1. **the difference between the implementations/utilizations of a functor and a lambda expression. When is appropriate to use each one?**

In C++, a functor refers to an object whose class has an operator() overload. With this overload, the object can act as a function. The implementation of a functor, as mentioned above, requires an overload of the operator() of the class, for example:

//SpellChecker.h

class SpellChecker {

std::string m\_badWords[6]{};

std::string m\_goodWords[6]{};

int m\_foundCnt[6]{};

public:

…

void operator()(std::string& text); // overloading operator()

//SpellChecker.cpp

void SpellChecker::operator()(std::string& text) {

size\_t pos{};

for (int i = 0; i < 6; i++) {

do {

pos = text.find(m\_badWords[i]);

if (pos != std::string::npos) {

text.replace(pos, m\_badWords[i].length(), m\_goodWords[i]);

m\_foundCnt[i]++;

}

} while (pos != std::string::npos);

}

}

A lambda express refers to a function that is defined ad-hoc, can be with or without a function name. For example:

<from workshop>

While functors are defined as classes, lambda expressions are functions that are defined on the fly. This means functors can be reusable, i.e. when an object needs to be used multiple times, whereas lambda expressions are good for when the function is only needed once.

//define a lambda expression

auto adjustPrice = [usdToCadRate, gbpToCadRate](sdds::Book& book) {

if (book.country() == "US") {

book.price() \*= usdToCadRate;

}

else if (book.year() >= 1990 && book.year() <= 1999) {

book.price() \*= gbpToCadRate;

}

};

//call the function

for (size\_t i = 0; i < library.size(); i++) {

adjustPrice(library[i]);

}

1. **the constructor for SpellChecker generates an exception in certain conditions. How would you change your code to achieve the same result, but without exceptions (both on the constructor and on the client side)? Compare the two solutions.**

In the SpellChecker constructor, to implement the exception without using the “thow” keyword and “try – catch” statements, I would create a member variable of type bool “m\_active” to keep track of the active status of SpellChecker. If the constructor successfully retrieves data from the file to get goodWords and badWords, m\_active will be true, or false otherwise. Additionally, if the file cannot be found, the constructor will also cout a message “Bad file name!” just like how it is done using the “throw” keyword. Furthermore, all member functions of SpellChecker will now check the state of the object (using m\_active) to make sure it is active before implementing any logic. For example, in operator(), prior to finding badwords and replacing with goodwords, it will check to see if m\_active is true or not to prevent unexpected behaviours.

Although it is still possible to accomplish the task by not using “throwing” and “try – catch” statements, the code can be redundant and error-prone as the active state will need to be checked for all logic. With using “throw” and “try – catch”, we make sure the thrown exception in the “try” code block is being caught properly and any logic afterwards is not going to be executed; hence, our code will be more readable and easy to debug.

//SpellChecker.cpp

SpellChecker::SpellChecker(const char\* filename) {

fstream inf(filename);

string str{};

if (inf) {

for (int i = 0; i < 6; i++) {

getline(inf, str);

m\_badWords[i] = str.substr(0, str.find\_first\_of(" \f\n\r\t\v"));

m\_goodWords[i] = str.substr(str.find\_last\_of(" \f\n\r\t\v") + 1);

}

}

else {

throw "Bad file name!"; // exception will be thown and caught in “try-catch”

}

//w5\_p2.cpp

try {

SpellChecker sp(argv[i]); // if the file cannot be found, exception will be caught in “catch”, below code will not be executed.

for (auto j = 0u; j < library.size(); ++j)

library[j].fixSpelling(sp);

sp.showStatistics(std::cout);

for (auto j = 0u; j < theCollection.size(); ++j)

theCollection[j].fixSpelling(sp);

sp.showStatistics(std::cout);

}

catch (const char\* err) { //exception is caught here and error message is printed

std::cerr << "\*\* EXCEPTION: " << err << std::endl;

}

1. **the classes Movie and Book contain almost identical logic in loading data from the file. How would you redesign the classes in order not to duplicate that logic?**

Since Book and Movie have a some similar member variables and functions, they can be derived from a base class (called “Media”). This base class is responsible for reading the data and extracting token from the argument string and storing only the common ones for all type of media (i.e., title, description, year, etc.). The derived classes (Book and Movie) will inherit the common data and continue to read and save the rest of the data from the string to their member variables (i.e., author, country and price for class Book). Similarly, the base class will have query functions like title(), display(), fixSpelling() function template so that the derived classes can inherit from.

1. **the classes Movie and Book are instantiated from the main() function using a custom constructor, but they also contain a default constructor. Is the default constructor necessary? Could you remove it? Justify your answer.**

The default constructor is required here because other constructors of the class have been defined. In C++, if any constructors other than the default one have been defined, the compiler will not create a default constructor. Then, if an object of that class is created using the default constructor (have no arguments, i.e., Book\* book = new Book()), the compiler will give error since it can not find the defined default constructor.