

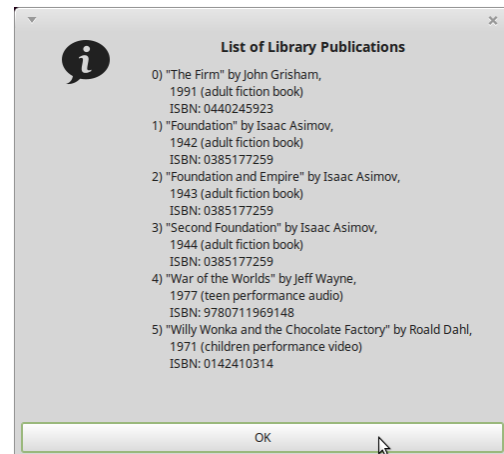
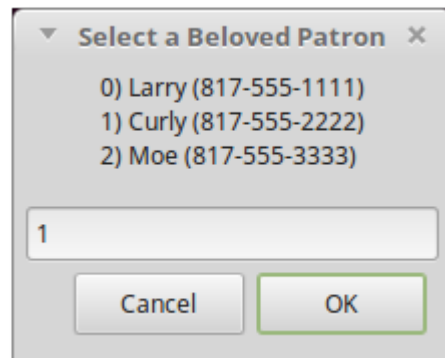
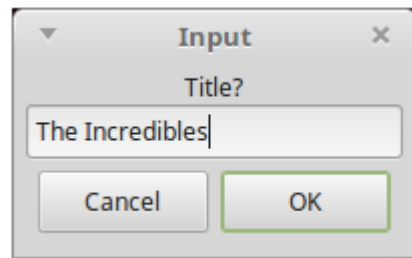
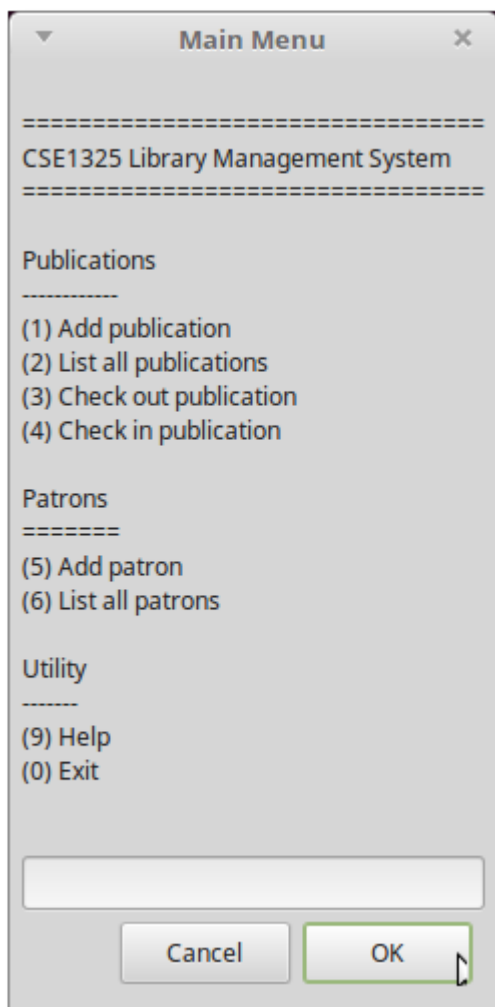
Getting Organized – Library Model

Sprint #2

CSE 1325 – Spring 2018 – Homework #6

Due Thursday, March 8 at 8:00 am

Command Line Interfaces (CLI) can be very productive for technical people, but are not acceptable to normal users. They prefer rich, graphical interfaces, and our job as professional software developers is to provide interfaces that delight our users. But not this week. Instead, we'll start by implementing a literal replacement of the console menus and inputs with the pre-defined gtkmm dialogs that we developed in class.



Full Credit

Port your or one of the suggested solutions (the bonus is recommended) to an all-GUI, dialog-only implementation. That is, every cin becomes a Dialogs::input or Dialogs::question, and every cout a Dialogs::message or Dialogs::image.

For this sprint, you **ONLY** need use the pre-defined gtkmm dialogs discussed in class and attached in source code form to Lecture 13. Or you may write directly using the gtkmm class library if you prefer.

