This is the probably most interesting workshop ever so far speaking of my Seneca experience. Every time I got excited at first and tortured right after. It literally explains what is called coding takes 2 hour, but patching leaks takes two day.

Anyways, the biggest take away is the DMA when it comes to an array of pointers. Also, the class relationship must be considered when we are implementing DMA.

1. Explain in the reflection what is the relationship between *Child* and *Toy*, and what is the relationship between *Child* and the ***array*** of pointers.

The relationship between Child and Toy would be composition. To be more specific, the Child class has a const member attribute called m\_Tppr, and it is pointing to a pointer under the Toy class. The code below would be my reference.

namespace sdds {

class Child {

const sdds::Toy\*\* m\_Tppr{};

size\_t m\_Counter{};

int m\_Age{};

std::string m\_Name{};

void deallocate();

….

By saying the composition, the Child class has the full control of the life time of each object pointed by the pointers inside the array of pointers, which means we are required to create and destruct the resource under the Toy type.

Take the code from my Child.cpp as an example:

This is a private method I created to deallocate the resource

void Child::deallocate(){

// class relation is composite//

// Clean up needs to reach 2nd depth//

for (size\_t i = 0; i < m\_Counter; i++){

delete m\_Tppr[i];

}

delete[] m\_Tppr;

}

This is the three argument constructor:

Child::Child(std::string name, int age, const Toy\* toys[], size\_t count){

m\_Name = name;

m\_Age = age;

m\_Counter = count;

if (toys&&count){

// since this is a constructor, there is no memory allocated yet for the member pptr//

// so do not deallocate//

m\_Tppr = new const Toy\* [count];

// deep copy//

// this is honestly same like what I did for the OOP beginner class project//

// Since Toy Class has no resources involved//

// using the compiler inserted constructor to build a nameless object on a dynamic mem//

// then assign it to the arr element//

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

m\_Tppr[i] = new Toy(\*(toys[i]));

}

}

}

As you can see, I am not only allocating the memory for the dynamic array but also allocating the memory for each pointers from that array to actually achieve the deep copy. Inside the destructor, I have to also take deallocate both level of memory I allocated.

1. Explain in the reflection what is the relationship between *ConfirmOrder* and *Toy*, and what is the relationship between *ConfirmOrder* and the ***array*** of pointers.

The relationship between ConfirmOrder and Toy would be an aggregation. Again, in the ConfirmOrder.h, there is a member attribute looks same like the one from the Child module.

namespace sdds {

class ConfirmOrder {

const Toy\*\* m\_Tppr{};

size\_t m\_ArrSize{};

…

But this time, this member attribute is working quite differently compared to the one from the Child module. Because the class relationship now is aggregation. So it pretty much means you can do anything to Toy object pointed by the array element pointers EXCEPT allocating memory and deallocate it( directly controlling the lifetime). However, as I asked prof, we are allowed to manipulate the resource pointed by the m\_Tppr. That is why we can resize the array but not manipulate the resources pointed by the elements of that array.

Taking my code as an example:

This is where I implemented resizing the array for the += operator:

….

if (m\_ArrSize){

// keep all the existing records in the temp//

const Toy\*\* temp = new const Toy \* [m\_ArrSize+1];

for (size\_t j = 0; j < m\_ArrSize; j++) {

temp[j] = m\_Tppr[j];

m\_Tppr[j] = nullptr;

}

// resize ptr arr pointed by m\_Tppr//

// deallocating//

delete[] m\_Tppr;

// Now you can increase the size//

m\_ArrSize++;

// allocating new memory for the ptr array//

m\_Tppr = new const Toy \* [m\_ArrSize];

// Swapping right back to the member pointer//

for (size\_t j = 0; j < m\_ArrSize; j++){

m\_Tppr[j] = temp[j];

temp[j] = nullptr;

}

delete[] temp;

// Finally shallow copy the resource onto i index//

m\_Tppr[i] = &toy;

}

Again I did not go on the 2nd depth of this pointer to the array of pointer. So, the relationship between the ConfirmOrder class and the Toy class is still aggregation.

1. the difference between the implementations of move and copy functions in a composition and an aggregation.

Let’s start from sharing the code of copy operator overloading and mover operator overloading.

