

Software Engineering
CSE 335, Spring 2006

C++, FLTK and Makefiles

Stephen Wagner

Michigan State University

FLTK

- F(ast) L(ight) T(ool)K(it)
 - Designed to help create GUI
 - Class Hierarchy ■
- What's a widget?

Lessons to be learned from FLTK

- Using a large, pre-existing class hierarchy
- Linking in non-standard libraries and using non-standard headers
- Adapting an existing set of classes to use role-based design

FLTK and Wrappers

- The FLTK classes are a bit clumsy
- Instead of using the FLTK classes directly, we will use “wrapper” classes

```
class Button : public Fl_Repeat_Button {
public:
    Button( int x, int y, const char* l );

protected:
    void                pressEvent();
    friend void buttonCallback( Fl_Widget* );
};
```

FLTK

```
int main(void)
{
    ContainerWindow      myContainer(500,150);
    Button               myButton(200, 80, "reset");
    myContainer.resizable(myButton);
    myContainer.end();
    myContainer.show();
    while(Fl::check());
}
```

Containers and Children

- `ContainerWindow` is a `FL_Window`
- Every widget created after the `myContainer` is declared and before `myContainer.end()` is a child of `myContainer`
 - Children are “inside” their parent
 - Moving the parent moves the children
 - Resizing the parent affects the children

Callbacks

- The code `Fl::check()` gives control to the GUI
 - Updates widgets
 - handles events
- How is our code called? ■
- Callbacks

Callbacks

```
Button::Button( int x, int y, const char* label )
  : Fl_Repeat_Button(x, y, 80, 50, label)
{
  callback(buttonCallback);
  show();
}
```

```
void buttonCallback(Fl_Widget* w)
{
  Button* b = dynamic_cast<Button*>(w);
  assert(b != 0);
  b->pressEvent();
}
```

Pitfalls

- Uses C strings, not C++ strings
- Not all calls copy strings

```
const char* Fl_Widget::label() const  
void Fl_Widget::label(const char*)
```

Get or set the current label pointer. The label is shown somewhere on or next to the widget. The passed pointer is stored unchanged in the widget (the string is not copied), so if you need to set the label to a formatted value, make sure the buffer is static, global, or allocated.

Pitfalls

```
void createWidgets()
{
    ContainerWindow      myContainer(500,150);
    Button               myButton(200, 80, "reset");
    myContainer.resizable(myButton);
    myContainer.end();
    myContainer.show();
}

int main(void)
{
    createWidgets();
    while(Fl::check());    }
```

Avoiding Pitfalls

- There are a lot of potential errors
- Read documentation
- Borrow code from the homework
- Understand the homework
- Code small examples

Compiling with FLTK

- `g++ -c -I/user/research/stire/fltk mycode.cc`
- `g++ -o app -L/user/research/stire/fltk/lib -lfltk mycode.o`
- What do `-I` and `-L` do?
- Compilation commands get very complex for large software

Make

- Essentially a *command generator*.
 - Using a description file, it creates a sequence of commands for execution.
 - Commands commonly relate to the maintenance of files that comprise a software-development project.
- Used primarily to sort out dependency relationships among files and rebuild selectively.
- Many other uses:
 - Document formatting;
 - Cleaning out directories;
 - Status reporting and project tracking.

Description file

- Typically named `Makefile` or `makefile`.
- Contains stanzas that comprise three components:
 - target** SCI (software configuration item) that is constructed from (and dependent upon) other SCIs;
 - prerequisites** the SCIs used to construct the target; and
 - commands** procedure(s) to construct the target from the prerequisites.
- Syntax: ¹

```
target: prereq1 prereq2 ... prereqn
    command1
    ...
    commandm
```

¹Note: Tab character used to identify command lines.

Example Makefile ²

```
program: main.o iodat.o lo.o /usr/fred/lib/crtn.a
    cc -o program main.o iodat.o lo.o \
        -L /usr/fred/lib -lcrtn

main.o: main.c
    cc -c main.c

iodat.o: iodat.c
    cc -c iodat.c

lo.o: lo.s
    as -o lo.o lo.s
```

²The backslash at the end of the first command line denotes a line continuation.

Large makefiles

- In real production software, Makefiles could become huge.
 - Easily thousands of lines long; sometimes longer.
 - Lots of seemingly redundant rules and dependencies.
- In practice, the size of a Makefile can be contained using two powerful features:
 - Macros, which represent long strings of text; and
 - Derived rules, which make can adapt to fit a particular context from a general pattern.

Makefile macros

- Symbolic names defined by some text.
- Used by placing their name in braces following a dollar sign.
- Example:³

```
CC = g++ -O2 -c
TEMPLATES = -fexternal-templates
...
ui.o: ui.cc
    ${CC} ui.cc
...
db.o: db.cc
    ${CC} ${TEMPLATES} db.cc
```

³CC and TEMPLATES are macros.

Deriving rules

- If a target has no rebuild rule in the Makefile, then make attempts to *derive* a rebuild rule for the target.
- Facilities for deriving rebuild rules:
 - Suffix rules.
 - SCCS/RCS retrieval.
 - A *default* rule specified using the .DEFAULT target.
- Disciplined use of these features can greatly reduce the size and improve the readability of a Makefile.

Suffix rules

- Rule schemas that match targets and prerequisites based on suffixes of their file names.

- Example

```
.SUFFIXES : .c .o .s
```

```
.c.o :
```

```
    ${CC} ${CFLAGS} -c $<
```

```
.s.o :
```

```
    ${AS} ${ASFLAGS} -o $@ $<
```

- The special macro \$<, which can only be used in suffix rules, refers to the prerequisite that triggered the rule.
- The special macro \$* refers to the prerequisite, *less* the suffix itself.

```
CPLUSPLUS      = g++
CPLPLFLAGS     = -I${FLTKHOME} ${XWININC} -I. -g
FLTKHOME       = /user/research/stire/fltk
XWINLIB        = -L/usr/X11R6/lib -lGLU -lGL -lX11 -lXext -lm
PTHREADLIB     =
APP            = app
OBJS           = app.o IvalSlider.o Button.o IvalText.o processGUIBehavior.o
```

```
include ${FLTKHOME}/makeinclude
```

```
LDLIBS         = -L${FLTKHOME}/lib -lfltk ${XWINLIB} ${PTHREADLIB}
```

```
.SUFFIXES: .cc .h .o
```

```
.cc.o:
```

```
    ${CPLUSPLUS} ${CPLPLFLAGS} -c $<
```

```
${APP}: ${OBJS}
```

```
    ${CPLUSPLUS} ${OBJS} ${LDLIBS} -o $@
```

```
clean::
```

```
    ${RM} *.o *~ ${APP}
```