資料結構與進階程式設計（108-2）

手寫作業四

B08705034 資管一 施芊羽

Question 1

Answer:

In the code on Page 12 and 13, the lecturer use a one-dimensional dynamic array to store the objects `Car* cars` this kind of allocation is not as good as the using of two-dimensional array on Page 14 and 15 `Car** cars`. Since when every one dimensional array is created `this -> cars = new Car[this -> Capacity]`, the compiler will keep calling the building constructors to build each object. As if we use the “CarArray” on Page 12 and 13, the building constructor of `Car` will be called for `this -> capacity` times and it'll become a waste of time and memory spaces, even though in this case the building constructor is short and does not require a lot of time to run it, in the future, if we use an one-dimensional dynamic array to build an object array it’ll waste a lot of times.

Therefore, if we use the way the lecturer did on Page 14 and 15, the first time we’re creating the first dimension of the array the compiler will not using a building constructor to create an object, it’ll just create a pointer. In this case, it will create `this -> capacity` number of pointers. Thus, it’ll save time and also prevent from the case that if the object array is not required to store that many objects(if we're not going to create `this -> capacity` number of cars, we don’t need to call that many constructors).

Question 2

Answer:

If we'd like to see the status string instead of 0 or 1 during the running of the `print` function of the objects, we need to rewrite `add` functions of `CarArray` and `PassengerArray` using the concept of polymophism. Since `CarArray` and `PassengerArray` are the child class of `EntityArray`, we have to modify `add` functions of `CarArray` and `PassengerArray` because we want to use the `print` member functions of `Car` and `Passenger` in order to print out their status string.

At first, we need to virtualize `add` function of their parent class `EntityArray`. Thus, when we are using `CarArray` and `PassengerArray`, we can use their own `add` functions. The second step, we need to modify the `add` function of `CarArray` and `PassengerArray`.

For `CarArray`, we'll change its `add` function as:

```
bool CarArray::add(string id, bool isOn, bool isSer, double lon, double lat)
{
  if(this->cnt < this->capacity)
  {
    this->entityPtr[this->cnt] = new Car(id, isOn, isSer, lon, lat);
    this->cnt++;
    return true;
  }
  else
    return false;
}
```

Since we make `entityPtr` point to a `Car` object, when we're going to print out objects stored in `CarArray`, we can use the `print` function of `Car` so that we can print out their status string.

We'll do the same thing for `PassengerArray`:

```
bool PassengerArray::add(string id, bool isOn, bool isSer, double lon, double lat)
{
  if(this->cnt < this->capacity)
```

```
  {
    this->entityPtr[this->cnt] = new Passenger(id, isOn, isSer, lon, lat);
    this->cnt++;
    return true;
  }
  else
    return false;
}
```

Therefore, we can use the `print` function of `Passenger` when we're trying to print the objects stored in `PassengerArray`.

To sum up, we have to apply the concept of polymophism so that when we want to use different functions of the child class.

Question 3

Answer:

On the slides, the lecturer didn't use `template` or create `CarArray` and `PassengerArray` on Problem 3. Therefore, in the main function, when we are going to create two arrays to store information about cars and passengers, the compiler will use `Entity` object to store the information about `Car` and `Passenger`. Thus, it'll result in the member functions and variables that is not inherited from `Entity` of `Car` and `Passenger` cannot be accessed(e.g. `print` functions).

Question 4

Answer:

In Problem 3, we use `if(this->cnt < this->capacity)` to determine whether the `EntityArray` has stored objects that's more than its capacity. However, it's not good enough because during the process of the codes, we won't get any announcement about what's happening even if there has no space to add more `Entity` to the `EntityArray`.

Therefore, after we modified the function to:

```
void EntityArray::add(string id, bool isOn, bool isSer, double lon, double lat) throw(overfl
ow_error)
{
  if(this->cnt < this->capacity)
  {
    this->entityPtr[this->cnt] = new Entity(id, isOn, isSer, lon, lat);
    this->cnt++;
  }
  else
    throw overflow_error(id + ": capacity limit reached!");
}
```

In the main function, we can catch the exception `id:capacity limit reach`, and know what's wrong during the execution of the codes.

Question 5 - (a)

Answer:

The following code is the way we use exception handling to determine whether a `logic_error` has happened: in `double averageNonzero(double grades[], int gradeCnt) throw(logic_error)`

```
if(nonzeroCnt == 0)
    throw logic_error("divisor is 0!");
  else
    return sum / nonzeroCnt;
```

and in `main` function we do:

```
try
  {
```

```
  {
    avg = averageNonzero(grades, gradeCnt);
  }
  catch(logic_error e)
  {
    avg = 0;
  }
  cout << avg;
```

The modified version of the code using fail-safe will be like:

```
int main()
{
  double grades[100] = {0}, avg = 0;
  int gradeCnt = 0;
  while(true)
  {
    double temp = 0;
    cin >> temp;
    if(temp == -1)
      break;
    else
    {
      grades[gradeCnt] = temp;
      gradeCnt++;
    }
  }
  int nonzero = 0;
  double sum = 0;
  for(int i = 0 ; i < gradeCnt ; i++){
      if(grades[i] != 0){
           nonzero++;
           sum += grade[i];
      }
}
if(nonzero != 0){
    cout << sum / nonzero;
}
  return 0;
}
```

Question 5 - (b)

Answer:

As the developer of the function, the using of exception handling will be better. That's because for other user of the codes or developers, they can receive the messages about what's wrong during the execution of the codes(they'll receive the message `logic_error("divisor is 0!")`). However, if using the fail-safe determination in `main`, other people can only see no print-out while not knowing what error has happened. Therefore, the way using exception handling is better.

Question 6 - (a)

Answer:

Because we used to create a `EntityArray` that can only contain no more than `capacity` many `Entity`. However, if we use `<vector>` instead of creating a two-dimensional array, we can store much more `Entity` objects. Therefore, we don't need to know whether the `EntityArray` is full or bot since the `vector` can store as many objects as we want(as if it's less that the `max_size` of that vector, e.g. in this case `this->entities.max_size()`). I think it's useful since it allows us to store much more objects. On the other hand, it'll not be suggested for those who are not well-understanding about the concepts of pointers and references, because it'll be even more troublesome for them to create and use the `EntityArray`.

Question 6 - (b)

Answer:

We'll create a new member function `int getEntityNumber()` in `EntityArray` as the getter to know how many `Entity` objects are stored in `vector<Entity> entities`:

```
int EntityArray::getEntityNumber(){
    return this->entities.size();
}
```

The function will return the size of the vector and we can know how many `Entity` are stored.

In [ ]:

```
```