Composition copy

Child& Child::operator=(const Child& src){

if (this!=&src){

// just wanna make sure arr of pointer has something inside//

if (src.m\_Tppr&&src.m\_Counter){

if (m\_Counter){

this->deallocate();

}

m\_Tppr = new const Toy\*[src.m\_Counter];

for (size\_t i = 0; i < src.m\_Counter; i++){

m\_Tppr[i] = new Toy(\*(src.m\_Tppr[i]));

}

}

m\_Age = src.m\_Age;

m\_Counter = src.m\_Counter;

m\_Name = src.m\_Name;

}

return \*this;

}

Aggregation Copy:

ConfirmOrder& ConfirmOrder::operator=(const ConfirmOrder& src){

if (this!=&src){

if (src.m\_ArrSize){

// checking if there is any memory has been allocated for m\_Tppr//

if (m\_Tppr) {

delete[] m\_Tppr;

}

//allocate the memory and ready to shallow copy all the object//

m\_Tppr = new const Toy \* [src.m\_ArrSize];

// This is the most confusing part//

// Shallow copying inside a deep copy logic?//

// Remember the class type is aggregation//

// My brain fights against to the finger muscle//

for (size\_t i = 0; i < src.m\_ArrSize; i++) {

// letting two ptr pointed at the same object//

// don't actually create memory for this//

// cuz you are not controlling the life time of that object//

// as for deallocation, not your business, because this is aggregation//

m\_Tppr[i] = src.m\_Tppr[i];

}

}

m\_ArrSize = src.m\_ArrSize;

}

return \*this;

Composition moving

Child& Child::operator=(Child&& src)noexcept{

if (this!=&src){

if (src.m\_Tppr&&src.m\_Counter){

if (m\_Counter){

this->deallocate();

}

m\_Tppr = new const Toy \* [src.m\_Counter];

for (size\_t i = 0; i < src.m\_Counter; i++) {

m\_Tppr[i] = src.m\_Tppr[i];

src.m\_Tppr[i] = nullptr;

}

delete[] src.m\_Tppr;

src.m\_Tppr = nullptr;

}

m\_Age = src.m\_Age;

src.m\_Age = 0;

m\_Counter = src.m\_Counter;

src.m\_Counter = 0;

m\_Name = src.m\_Name;

src.m\_Name = "";

}

return \*this;

}

Aggregation moving:

ConfirmOrder& ConfirmOrder::operator=(ConfirmOrder&& src)noexcept{

if (this!=&src){

if (src.m\_ArrSize){

if (m\_ArrSize) {

delete[] m\_Tppr;

}

// I know this is also confusing//

// only shallow copy the pointer pointing to the array of pointers//

// What about the elements inside that array?//

// Would that be memory leak?//

// Again this is a aggregation//

// We only care about the 1st depth matters//

// 2nd depth's is not our business//

// If there is a leak, that must be the other people's problems//

m\_Tppr = src.m\_Tppr;

src.m\_Tppr = nullptr;

}

m\_ArrSize = src.m\_ArrSize;

src.m\_ArrSize = 0;

}

return \*this;

}

In the composition copy operator, I allocated the memory for the array first, then make each element pointed to a newly allocated memory contains a nameless object created by the Toy constructor.

However, in the aggregation copy operator, after allocating the memory for the array, I just simply let the element pointers pointed to the element from the array of src. So two pointers pointing at the same memory. This needs some courage because OOP244 told us this is a big no no. But remember, in ConfirmOrder module, we are not allowed to manipulate the element pointers resource. So no matter how many pointers pointing to the chunk of memory. As long as let the other module delete it and only once. It is okay. No memory thrown.

For the composite moving operator, I first created new memory for the array, and then used a for loop to let each element pointing to the array element resource of src. Then I need to deallocate the src array resource to mimic the moving.

As for the Aggregation moving operator, I just simply letting the m\_Tppr pointing at the resource of src.m\_Tppr, and let the let the src.m\_Tppr pointing at nullptr.

1. Identify places in your code where you have used **association**, **aggregation** and **composition** relations.

In this workshop I did not use any association relationship in my code. As for the aggregation and composition. Apart from the points I listed above, I also would like to mention that difference between the aggregation class and composition class.

Aggregation class destructor:

ConfirmOrder::~ConfirmOrder(){

// again the whole design makes us to understand what is a real aggregation//

// You can only touch the array pointer's memory//

// Going one depth further would be causing issues and violating the aggregation rules//

delete[] m\_Tppr;

}

Composition destructor:

Using the deallocate private member function.

void Child::deallocate(){

// class relation is composite//

// Clean up needs to reach 2nd depth//

for (size\_t i = 0; i < m\_Counter; i++){

delete m\_Tppr[i];

}

delete[] m\_Tppr;

}

Child::~Child(){

this->deallocate();

}

As you can read from the comments of my code, the difference reflects the class relationship’s difference.

Once more, the composition class has the full control of the life time of an composed class object, which the aggregation class is not able to do so.