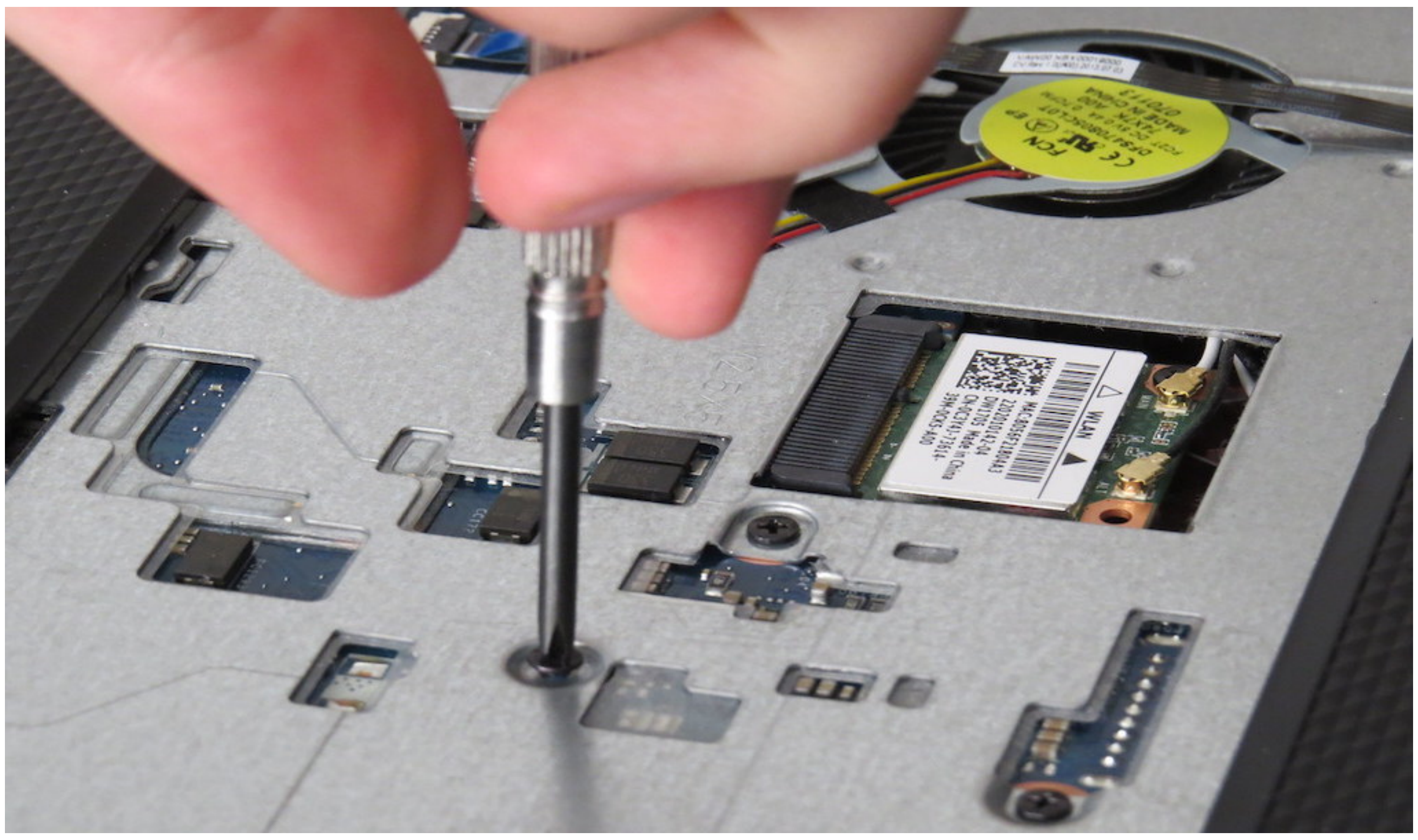
# Lab 6 Friend and Static


Credit: Luke Jones, via https://www.flickr.com/photos/befuddledsenses/13120237513, photo resized by the lab creator.

Note: Skeleton code is updated on Nov 3, at 11AM.