Attached find a new Scrum spreadsheet with the full set of features in the Product Backlog – for Sprint #1 (completed last time), Sprint #2 (for this week), and Sprint #3 (for next week). **Copy your “Sprint 01 Backlog” tab (or copy its data) from last week’s spreadsheet to this one.** Then **do your sprint planning for this week on the “Sprint 02 Backlog” tab.**

As in Sprint 01, you will be graded on the extent to which you cover the requirements and the quality of your code. As always, use git to version control your software. **You will receive partial credit if you only implement a portion of the requirements**, so don’t hesitate to submit partial results early. 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 ever used a library. **If you have questions on *anything*, ask!!!**

Suggested Approach

I recommend the following approach for this sprint.

1. “Baseline” your CLI version in git. If you’re building from your own code, you need do nothing, but if you adopt a suggested solution, **put it under git as is FIRST.**

Then **build and test the resulting CLI application using your Makefile.**

(a) Each of the suggested solutions included a script named “**test_all**” (run with ./test_all, of course) that will “make clean” and then build and run all tests. You should see no output other than the g++ commands from make, in which case all is well.

(b) Use just “make” with a suggested solution to **build the “lms” executable**, and then test it interactively (./lms). You can use “make debug” if you run into issues and need to run the debugger (it will execute a “make clean” first).

You must start with a solid, working CLI application before making any gtkmm changes.

2. Add the necessary **gtkmm header and initialization to main.cpp** – that’s “#include <gtkmm.h>” and “Gtk::Main kit(argc, argv);” right after “int main”. Verify that the program still compiles and runs, remembering to add ``/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`` to your compile commands in your Makefile (note those **backticks**, not apostrophes, are *required!*). It should behave identically to the pre-modification version.

3. Now, make the necessary change for *just the main menu* to work as a dialog box. Remember – **baby steps!** (Hint: If you followed the Model – View – Controller pattern, then **the ONLY classes you should need to modify are Controller and View.**)

How you do this depends on your code. If you use the suggested Bonus solution from P4, consider modifying “void View::show_menu()” to “int View::select_from_menu()”, modifying the implementation to use Dialogs::input(menu) and converting the user’s input to an int using std::stoi(string). Remember to handle the case where the user presses the Cancel button (Dialogs::input will return “CANCEL”, remember?), the case where the user enters something other than an integer (stoi will throw an invalid_argument exception, so try / catch is your friend here), and the case where the user selects an integer not in the menu (pop up an error using Dialogs::message, then try again).

Build and interactively test your modification. See the FAQ on common error messages and how to address them, to save time and frustration. **The debugger is your friend.** Don't proceed until it works well, and you understand *why* it works well.

4. Now, **one at a time, make the necessary changes for each remaining console prompt or report** – in essence, each of the use cases – testing carefully and adding to git after each update. This is where your Sprint Backlog (your task list) is handy, as you can list out each change you need to make in advance, then check them off as you go.

You may want some additional changes to View so that its methods return the user selection. For example, instead of “void View::show_publications()”, you might have a private “string View::get_publications()” that just returns the list of publications as a string, a public “void View::list_publications()” that displays that list in a Dialogs::message window, and a public “int View::select_publication()” that displays the list and obtains the user’s selection via a Dialogs::input window, using code very similar to our “int View::select_from_menu()” method above (and handling the same exceptional cases, too).

Also note that your test_view.cpp regression test (if you had one – the suggested solution did!) likely no longer makes sense, as testing GUIs is usually an interactive affair. Don’t forget to delete it and update your Makefile. Your other regression tests *shouldn’t change at all* as long as you separated I/O from the model, as we have (repeatedly) suggested.

Remember – **baby steps!** Address one use case at a time, get it compiled and working well, add it to git, and move on. **Remember to make periodic backups**, too – don’t get near the end and lose all of your work!

Once every console prompt and report has been converted and stashed into git, you’ll have a really simplistic, but completely working, GUI!

Deliverables

As in previous assignments, you will deliver to Blackboard a **CSE1325_06.zip** file containing:

- Your **updated Scrum spreadsheet** named **Scrum_P6.ods**, **Scrum_P6.xlsx**, **Scrum_P6.lnk** or **.url** (containing a link to Google Sheets), or other spreadsheet format *that your grader agrees to accept* for this sprint at the root level – one file for all levels.
- A **full_credit subdirectory** containing:
 - Your **local git repository** including **.h** and **.cpp** files (including any **regression test files**, **dialogs.h**, and **dialogs.cpp** if you use them, all under git control) and a **Makefile** that is competent to rebuild only files that have been modified. Note that **we don't normally try to write regression tests for the GUIs themselves**, as this is quite difficult, thus you may omit those.
 - **Screenshot(s)** in PNG format as needed to help the graders know what to expect when they “make”.
 - (Optional) Your **library.xmi** representing your class diagram only if you modified your design since Sprint #1.

Bonus

Modify your main menu and help dialogs to use rich text – bold, italic, different fonts, whatever! This requires 2 steps.

First, you need to modify the Dialogs methods to enable Pango interpretation. You may remember that Pango is an HTML-like markup language for text you display in windows, but Pango interpretation is not automatically enabled. See <https://developer.gnome.org/pango/stable/PangoMarkupFormat.html> for how to enable this. Half of the points is getting this to work.

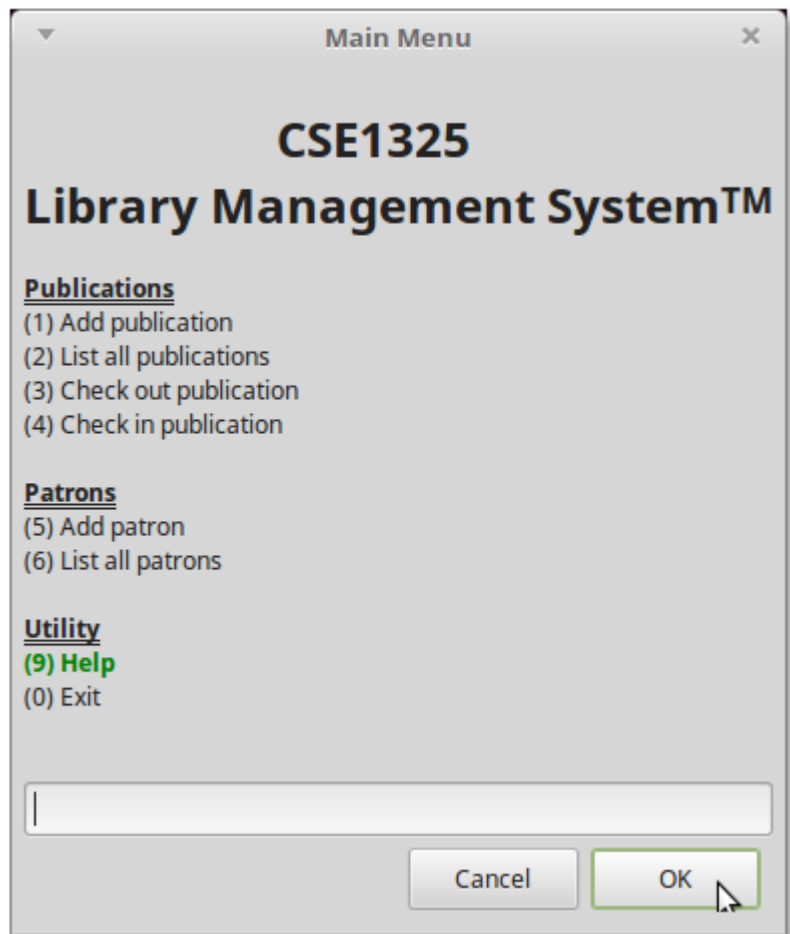
Second, you need to add the markup to your main menu and help dialogs to show off a bit for the remaining points. Getting the dialogs “right” is a matter of trial-and-error until you have some experience. Yours does **NOT** have to look anything like mine (as shown to the right). Explore and have fun!

You must (at a minimum) show 5 *different* tags across the dialogs, e.g., in my menu: bold, superscript, font size, font color, and double underline.

Deliverables

As in previous assignments, you will add to your **CSE1325_06.zip** the following:

- A **bonus subdirectory** containing:
 - Your **local git repository** including **.h** and **.cpp** files (including your improved **Dialogs** file, all under git control) and a **Makefile** that is competent to rebuild only files that have been modified.
 - Any **Screenshot(s)** that help the graders know what to expect when they “make”.



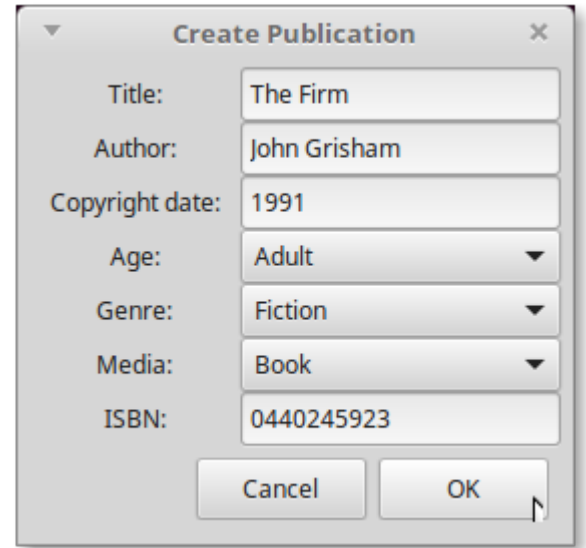
Extreme Bonus

Replace the series of dialogs collecting data for a new publication with a single, custom Add Publication dialog – a Dialog instance, with labeled input fields for title, author, copyright, and isbn, and drop-down lists for genre, media, and age. Include OK and Cancel buttons to accept the data or cancel the operation, respectively. Use the data to actually add the specified publication to the list.

One possible design is show to the right, but you are NOT required to design yours this way. Get creative!

You have some options for specifying the layout. For example, for the dialog to the right, you could use a 2x7 Grid container, placing each widget in its cell. Or you could put each Label and Entry / ComboBoxText in an HBox, then add each of them to the Dialog's VBox in turn. Or perhaps you could find a better way...

The remainder of the program need not change. We'll add a main window in Sprint 3.



The image shows a 'Create Publication' dialog box with the following fields and values:

Field	Value
Title:	The Firm
Author:	John Grisham
Copyright date:	1991
Age:	Adult
Genre:	Fiction
Media:	Book
ISBN:	0440245923

Buttons: Cancel, OK

Deliverables

As in previous assignments, you will add to your **CSE1325_06.zip** the following:

- An **extreme_bonus subdirectory** containing:
 - Your **local git repository** including **.h and .cpp files** and a **Makefile** that is competent to rebuild only files that have been modified.
 - **Screenshot(s)** demonstrating the custom Add Publication dialog as needed, with selections made for all fields similar to mine above, as images in PNG format.

Frequently Asked Questions

Q: I'm getting error "...". What causes that?

Good question! Here are some common errors, what they mean, and how to correct them.

**main.cpp:1:19: fatal error: gtkmm.h: No such file or directory
compilation terminated.**

You forgot to add ``usr/bin/pkg-config gtkmm-3.0 --cflags -libs`` to the end of your g++ compiler line for the indicated file (in this case, main.cpp) in your Makefile. That's the clause that tells g++ where to find all of the gtkmm-related header files.

**main.o: In function `main':
main.cpp:(.text+0x34): undefined reference to `Gtk::Main::Main(int&, char**&, bool)'
(with a LOT of other text!)**

You forgot to add ``usr/bin/pkg-config gtkmm-3.0 --cflags -libs`` to the end of your g++ linker line (the g++ line with all the .o files instead of .cpp files) in your Makefile. You made a reference to a Gtk+ resource that relies on a gtkmm-related compiled (.o) file, but didn't tell g++ where to find it.

undefined reference to `Dialogs::input' (with a LOT of other text!)

You forgot to add dialogs.o to your linker line (the g++ line with all the .o files instead of .cpp files).

Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.

This (legitimate) complaint by gtkmm shows up on cerr every time you pop up a dialog, complaining that you don't have a main window. As I mentioned, that's not really suitable for production, but we can ignore it for now. *Next* sprint we'll fix it!

If you'd like to not see it anymore, run your program like this (Linux / Mac only):

```
./lmi 2> /dev/null
```

This literally sends your output stream #2 (STDERR / cerr) to /dev/null, an output device that simply ignores all output in Unix-like systems. Yes, it's very useful at times. Note that it will also discard any error messages that YOU send to cerr - caveat scriptor!

Q. I want to add icons to my dialogs. Where do I get an icon to use?

Oh, anywhere you like – *as long as you respect copyright law*. Remember, we talked about this!

- 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 suggested solution icon (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 #4

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 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. Design something! Here are a few ideas to get you started.

Option #1: They are well-suited to be enums, with the enumerated values given in 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 some way to convert a value back to a string.

- Implement a (possibly private) method of Publication 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, e.g., "string genre_to_string(Genre genre);".
- Implement a "helper function" of the same name that exists in global space. This is u-g-l-y from an object-oriented programming perspective, though – we don't leave functions just floating around in global space (it's not even *possible* in Java!).
- Use a vector of strings, e.g., given "enum Color {red, green, blue}; vector<string> color_to_string = {"red", "green", "blue"};", you could convert a Color variable to a string using, e.g., "Color color = Color::red; cout << color_to_string[color];"

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). This is the most object-oriented solution.

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.

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, e.g., int main() { }.

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?

Since main.cpp just contains the main() function, which is invoked when you run the program, you don't need and shouldn't create a main.h file.

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

No. An object instantiated 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 instantiated 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.

If this worries you, consider using different names for each class' methods. That's not necessary, but it's certainly permissible.