This is supposed to be an easy workshop, but I did take some time on studying the algorithm and reinventing the wheel again.

But I don’t think all my efforts got wasted. First of all, I had a deeper understanding about the try, catch, and throw exceptions. Also, after used all my efforts, I finally released how efficient the smart pointers can be.

1. the advantage that smart pointer syntax provides. Use example from your code to support your explanation.

The advantages are very obvious. This is my writeRaw function in the WritingRecordModule.

GeneratingList<EmployeeWage> writeRaw(const GeneratingList<Employee>& emp, const GeneratingList<Salary>& sal) {

GeneratingList<EmployeeWage> activeEmp;

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

for (size\_t j = 0; j < sal.size(); j++) {

if (emp[i].id == sal[j].id) {

EmployeeWage\* temp = new EmployeeWage(emp[i].name, sal[j].salary);

try {

temp->rangeValidator();

if (sal.LuhnsAlogrithm(sal[j].id)) {

activeEmp += temp;

delete temp;

}

else {

throw "Your Luhn sucks lol\n";

}

}

catch (std::string err) {

// intercept the err and throw it again//

// Because you want to throw the err to the main

// And you want to deallocate too.//

delete temp;

throw err;

}

}

}

}

return activeEmp;

}

To avoid the memory leak, I did have repetitive statements everywhere to avoid the memory leak . Also, I had to use a catch block to be a exception forwarder and memory cleaner when there is an exception got thrown.

By using the unique pointer, my code gets much cleaner than the above function.

GeneratingList<EmployeeWage> writeSmart(const GeneratingList<Employee>& emp, const GeneratingList<Salary>& sal) {

// The only difference is now you don't need to manually free the memory anymore.//

GeneratingList<EmployeeWage> activeEmp;

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

for (size\_t j = 0; j < sal.size(); j++) {

if (emp[i].id == sal[j].id) {

std::unique\_ptr<EmployeeWage> temp (new EmployeeWage(emp[i].name, sal[j].salary));

temp->rangeValidator();

if (sal.LuhnsAlogrithm(sal[j].id)) {

activeEmp += temp;

}

else {

throw "Your Luhn sucks lol\n";

}

}

}

}

return activeEmp;

}

Because I used the unique pointer, the memory leak won’t be worried because the destructor will automatically clean the memory. Therefore, I do not need to implement unnecessary logics to forward the exception and clean up the memory in the try and catch structure, which saves a lot of efforts. I am currently studying more uses regarding the smart pointers, and I will make sure ask a lot of questions regarding this topic during the lecture.

1. the difference between raw and smart pointer syntax in your solution.

First let’s use the above code as example:

Raw pointer version:

EmployeeWage\* temp = new EmployeeWage(emp[i].name, sal[j].salary);

This pretty much means I let the new declared pointer pointing to a pointer returned by new operator which contains a EmployeeWage object created by the constructor.

Smart pointer version:

std::unique\_ptr<EmployeeWage> temp (new EmployeeWage(emp[i].name, sal[j].salary));

First I used the template of unique\_ptr supplied with the data type as EmployeeWage, name is temp and called the constructor supplied with the argument which is unique address of the resource created by the EmployeeWage’s constructor.

Second let’s look into the += operator overload:

// Very delicate process, it has to be a reference to a unqiue pointer then it will work.

// Otherwise, it won't compile//

void operator += (std::unique\_ptr<T>& srcPtr) {

// dereference //

list.push\_back(\*srcPtr);

}

//TODO: Overload the += operator with a raw pointer

// as a second operand.

void operator += (T\* srcPtr) {

// dereference //

list.push\_back(\*srcPtr);

}

It is pretty obvious that the only difference is, in the smart pointer version, I have to use the reference. But what’s the reason though?

Remember we learned that unique\_pointer class disabled the copy assignment operator and copy constructor. So you can not directly pass a unique pointer by value like what I did for the raw pointer.

Instead we could use passing by pointer or passing by reference. But to be honest, passing by pointer of a unique pointer is kind of complicated. Passing by reference is simple enough for me. That’s why I choose to use this way.