## Computer Repair System

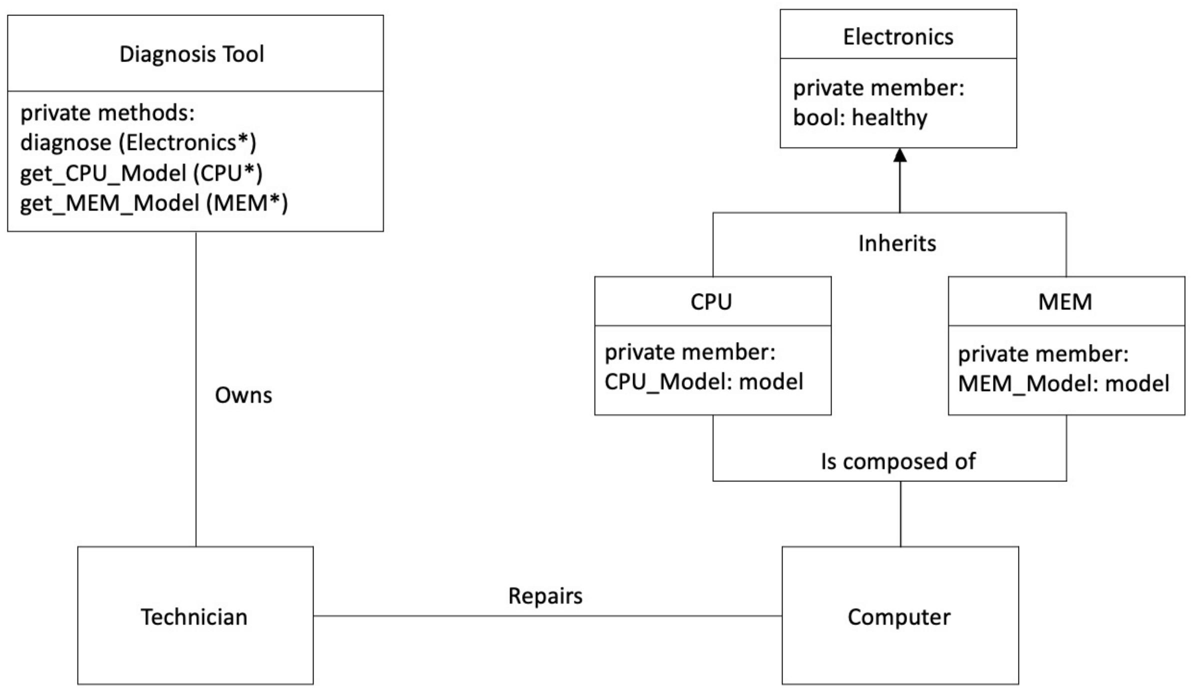
In this lab, you will implement a Computer Repair System with the concept of `friend` and `static`. Before starting this lab, you should read the friend and static lecture notes. The skeleton files are available HERE.

### Story

Imagine you have just been promoted to manager in the Apple Genius Bar. You have to manage your fellow Technicians to repair Computers. Your boss is very concerned about the working efficiency of Technicians, and the number of defective computers sold to customers. With exceptional programming skills, you will implement a Computer Repair System which can simulate the flow of repair and generate reports.

### Overview

Before diving deep into each .h/.cpp file in the source file, we will introduce how the various components in the system interact with each other.



Menu
- Introduction
- Skeleton Files
- Lab Tasks
- Submission Guidelines

Page maintained by

Ivan Choi
Last Modified: 11/10/2022 18:55:44

Homepage

Course Homepage

*<u>Note:</u> The diagram does not follow the standards of UML class diagrams. It is just an illustration for better understanding.*
The following will explain the diagram in detail:

1. `Technician` repairs `Computer`.
2. Each `Computer` is composed of a `CPU` and a `MEM` (memory).
   `Computer` has a private pointer to its `CPU` and a private pointer to its `MEM`.
3. `Electronics` is the base class of `CPU` and `MEM`.
4. `Electronics` has a private member called `healthy`, indicating whether such an electronics is functional.
5. `CPU` has a private member called `model`, which is of type `CPU_Model`.
   `CPU_Model` is an enum class which contains all the models of `CPU`.
   When `Technician` finds out the CPU inside a `Computer` is defective, he/she has to replace it with another `CPU` of the same `CPU_Model`.
6. `MEM` has a private member called `model`, which is of type `MEM_Model`.
   `MEM_Model` is an enum class which contains all the models of `MEM`.
   When `Technician` finds out the MEM inside a `Computer` is defective, he/she has to replace it with another `MEM` of the same `MEM_Model`.
7. Each `Technician` is equipped with a `Diagnosis_Tool` to help them investigate the status of `Computer`.
   `Diagnosis_Tool` consists of private member functions which can diagnose `Electronics`, or get the model of `CPU` or `MEM`.

