

1. Advantages of using heap-dynamic variables

Flexibility:

In the "ramenRestaurant.cpp" file, the RamenRestaurant class uses a dynamic array of Ingredient pointers called "ingredientStorage" to store the prepared ingredients. The size of this array is determined by the "ingredientStorageCapacity" parameter provided during the instantiation of the RamenRestaurant object.

```
RamenRestaurant::RamenRestaurant(int ingredientStorageCapacity) : ingredientStorageCapacity(ingredientStorageCapacity)
{
    ingredientStorage = new Ingredient*[ingredientStorageCapacity]; //create a dynamic array of Ingredient pointers, of size ingredientStorageCapacity
    for(int i=0; i<ingredientStorageCapacity; i++) //we should set all ingredientStorage slots to nullptr's since our storage is literally empty at the beginning
        ingredientStorage[i] = nullptr;
}
```

The advantage of using heap-dynamic variables in this case is that it allows flexibility in storing a varying number of ingredients. The capacity of the ingredient storage can be adjusted at runtime.

Moreover, When dealing with different types of classes that share a common base class, using heap-dynamic variables allows you to store objects of those classes in the same array. For example in RamenRestaurant, the different ingredients can be stored on the ingredientStorageCapacity which the stack variable can't do or hard to do.

```
bool RamenRestaurant::prepareNoodle(int softness)
{
    if(isStorageFull()) //cannot prepare new noodles when the storage is full
    {
        cout << "Whoops! No more storage space for the new noodle! :(" << endl;
        return false;
    }
    addFoodToStorage( food: new Noodle(softness)); //prepare it and store it
    cout << "Noodle (" << softness << "% softness) has been prepared and added to storage! :)" << endl;
    return true;
}

bool RamenRestaurant::prepareSoup(int spiciness)
{
    if(isStorageFull()) //cannot prepare new soup when the storage is full
    {
        cout << "Whoops! No more storage space for the new soup! :(" << endl;
        return false;
    }
    addFoodToStorage( food: new Soup(spiciness)); //prepare it and store it
    cout << "Soup (" << spiciness << "% spiciness) has been prepared and added to storage! :)" << endl;
    return true;
}
```

Storage efficiency in some situations:

Compared to the stack variable, the stack variable can't delete in specific operation. The stack variables have automatic storage duration, meaning they are automatically allocated and deallocated as they go in and out of scope. However, if the stack variable were create a lot we can't delete in now scope.

If we use heap variables, because heap variables have dynamic storage duration and are allocated on the heap using dynamic memory allocation (e.g., new operator). The memory can be released and the memory can be reused by other variables. In this case, the heap variable have more efficiency in this situation.

```
using Ingredient = Ingredient*;
RamenRestaurant::RamenRestaurant(int ingredientStorageCapacity) : ingredientStorageCapacity(ingredientStorageCapacity)
{
    ingredientStorage = new Ingredient*[ingredientStorageCapacity]; //create a dynamic array of Ingredient pointers, of size ingredientStorageCapacity
    for(int i=0; i<ingredientStorageCapacity; i++) //we should set all ingredientStorage slots to nullptr's since our storage is literally empty at the beginning
        ingredientStorage[i] = nullptr;
}

RamenRestaurant::~RamenRestaurant()
{
    for (int i = 0; i < ingredientStorageCapacity; i++) {
        delete ingredientStorage[i];
    }
    delete[] ingredientStorage;
}
```

```
delete suitableNoodle;
delete suitableSoup;
delete suitablePork1;
ingredientStorageUsed = ingredientStorageUsed -3;
if (num_pork == 2) {
    delete suitablePork2;
    ingredientStorageUsed--;
}
```

2. Disadvantages of using heap-dynamic variables

Runtime efficiency:

