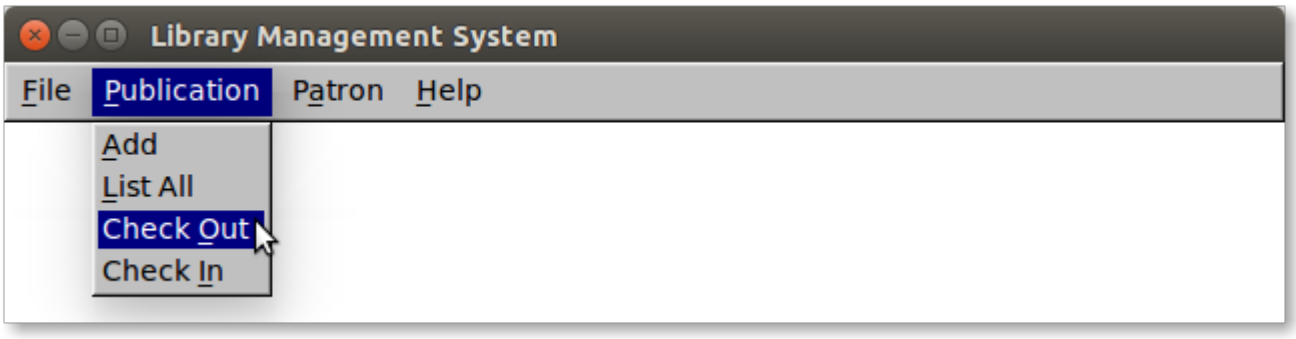
Getting Organized – Library Model

Sprint #3

CSE 1325 – Spring 2017 – Homework #4
Due Tuesday, March 7 at 8:00 am

IMPORTANT: You may NOT work on any homework project in teams until explicitly instructed to do so. Team-optional projects begin AFTER the second exam.

While Graphical User Interfaces (GUI) are better received by most computer users than Command Line Interfaces (CLI), users expect to have a "main window" with menus providing access to the programs functions rather than a stream of dialogs. In this third and final sprint, we will replace the fl_input-based main menu with a main window complete with menu bar, tool bar, and display area listing our publications.



Requirements

Port your solution (or the suggested solution provided with this assignment on March 1, if you prefer), to a fuller GUI implementation. For this sprint, you will implement a main window with pull-down menu bar (full credit level), add a task bar (bonus level), and provide a scrolling list of publications below the task bar (in the extreme bonus level).

(**You will receive partial credit if you only implement a portion of the requirements.** You will be graded on the extent to which you cover the requirements and the quality of your code. **For FLTK programs only,** you are permitted – but not required – to put all of your code into a single source code file. Use git to version control your software, and Scrum to manage this third and final sprint. If you don't understand the *intent* of a requirement, feel free to ask – although reasonable assumptions without asking are also fine if you've used a library.)

Suggested Approach

FLTK can be... temperamental. Therefore, I recommend the following approach.

1. First **build and verify the application from sprint #2 that you will be using for this assignment.** If using the instructor-supplied solution, a simple "make" should suffice. Your solution, if you prefer, may use "`fltk-config --compile library_gui.cpp`". Identifying the correct command(s) to compile your sprint #2 baseline code is your responsibility – I cannot answer vague questions such as "my code doesn't compile, why not?", though I'm willing to advise given your exact source files, Makefiles, and compilation command(s) when used in our standard environment.

2. Add the necessary additional FLTK headers to the top of the single source file. For the full credit level, that's probably FL/Fl_Window.H and FL/Fl_Menu_Bar.H. Verify that the program still compiles and runs. It should behave identically to the pre-modification version.

3. Now, make the necessary changes to create a main window, perhaps using as a pattern the FL Paint 1325 program from lectures 12 and 13. Keep it simple – remember, baby steps, although you'll have to move the entire main menu to the menu bar in one step. (Hint: **If you followed the Model – View – Controller pattern, then FL::run() replaces Controller entirely.** The code in the "if" or "switch" clauses of Controller::execute_cmd in sprint #2 now becomes (virtually unmodified) the code for the callbacks from your menu entries. If you switch to pointers for your Library and View instances, then remember to change e.g., library.add_publication to library->add_publication – but using pointers is NOT required.)

   You may use the same dialogs as in sprint #2 to gather information and respond to the menu-initiated request. If you did the bonus or extreme bonus levels on earlier sprints, you can leverage that for additional points for this sprint – **as in real life, doing solid extra work early builds momentum quickly toward superior and valuable software.**

   Build and interactively test your modification. The debugger is your friend. Don't proceed until it works well, and you understand *why* it works well.

4. Now, successively make any other necessary changes to meet the full credit, bonus, and extreme bonus levels, testing carefully and adding to git after each one. These can generally be implemented in baby steps – a tool bar with one button is a giant baby step toward a full tool bar, and a simple publication table below the tool bar can be incrementally improved until it offers the capabilities you desire.

