This work shop is the second hardest workshop I had to deal with through this course, and by doing this workshop I was able to go over almost every single algorithm prof listed during the lecture. Moreover, I self-studied more algorithms from the C++ reference website. Although it is work load is pretty ridiculous, I finally learned some real hardcore C++ stuff, which I am so very happy about this.

1. list all the STL functions that you have used in this workshop; explain your reasoning for choosing each one, and not another.

I will do this function by function:

1. I used 2 algorithms in display function:

void CovidCollection::display(std::ostream& out) const{

std::for\_each(m\_collection.begin(), m\_collection.end(), [&out](const Covid& theCovid) {out << theCovid<<"\n"; });

size\_t CaseResult{};

size\_t DeathResult{};

CaseResult = std::accumulate(m\_collection.begin(), m\_collection.end(), 0ull, [](size\_t acc, const Covid& src2) {return acc + src2.m\_caseNum; } );

DeathResult = std::accumulate(m\_collection.begin(), m\_collection.end(), 0ull, [](size\_t acc, const Covid& src2) {return acc + src2.m\_deaths; });

out << "----------------------------------------------------------------------------------------\n";

out << "|"<<std::setw(78)<<"Total Cases Around the World:"<<" " << std::setw(5) << CaseResult << " |" << std::endl

<< "|"<< std::setw(78) <<"Total Deaths Around the World:"<<" " << std::setw(5) << DeathResult << " |\n";

}

First one is the for each used to print out every single record. In C++ for\_each algorithm was categorized into the query type, so I assume I should not use this algorithm as a modifier or other uses. For each is the best way to loop through the vector and print out single record as far as I can come up with.

Second one here is the accumulate algorithm from number library. This one is just making my life easier because I do not need to use for each again to loop through the vector and combined with the other algorithms to get the total number again.

1. I used one algorithm here.

void CovidCollection::cleanList(){

// overload an assignment operator for Covid class//

std::replace\_if(m\_collection.begin(), m\_collection.end(), [](const Covid& src) {return src.m\_variant == std::string("[None]"); }, std::string(""));

}

This replace\_if is pretty amazing when doing this kind of task, at first I had some problems to let the empty string to replace the Covid record’s member attribute. After understanding part of the logic, I release that I could overload the assignment operator for the covid struct. I was so happy to make this work by myself without anyone hints me.

1. The sort in the sort function reason is kind of obvious so I would not put the code example here.
2. I used one algorithm here

bool CovidCollection::inCollection(const char\* VName) const{

std::string paraAdapter (VName);

return std::any\_of(m\_collection.begin(), m\_collection.end(), [=](const Covid& src) { return src.m\_variant == paraAdapter; });

}

First thing I see is this function returns a Boolean value and also this is a query function. This function is designed for find out if the vector has any of that variant. So I genially think the any\_of will do the work. Combination of for\_each and find\_if will do the work too. But why bother? Picking the low hanging fruits, right?

1. I used to have four, but now I reduced this to 2.

std::list<Covid> CovidCollection::getListForCountry(const char\* countryName) const {

std::string paraAdapter(countryName);

std::list<Covid> resList(std::count\_if(m\_collection.begin(), m\_collection.end(), [=](const Covid& src) {return src.m\_country == paraAdapter; }));

// learned the copy\_if algorithm from C++ reference.

std::copy\_if(m\_collection.begin(), m\_collection.end(), resList.begin(),

[=](const Covid& srcCollection) {return !srcCollection.m\_country.compare(paraAdapter); });

return resList;

}

Firstly, when declaring the size of the returned list, I used the count\_if to get a size for creating the result list. Before I thought different container could not use in the same manipulation algorithm. But after I released that the problem was not that. I could reduce the number of the algorithms involved there.

Second algorithm is copy\_if. This one I learned from the C++ reference website. The predicate is used to determine whether we should copy the value into the destination container, which saves so much logic for completing this. I could have use the copy the whole thing over and then trimming the unwanted element one by one with the other algorithm. But I would say why bother. This is already simple enough.

1. I used 2 algorithms here:

std::list<Covid> CovidCollection::getListForVariant(const char\* variantName) const {

std::string paraAdapter(variantName);

std::list<Covid>resList(m\_collection.size());

std::transform(m\_collection.begin(), m\_collection.end(), resList.begin(), [=](const Covid& src) {Covid res{}; if (src.m\_variant == paraAdapter) res = src; return res; });

resList.erase(std::remove\_if(resList.begin(), resList.end(), [](const Covid& src) {return src.m\_country == ""; }),resList.end());

return resList;

}

First of all, this time, instead of creating a proper size of the final result list. I firstly use the m\_collection’s size to create this container.

Then I looked at the prof’s code example and came up this one.

The transform will convert the value from the vector into the destination container based on the predicate condition. I made this returning the empty value if it does not satisfy the condition.

Then I used remove\_if ( learned from C++ reference website to finish trimming the result list. So the list would be resized and return back. I was kind of out of ideas at that moment so this is the only option I have at that stage.

1. present the advantages of using STL algorithms vs. implementing the functionality by yourself using loops.

After forcing myself to use algorithms from STL to complete this workshop, I can see why those OG coders prefer the algorithms than the manual loop.

First of all, the code is so clean and concise. I was able to finish all the tasks in a few lines. I feel most of the coders should pursuit writing their codes in a clean and short way as long as it is still readable.

Secondly, I am assuming the run time speed would be faster by using these algorithms. Think about this, these algorithms are written by all the industrial pioneers. They already have the way to build a perfect wheel. If we have to reinvent it( by implement these logics again in our way), we must have to experience the stage when we invented a square wheel and trimming it here and there to make it runnable. Isn’t that time consuming by using the pioneers’ logics? Are we all that confident about our logic is faster than those OG coders? Maybe not right now.

1. the function that sorts the collection of covids receives as a parameter the field name to use; this can create errors if the client specifies an invalid field (or misspells it). How would you redesign that function to prevent the client to specify an invalid string, allowing any error to be detected by the compiler?

My current design is like this:

void CovidCollection::sort(std::string keyWord){

if (keyWord=="country") {

std::sort(m\_collection.begin(), m\_collection.end(), [](const Covid& src1, const Covid& src2) {return src1.m\_country < src2.m\_country; });

}

else if (keyWord == "variant") {

std::sort(m\_collection.begin(), m\_collection.end(), [](const Covid& src1, const Covid& src2) {return src1.m\_variant< src2.m\_variant; });

}

else if (keyWord == "cases") {

std::sort(m\_collection.begin(), m\_collection.end(), [](const Covid& src1, const Covid& src2) {return src1.m\_caseNum < src2.m\_caseNum; });

}

else if (keyWord == "deaths"){

std::sort(m\_collection.begin(), m\_collection.end(), [](const Covid& src1, const Covid& src2) {return src1.m\_deaths < src2.m\_deaths; });

}

else{

std::cout << "You enter some wrong stuff\n";

}

}

But now if you want the compiler detect the error, we should use the throw and try and catch structure.

In the else block, I could just use the throw “You have entered the wrong keyword”;

In the main code, I need to implement the try and catch like this.

try{

theCollection.sort("country");

theCollection.display(std::cout);

}

Catch(const char\* err){

Std::cout<< err<< “\n”;

}

In this way the error would be showed the process will be ended before the program goes to display and the error message will be displayed.