Allocating memory on the heap using the 'new' keyword incurs runtime overhead. Deallocating memory using the 'delete' keyword also takes time. In the destructor of the RamenRestaurant class, the ingredientStorage array is deallocated using 'delete[]'.

```
~RamenRestaurant():~RamenRestaurant(int ingredientStorageCapacity) : ingredientStorageCapacity(ingredientStorageCapacity)
{
    ingredientStorage = new Ingredient*[ingredientStorageCapacity]; //create a dynamic array of Ingredient pointers, of size ingredientStorageCapacity
    for(int i=0; i<ingredientStorageCapacity; i++) //we should set all ingredientStorage slots to nullptr's since our storage is literally empty at the beginning
        ingredientStorage[i] = nullptr;
}

RamenRestaurant::~RamenRestaurant()
{
    for (int i = 0; i < ingredientStorageCapacity; i++) {
        delete ingredientStorage[i];
    }
    delete[] ingredientStorage;
}
```

Accessing stack-dynamic variables is generally faster compared to accessing heap-allocated memory. Stack memory is typically allocated in a contiguous block, allowing for more efficient memory access. However, heap memory can be fragmented, leading to slower access times.

The left screenshot shows a debugger window with a list of variables. The 'ingredientStorage' array is highlighted, showing its address as 0x7ffefb7f0000. The right screenshot shows a similar view for the 'ingredientStorage' array, with its address listed as 0x7ffefb7f0000. Both screenshots show the array's contents, which are all nullptr.

We can see that in the heap variable, ingredientStorage[0] and ingredientStorage[1]. They are not continuous addresses. However, the stack variables are continuous addresses.

The screenshot shows a debugger window with a list of variables. The 'ingredientStorage' array is highlighted, showing its address as 0x7ffefb7f0000. The right screenshot shows a similar view for the 'ingredientStorage' array, with its address listed as 0x7ffefb7f0000. Both screenshots show the array's contents, which are all nullptr.

Reliability:

Using heap memory variables may cause Dangling Pointers, freeing memory before it is finished using. If the dangling pointer has been used, the system will cause unpredictable behavior.

```
for(int i=0;i<4;i++){
    ingredientStorage[dA[i]] = nullptr;
}

delete suitableNoodle;
delete suitableSoup;
delete suitablePork1;
```

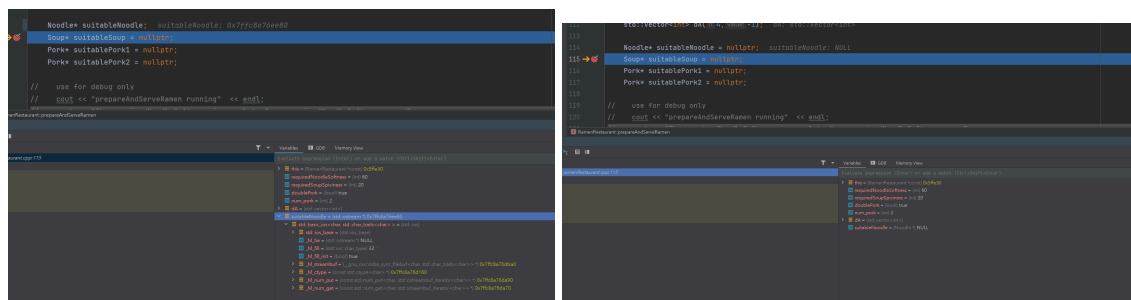
For example, if we don't set the variable to nullptr, it will become the dangling Pointers. Then it will crash because of the dangling pointers.

```
===== 5th hour =====
Teemo: my ramen restaurant is now open!
Teemo: and we already have a customer!
Teemo: she wants at least 60% soft noodles and at least 20% spicy soup! double pork as well!
Teemo: let's prepare a nice bowl of ramen for her!

Ramen has been skillfully prepared and happily served! :)

Currently, we have 6 ingredients in the restaurant storage:
Slot 0: Noodle (70% quality, 0% softness)
Slot 1: Noodle (70% quality, 55% softness)
Slot 2:
PS C:\Users\tch0905\CLionProjects\csci3180ass1>
```

Moreover, the heap memory variable needs to be carefully initialize, if we do not point the pointer to nullptr, the pointer will point to an unknown address.



If we point to nullptr. The pointer won't point to the unknown address.

And we need to be careful to manage the heap variable to avoid the memory leak, where memory is allocated but not properly deallocated. For example, we need to delete the unused variable to deallocate the memory properly.

```
delete suitableNoodle;
delete suitableSoup;
delete suitablePork1;
```

Therefore, we need to be careful when using the heap variable.