Grading

- **Full Credit** – For the third sprint, update the Product Backlog tab of the Scrum spreadsheet with the new features (you can **manually merge** rows from the included Scrum_P6 spreadsheet – do not use it directly!). Then, duplicate the Sprint_02_Backlog tab and rename it Sprint_02_Backlog, **leaving Sprint_01_Backlog and Sprint_02_Backlog unmodified** (this will be kept permanently as a record of the first and second sprint, respectively). Then update cells B1:B3 with the updated info. Delete all of the task rows completed in sprint #2, then add rows for the tasks needed to complete the new features for sprint #3. Use this tab to track your progress during sprint #3.

  You will deliver your .h and .cpp class implementations, or a single file named library_gui.cpp, along with a Makefile that is competent to rebuild only files that have been modified, if applicable. Also include screenshots demonstrating each menu of the main window as images in PNG format, and your updated Scrum spreadsheet for the second sprint, including the Scrum results from your first and second sprint.

- **Bonus** – If you modify the main window to also include a tool bar immediately below the menu bar, with at least buttons to list all publications, add a publication, check out a publication, check in a publication, list all patrons, add a patron, and display minimal help. Each button must include a (copyright-compatible) icon and tool tip that displays an short English hint (e.g., “Add publication”) for each button. Additional tool bar functionality will be worth additional points.

  You will deliver your updated class implementations or library_gui.cpp, a Makefile, screen shots of each tool tip in PNG format, and your updated Scrum spreadsheet for the third sprint showing the additional tasks.

- **Extreme Bonus** – If you add a scrollable display of publications to the main window below the tool bar, using e.g., the Fl_Table widget. The table should list the 7 attributes of each publication (title, author, copyright, genre, media, age, and isbn) in columns, with one row per publication. You will receive even more points if the publications can be sorted by each column, and a ludicrous number of points if the tool bar includes a text field that can be used to filter the table to only include publications matching the string or pattern entered.

  You will deliver your updated class implementations or library_gui.cpp, a Makefile, a screenshot of the main window in PNG format **with enough publications created to make the scroll bars visible**, and your updated Scrum spreadsheet for the third sprint showing the additional tasks.

Frequently Asked Questions

Q. Where do I get icons to use for my tool bar?

Oh, anywhere you like – *as long as you respect copyright law*.

- You may draw your own, if you have a vaguely artistic bent, or just don't care. The graders will not be judging your artistic ability.
- You may go to a public domain collection, as I did for my first book, or to a creative commons library, as I did for the icon for the Homework #5 solution (from The Noun Project's Sathish Selladurai collection). The Wiki Commons is a good place to find art licensed under the Creative Commons or the Gnu Free Documentation License (GFDL).
- You may use licensed clipart that came with a program you own, or a clip art collection you previously purchased – I have several million images that for which we've purchased a perpetual license over the years (disk space is cheap).

If the grader detects a violation of copyright law, your grade could be reduced as low as a 0, and for more egregious offenses may result in your being reported for an honor code violation.

Integrity matters.

From Homework #5:

Q: The FLTK documentation for setting icons and window titles in dialogs is rather sparse. Can you provide some additional guidance or a code example, since we didn't cover that in class?

I'd be delighted. Both the window title and the icon are controlled by separate functions in FLTK.

The window title is fairly simple – just call fl_message_title before every dialog, specifying the string that you want to see in the title bar. If the string is a variable rather than a constant, use the c_str() method to convert it to a char* per the fl_message_title declaration in the documentation.

The icon is simple an Fl_Box object, a pointer to which is returned by the function fl_message_icon. You can do anything to this Fl_Box object that you can do to any other Fl_Box object, but note that unlike fl_message_title, which you must call before every dialog, changes to this Fl_Box object are persistent – add an icon once, and you'll get an icon every dialog until you change it.

Two options are of primary interest.

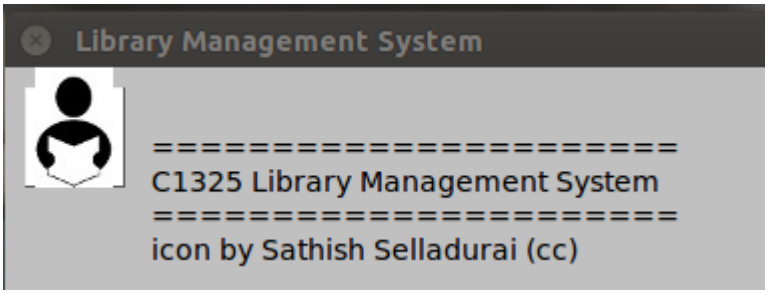
- You can add an icon proper by loading an image into memory (as shown in lecture 9), and then calling the Fl_Box object's image method with a pointer to that image. Fl_Box does NOT scale this image, so you need to make it of the appropriate size.
- OR, you can set the label of the Fl_Box to the single character that you want to use as an "icon", and that usually turns out pretty good.
    - If you are using an image as the icon, you need to set the label to a null string, otherwise you'll get both your image and a question mark.
    - Similarly, if you used an image earlier and now want to use a label, you need to set the

image to null.

