Usually you will destroy contexts, pixmaps, or other resources used by the window, and then call `Fl_Window::hide()` to get rid of the main window identified by `xid()`. If you override this, you must also override the destructor as shown:

```
void MyWindow::hide() {
  if (mypixmap) {
    XFreePixmap(fl_display,mypixmap);
    mypixmap = 0;
  }
  Fl_Window::hide(); // you must call this
}
```

virtual void Fl_Window::~Fl_Window()

Because of the way C++ works, if you override `hide()` you *must* override the destructor as well (otherwise only the base class `hide()` is called):

```
MyWindow::~MyWindow() {
  hide();
}
```

Setting the Icon of a Window

FLTK currently supports setting a window's icon *before* it is shown using the `Fl_Window::icon()` method.

void Fl_Window::icon(char *)

Sets the icon for the window to the passed pointer. You will need to cast the icon `Pixmap` to a `char *` when calling this method. To set the icon using a bitmap compiled with your application use:

```
#include "icon.xbm"

Pixmap p = XCreateBitmapFromData(fl_display, DefaultRootWindow(fl_display),
                                 icon_bits, icon_width, icon_height);

window->icon((char *)p);
```

WIN32-Specific Interface

```
#include <FL/x.H>
```

The `<FL/x.H>` header file defines the interface to FLTK's WIN32-specific functions. Be warned that some of the structures and calls in it are subject to change in future version of FLTK. Try to avoid doing this so your code is portable.

Handling Other WIN32 Messages

By default a single WNDCLASSEX called "FLTK" is created. All `Fl_Windows` are of this class unless you use `Fl_Window::xclass()` . The window class is created the first time `Fl_Window::show()` is called.

You can probably combine FLTK with other libraries that make their own WIN32 window classes. The easiest way is to call `Fl::wait()` , it will call `DispatchMessage` for all messages to the other windows. If necessary you can let the other library take over (as long as it calls `DispatchMessage()`), but you will have to arrange for the function `Fl::flush()` to be called regularily so that widgets are updated, timeouts are handled, and the idle functions are called.

extern MSG fl_msg

The most recent message read by `GetMessage` (which is called by `Fl::wait()`. This may not be the most recent message sent to an FLTK window, because silly WIN32 calls the handle procedures directly for some events (sigh).

void Fl::add_handler(int (*f)(int))

Install a function to parse unrecognized messages sent to FLTK windows. If FLTK cannot figure out what to do with a message, it calls each of these functions (most recent first) until one of them returns non-zero. The argument passed to the fuctions is zero. If all the handlers return zero then FLTK calls `DefWindowProc()`.

HWND fl_xid(const Fl_Window *)

Returns the window handle for a `Fl_Window`, or zero if not `shown()`.

Fl_Window *fl_find(HWND xid)

Return the `Fl_Window` that corresponds to the given window handle, or `NULL` if not found. This uses a cache so it is slightly faster than iterating through the windows yourself.

Drawing Things Using the WIN32 GDI

When the virtual function `Fl_Widget::draw()` is called, FLTK has stashed in some global variables all the silly extra arguments you need to make a proper GDI call. These are:

```
extern HINSTANCE fl_display;
extern HWND fl_window;
extern HDC fl_gc;
COLORREF fl_RGB();
HPEN fl_pen();
HBRUSH fl_brush();
```

These global variables are set before `draw()` is called, or by `Fl_Window::make_current()`. You can refer to them when needed to produce GDI calls. Don't attempt to change them. The functions return GDI objects for the current color set by `fl_color()` and are created as needed and cached. A typical GDI drawing call is written like this:

```
DrawSomething(fl_gc, ..., fl_brush());
```

It may also be useful to refer to `Fl_Window::current()` to get the window's size or position.

Setting the Icon of a Window

FLTK currently supports setting a window's icon *before* it is shown using the `Fl_Window::icon()` method.

void Fl_Window::icon(char *)

Sets the icon for the window to the passed pointer. You will need to cast the `HICON` handle to a `char *` when calling this method. To set the icon using an icon resource compiled with your application use:

```
window->icon((char *)LoadIcon(fl_display, MAKEINTRESOURCE(IDI_ICON)));
```

How to Not Get a MSDOS Console Window

WIN32 has a really stupid mode switch stored in the executables that controls whether or not to make a console window.

To always get a console window you simply create a console application (the "/SUBSYSTEM:CONSOLE" option for the linker). For a GUI-only application create a WIN32 application (the "/SUBSYSTEM:WINDOWS" option for the linker).

FLTK includes a `WinMain()` function that calls the ANSI standard `main()` entry point for you. *This function creates a console window when you use the debug version of the library.*

WIN32 applications without a console cannot write to `stdout` or `stderr`, even if they are run from a console window. Any output is silently thrown away.

Known Bugs

If a program is deactivated, `Fl::wait()` does not return until it is activated again, even though many events are delivered to the program. This can cause idle background processes to stop unexpectedly. This also happens while the user is dragging or resizing windows or otherwise holding the mouse down. I was forced to remove most of the efficiency FLTK uses for redrawing in order to get windows to update while being moved. This is a design error in WIN32 and probably impossible to get around.

`Fl_Gl_Window::can_do_overlay()` returns true until the first time it attempts to draw an overlay, and then correctly returns whether or not there is overlay hardware.

Cut text contains ^J rather than ^M^J to break lines. This is a feature, not a bug.

`SetCapture` (used by `Fl::grab()`) doesn't work, and the main window title bar turns gray while menus are popped up.

FLUID does not support BMP files yet.

G - Software License

GNU LIBRARY GENERAL PUBLIC LICENSE

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the libary" and a "work that uses the library". The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

> ***a)*** *The modified work must itself be a software library.*
>
> ***b)*** You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
>
> ***c)*** You must cause the whole of the work to be licensed at no charge to all third parties under

the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

> ***a)*** *Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)*
>
> ***b)*** Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
>
> ***c)*** If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
>
> ***d)*** Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

> ***a)*** *Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.*
>
> ***b)*** Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU

OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS