COMP26020 – Assignment 2
Library Management Software in C++

A. Assignment

The goal of this assignment is to implement a set of C++ classes that aim to be used in a library management software. The library holds documents of different types (novels, comics and magazines) with various attributes (title, author, etc.) and users can borrow and return documents.

- `library.h` has the different classes definitions including inheritance relationships as well as the prototypes for all classes' methods.
- `library.cpp` contains the implementations of the classes.
- `test-suite.cpp`, `catch.h`, and `catch.cpp` implement a test suite that checks the correctness of the library code.

The code in your repo has everything already implemented and it's 100% functional and (mostly) correct!

Wait. What's the point of this assignment then?

Well, the code was written by someone who only watched some of the videos in the first week and then gave up. As such, the code is more C than modern C++, it does not use the standard library, and it does not follow any of the C++ design approaches. Ah, and it also has a few memory leaks. This is not good C++ code.

Your **actual goal** is to rewrite **`library.{cpp,h}`** using the concepts and approaches we covered in the lectures. This includes:

- RAII
- new/delete considered harmful
- C-arrays, C-strings considered redundant
- C++ standard library containers, utilities, and algorithms
- Improved type safety
- Using linters, static analyzers, and the C++ Core Guidelines to uncover errors and bad coding patterns

Your rewrite should not change the externally visible behaviour of the library classes: **any** possible external function calling **any** of the public member functions in `library.h` should behave the exact same way it did before the rewrite. In practice this means that:

a) You are allowed to edit only `library.h` and `library.cpp`
b) You are not allowed to remove existing public member functions in `library.h`
c) You are not allowed to change the visibility of existing member functions
d) You are not allowed to change the argument types or the return value types of public member functions in `library.h` (**with one exception, see below**)
e) Your member functions need to return the exact same thing they would have returned originally.

On the other hand, you are free to:

a) add new member functions (private, protected, or public)
b) change the internal implementation of any member function
c) Add/remove private member variables in `library.h` or change their types.

The only allowed change of types in function prototypes is between types that are implicitly convertible. For example, an int with allowed values 0/1 can be replaced by a bool. The testing code is also designed to treat all different kinds of strings as interchangeable. If you're not sure, test it.

B. How to approach the assignment

As is, the code is extremely unsafe. Try to use (modern) C++ capabilities that attack that problem first. If you do it correctly, you will be able to remove large parts of the code, eliminate some warnings, and multiple guidelines suggestions. This will make it much easier to identify any remaining issues and understand what is causing the various warnings.

A related advice is to keep thinking about whether you actually need every piece of code in your code base. If you can remove some function/loop/code-block without affecting the functionality of the program, remove it.

Test your code after every single change you make. Does it still compile? Does it still pass the tests? If you made a mistake somewhere, it's better to find out immediately rather than after another 5-10-20 changes.

Warnings from the compiler, clang-tidy and the static analyzer are often interconnected and multiple warnings are just symptoms of the same bad design choice. In such cases, try to eliminate the actual problem, not each symptom individually.

C. Building and testing

Test Suite

To run and test your implementation you are given a basic test suite in the form of a C++ file, `test-suite.cpp`. It's a C++ program (i.e. it contains a main function) that includes `library.h`. It instantiates/manipulates objects from all the library classes and performs sanity checks. This program uses the *Catch*[1] test framework and requires two additional source files to be compiled: `catch.cpp` and `catch.h`. You can compile the program manually:

```
$ g++ test-suite.cpp library.cpp catch.cpp -o test-suite
```

and execute the test suite with:

```
$ ./test-suite
```

You don't need to understand the testing code fully, but if you get failed tests, it might be useful to check the code that is associated with the failed test in order to understand what went wrong. Overall, the suite is divided into tests cases enclosed into `TEST_CASE() { ... }` statements. A test case will fail when one of the `REQUIRE( <condition> )` or `CHECK( <condition> )` statements it contains fails, i.e. when `condition` evaluates to false.

To complete the assignment you do not need to understand the content of `catch.h` and `catch.cpp`.

Passing all tests does not mean your code is 100% correct. Your solution is 100% correct only if the public behaviour of **all public methods** remains **exactly the same** under **all possible usage scenarios**[2]. When marking, we will test whether your code is correct with **a special test suite containing additional cases**. So, make changes carefully. If you are not certain about some change, add your own test case to test that change or discuss it with us in the lab.

Make

For your convenience, we provide a Makefile that can help you automate building and testing your code.

To build the whole test suite and check the library code for warnings:

```
$ make
```

To run the gcc static analyzer

```
$ make analyze
```

1 https://github.com/catchorg/Catch2

2 You can assume that input arguments are always valid though: no null pointers, number values are within valid ranges, filenames are legal, etc.

To run the C++ core guidelines checker and parts of the clang static analyzer:

```
$ make guidelines
```

You can change the three variables at the top of the Makefile to match your setup. CXX should point to your C++ compiler. Make sure it's one from the last 3 years, supporting C++ 17. Also, note that `make analyze` will only work with g++>=10.0. Use your compiler's static analyzer, if you're using another compiler. TIDY chooses which clang-tidy to use. The default is to use one in the system path.

Workflow options

1. Lab workstations locally

The lab Linux workstations are already setup in the exact same way as the machine where your code will be tested. So, it's the easiest option to use and you will be 100% certain that if your code runs correctly for you, it will run correctly for us too.

2. Lab workstations remotely

Same advantages as above but you can connect to them from home (through VPN) or from other university locations. You will need an ssh client, such as openssh (Linux) or PuTTY (Windows). To connect set the remote hostname to one in the range `e-10cki18001.it.manchester.ac.uk` to `e-10cki18060.it.manchester.ac.uk`. For example in Linux you can do:

```
$ ssh -Y <username>@e-10cki18001.it.manchester.ac.uk
```

If the command says that it cannot find a route to the host, try a different hostname in the valid range.

3. Your own Ubuntu Installation (VM or not)

For Ubuntu 22.04 (preferred):

```
$ sudo apt install make g++-11 clang-tidy-12
```

For Ubuntu 20.04:

```
$ sudo apt install make g++-10 clang-tidy-12
```

and edit Makefile to say CXX := g++-10

With these changes you should be getting results similar to what you would get in the lab. But keep in mind that we will not be able to help you if you run into any problems caused by any differences in the setup.

4. Other systems

This is the riskiest option. We will not be able to help you with any technical issues. It's also possible that your system displays different behaviour than the lab workstations. For example, we have seen multiple cases where the submission does not compile on our side because of a missing header file, but does compile on the student's side because their compiler indirectly included that header. So, be aware of the risks and try to test your code on a lab workstation before submission.

The main thing to do is to install an appropriate compiler and clang-tidy. After that set CXX and TIDY in the makefile to point to your compiler and clang-tidy respectively.

Building and running the test suite only requires a relatively modern C++ compiler so this should be easy to get working correctly. make analyze will only work with g++ > 10.0. If you don't have access to an appropriate version, check whether your compiler has a static analyzer you can use. make guidelines requires clang-tidy. If you're using Visual Studio, you could use the C++ Core Check instead.

If you don't have a recent g++ or clang-tidy, consider connecting to the lab workstations and testing your code there following the instructions above.

D. Submission

Deliverables, Submission & Deadline

There are two deliverables: the completed `library.cpp` and `library.h` files. The submission is made through the CS Department's Gitlab. You should have a fork of the repository named "*26020-lab2-S-CPlusPlus_<your username>*". **Make sure you push to that precise repository and not another one**, failure to do so may result in the loss of some/all points. Submit your deliverables by pushing the corresponding files on the master branch and creating a tag named **`lab2-submission`** to indicate that the submission is ready to be marked.

The deadline for this assignment is 08/12/2022 6pm UTC.

Submission Checklist

1. Does your submission compile successfully **on the lab machines**?
2. Does your code **pass all tests**?
3. Have you c**ommitted all local changes**? Check using git status.
4. Have you pushed your changes to the **right repository**?
5. Have you tagged your **latest** commit with "**lab2-submission**"?

Marking Scheme

The exercise will be marked out of 10, using the following marking scheme:

- The program is functional, and passes the basic test suite /1
- The program passes all extended tests /1
- The code takes advantage of C++ capabilities to make the code clearer and more concise /1
- The library code follows RAII principles /1
- The library code uses Standard Library containers and data types whenever appropriate /2
- The library code uses Standard library algorithms whenever there is a clear benefit from doing so /1
- The library code does not produce significant warnings when compiled with `-Wall, -Wextra`, -Wpedantic, or the static analyzer and it does not suffer from memory errors /1
- The library code follows the C++ Core Guidelines (at least the ones that we discussed in the lectures, as well as the ones tested by clang-tidy) /1
- The code is clear, well commented, and follows good practices /1

Plagiarism/Collusion

This is an individual coursework exercise. You should not share your code with others, you should not try to see or copy the source of anyone else, and you should not work together with other people. Exchanging ideas is also collusion[3] and it will definitely treated as such if the submissions are very similar.

All submissions will be checked using a standard code plagiarism detection program, as well as a checker tailored to this assignment. We will manually examine any submissions that are flagged. If their similarities are because they contain the code we gave you or because they applied the same obvious changes, then there will be no further action. If this is not the case and their similarities are extremely unlikely to be due to chance, they will either be penalised or forwarded to a disciplinary panel.

CS Guidance UoM-wide Guidance Academic Malpractice Procedure

3 Academic Malpractice Procedure – 3.1.2.3:"The methods of collusion may include, but are not limited to, sharing of work, ideas or plans by social media or other electronic communication means, and/or physical sharing of work, ideas or plans."

E. FAQ (Check blackboard for updates)

0. I have the X problem on my Y laptop	If you are not confident that you can fix your problems easily, just connect to the lab workstations and work there. This is the point of lab workstations to begin with: a single hardware/software configuration where the code is guaranteed to run and you can be confident that we will get the same results you get when we test your code.
1. "unrecognised command line option -fanalyzer"	Do you have g++ > 10 installed? If you don't have g++ on your platform, find the static analyzer of your compiler or try the g++ static analyzer installed on the lab workstations
2. command g++-11 not found	You don't have g++-11. Either install it or modify CXX in the Makefile to point to the binary of your preferred compiler
3. Do I need make guidelines and make analyze?	No, you can identify all code issues without using the two tools. But why make your life difficult? Just use the lab workstations.
4. Do I need to eliminate all warnings from make analyze and make guidelines?	Probably not. Some warnings might be bogus. Use your judgement. Also suppressed warnings in make guidelines are about library code, so it's beyond the scope of this coursework
5. If I eliminate all guideline warnings, will I get the full mark?	Probably not. clang-tidy does not enforce all guidelines, so there will be issues with your code that it will not complain about.
6. How will I know that I have fixed the code? What does the perfect solution look like?	There is no way of telling you that without giving you an exhaustive list of what you need to do. But if you attack the big problems that I kept talking about in the lectures (i.e, "new considered harmful", RAII, "C-arrays considered redundant", avoiding explicit bounds checking), you should already be close to a perfect solution.
7. Can I modify the argument types of any member functions?	Normally no. Straightforward substitutions like bool instead of an integer that holds binary values, or std::string / std::string_view instead of c-strings are normally okay. The testing code is written in a way that it will handle such cases gracefully. Other substitutions might change the public interface and thus break the testing code. If you just want to change how you use a function internally, just overload the function with a version that has your preferred argument types.
8. "cc1plus: warning: analysis bailed out early"	This only indicates that the code is too complex for the static analyzer. You can ignore it.
9. Can I mix raw and smart pointers?	Yes. Smart pointers are meant to be used only for owning data. Replacing non-owning raw pointers with smart pointers is usually wrong.
10. Can we add a destructor for class X, even if it was not originally declared in library.h? Can we remove the destructor for class Y, even if was originally declared? Can we remove a copy/move constructor/assignment operator?	Yes, yes, and yes. The class has the set of five special methods whether you declare them explicitly or not (unless you set them to deleted of course). Assuming you don't do anything that affects visibility, there is no change to the public interface. The only change is whether the special methods are explicit or not
11. Can we remove/modify/add private variables?	Yes. The testing code cannot see private variables or functions, so any changes you make cannot affect the tests.
12. The test suite reports something like: test-suite.cpp:201: FAILED: REQUIRE(some_fnct()) with expansion: false	REQUIRE checks whether whatever is in the following () evaluates to true or 1. The message indicates that some_fnct() returns the wrong value. Try to figure out why.
13. Can I change the arguments/return values/methods to const?	Definitely not the return values. External code might depend on them being non-const, even if they are not meant to be modified. Modifying arguments and methods should be okay.
14. Can I change an argument from pointer to reference?	NO! Pointers and references have similar properties but they are not syntactically interchangeable. If I call a function passing it a pointer and you change that function to get a reference, then the testing code will not compile correctly.