So, based on that, my code for the main menu looks something like this:

```
fl_message_title("Library Management System");
string no_label = "";
fl_message_icon()->label(no_label.c_str());
fl_message_icon()->image(*jpg);
cmd = atoi(fl_input(menu.c_str(), 0));
fl_message_icon()->image(0);  // be ready for label icons
```

and the resulting icon like this:

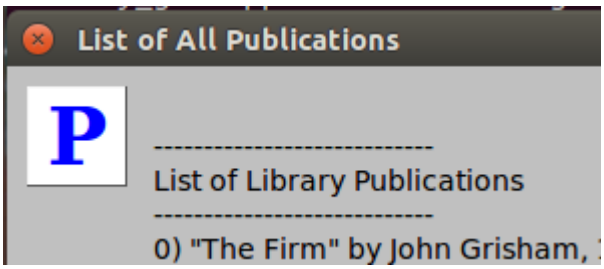


I had to add some whitespace at the top of the icon, because FLTK 1.3.3 insists on positioning it above the box. Perhaps you can do better.

For the rest of the dialogs, I just used a capital letter as the icon, e.g.,

```
fl_message_title("List of All Publications");
fl_message_icon()->label("P");
fl_message(list.c_str());
```

resulting in



That's all there is to it. Just lather, rinse, repeat for all the dialogs and you're there.

Q: When I run a simple fl_input program, I get a segmentation fault. Why?

I've been able to reproduce this problem on some (but not all) systems. See the Suggested Approach above for a work around.

From Homework #4:

Q. How do I implement show_menu()()?

That was a typo. I've updated the class diagram above to remove the redundant parentheses.

Q. How do I create a new Publication object without a constructor?

Another typo (though technically permissible). you'll need to add a constructor to the Publication class into which you'll pass the title, author, copyright, etc. I added one to the class diagram above.

Q. What does Publication::to_string() do?

As in earlier homework assignments, to_string returns a string containing a textual representation of the object. An example is shown in paragraph 5 of the requirements.

> "The Firm" by John Grisham, 1991 (adult fiction book) ISBN: 0440245923
> Checked out to Professor Rice (817-272-3785)

Q. What types are Genre, Media, and Age?

The types Genre, Media, and Age are left to your preferred implementation.

Option #1: They are well-suited to be enums, with the enumerated values given in paragraph 2 of the requirements:

> The library consists of a lot of publications in several types of **media** – books, periodicals (also called magazines), newspapers, audio, and video. Each publication falls into a particular **genre** – fiction, non-fiction, self-help, or performance – and target **age** – children, teen, adult, or restricted (which means adult only).

The problem with enums is that it's obviously unacceptable to print "age 3" or "media 5" when printing out a publication – you'll need a helper function (NOT a good object-oriented approach) or a (possibly private) method of Publication (reasonable) to convert the resulting int back into a string for Publication::to_string. Calling these methods "genre_to_string" et. al. may offer a certain consistency.

Option #2: Or, you can make them each a class with an intrinsic to_string method of their own (called from the publication's to_string method).

Option #3: Or, to stretch the definition a bit, you could just make them a string, either directly or via typedef, which we haven't covered yet - see http://www.cplusplus.com/doc/tutorial/other_data_types/. The downside is that you'll need some significant data validation code to ensure that *every string in every object is always correct*. If you just let users type whatever they want, you'll have a disaster instead of a database.

I used a simple class in the example solution (revealed Tuesday), because that's the more OOP way, but any of those 3 (or something C++ish that you think of yourself) is fine.

Q. The UML shows a Main class with a main() method. How is this implemented in C++?

Remember that the UML is not specific to C++ - a UML class diagram can be implemented in any language. Since some languages (looking at you, Java) require that main() be part of a class, a UML class diagram may show the design in this way.

As discussed in previous assignments, however, C++ is unable to specify main() as part of a class, so it **must** be implemented as a stand-alone function:

```
int main() {
  // your code here
}
```

Part of understanding UML is to be able to translate from constructs valid in the UML to constructs that a C++ compiler can handle. This is one such case.

Q. What type do I return if the UML specification of a method doesn't list a type at all?

We always implement the return type of a method specified in the UML with no return type as a void in C++.

(For a constructor, which of course is not a method, we would never specify a return type in C++ - the return type is an object, and is implicit.)

Q. How do I implement main.h?

If you choose to implement a main.h, you would implement it exactly as for the other classes. We don't normally provide a .h for the file containing the main function in C++, though, since it would be a little odd to include that file in another file. But nothing prohibits it.

Q. How do I deal with check_in and check_out methods in both the Publication and Library classes? Don't the conflict with each other?

No. An object instanced from the Publication class represents an individual book, magazine, DVD, etc. Think of one of the textbooks for this class. When you call check_out on this object, you are checking out the associated media - thus you only need to know the patron_name and patron_phone of the person who will temporarily take possession of the associated artifact.

An object instanced from the Library class represents a collection of books, magazines, DVDs, etc. Think of the UTA library, which contains a large collection of media. When you call check_out on this object, you must also specify which specific publication to check out from the library.

Similarly, calling check_in on a publication object requires no parameters - it's checking in the publication associated with that object. But calling check_in on the library object requires that you specify as a parameter which publication in the library is being checked back in.