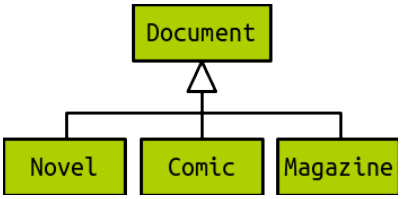
15. When I run make guidelines I get this error: clang-tidy: not found	The makefile cannot find the system clang-tidy. Install one.
16. Can I cut everything in library.h and paste it in library.cpp?	NO! This will break make and modular compilation in general, i.e. your code will not compile when tested. Also it's an awful coding practice.
17. Can we use standard library header X/Y/Z?	You are allowed to use any standard library header you want, as long as that header exists in C++ 17. It's not a great idea to use some of these headers, but it will not break the testing code.
18. "No space left on device"! What happened?	You've ran out of space. If you are using a lab machine, you don't have unlimited space in your university's home folder. Use **du -sm** * from the command line in your home folder to figure out what's using all your space. If you see something that you don't need and it takes a lot of space, delete it and try again.
19. My tagged commit did not modify library.{cpp,h} but an earlier commit did. Will this cause a problem?	No, it will be fine. Our code will clone your whole repo at the point in time you tagged it, not just the files you committed when you tagged it. Out of this clone, we will copy library.{cpp,h} into our own folder where we will build it with our own Makefile.

E. Appendix

Class Hierarchy

The `library.h` header defines 5 classes which are briefly presented below. Note that more detailed information about the methods is present in the header's source code in the form of comments.

- `Document` is an abstract class defining attributes and methods common to all the documents that can be held in the library. Attributes include the document's *title* and *quantity* held in the library. It defines various methods for printing the document's information on the standard output, getting its concrete type (novel/comic/magazine), and various getters/setters including methods for borrowing/returning the document from/to the library.
- `Novel`, `Comic` and `Magazine` represent the concrete types of documents. Each inherits from `Document` as depicted on the figure on the right. They differ slightly in attributes: a novel and a comic have an *author*, while comics and magazines have an *issue number*. Each class also defines the relevant getters/setters.



Document class hierarchy

- The last class, `Library`, represents the library i.e. a collection of documents. The documents are held in an array of Document pointers. The library class defines various methods for actions such as adding, removing, searching, borrowing, returning documents, printing the library content on the standard output or dumping it in a CSV file.

Documents/Library Printing and CSV Output Formats

The `print()` method, when called on a novel, prints on the standard output the novel's attributed in this format:

```
Novel, title: Monstrous Regiment, author: Terry Pratchett, quantity: 1
```

For a comic:

```
Comic, title: Watchmen, author: Alan Moore, issue: 1, quantity: 10
```

And for a magazine:

```
Magazine, title: The New Yorker, issue: 1, quantity: 20
```

The `print()` method called on a library containing these 3 documents produces:

```
Novel, title: Monstrous Regiment, author: Terry Pratchett, quantity: 1
Comic, title: Watchmen, author: Alan Moore, issue: 1, quantity: 10
Magazine, title: The New Yorker, issue: 1, quantity: 20
```

The `dumpCSV()` method called on the same library produces a file with the following format:

```
novel,Monstrous Regiment,Terry Pratchett,,1
comic,Watchmen,Alan Moore,1,10
magazine,The New Yorker,,1,20
```