## What is an enum class?

`CPU_Model` and `MEM_Model` are enum classes. You have learnt enum in COMP2011. In this lab, we use another new feature introduced in C++11, enum class. More specifically, the enum we learnt in COMP2011 is called unscoped enum, whereas enum class is called scoped enum.

```
enum Department = {CSE, ECE, MATH}; // Unscoped enum
enum class Department = {CSE, ECE, MATH}; // An enum class, scoped enum
```

There are 2 major differences between them:

1. Enum classes are scoped
   When we use an unscoped enum, we can access the enumerators (i.e. CSE, ECE... those defined inside the curly braces) directly, i.e.:

   ```
   Department d = CSE; // Works fine
   ```

   However, for the scoped enum, we need to use the scope resolution operator `::` to access the enumerators, i.e.:

   ```
   Department d = CSE; // Error!
   Department d = Department::CSE; // Ok
   ```

2. Scoped Enum does not support Implicit Conversion
   For scoped and unscoped enum, if we do not specify the type, its underlying type for every enumerator (i.e. CSE) is integral. However, only unscoped enum supports implicit conversion. For example:

   ```
   enum Department = {CSE, ECE, MATH}; //unscoped enum, CSE = 0, ECE = 1, MATH
   = 2
   int n = MATH; //ok, now n = 2
   ```

   whereas for scoped enum, implicit conversion is not supported:

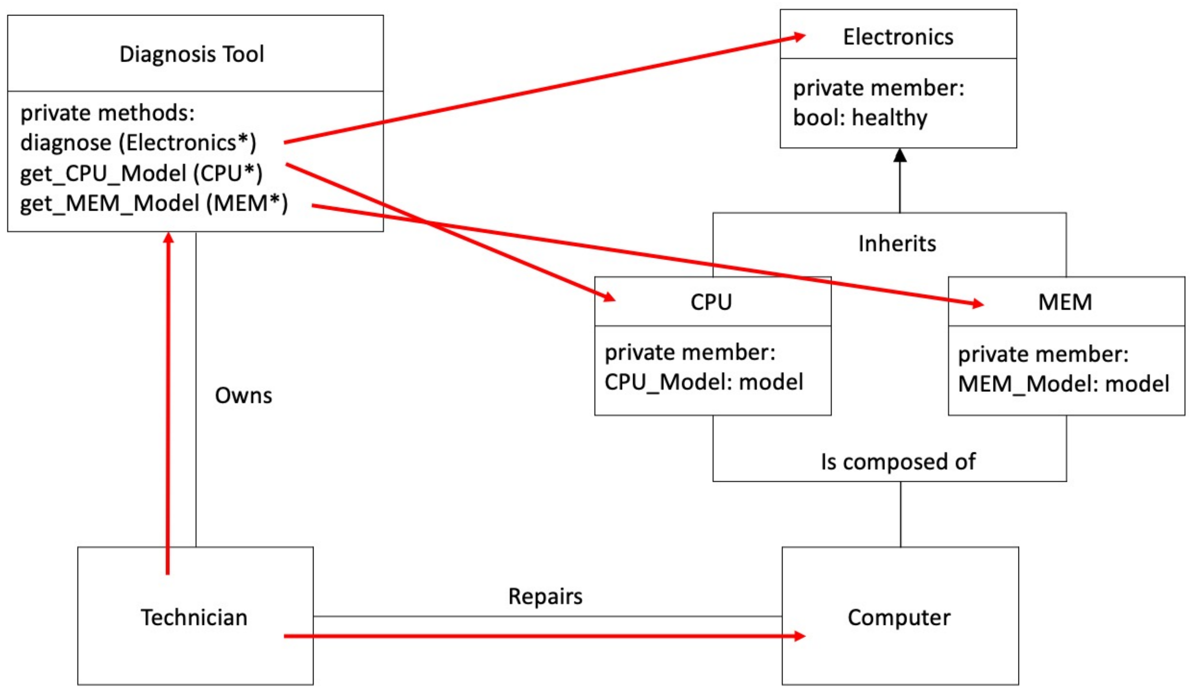
   ```
   enum class Department = {CSE, ECE, MATH}; // Scoped enum, CSE = 0, ECE = 1,
   MATH = 2
   int n = Department::MATH; // Error, no implicit conversion
   int m = static_cast<int>(Department::MATH); // Explicit conversion works
   fine
   ```

Generally, scoped enum is preferred to unscoped enum because the prevention of implicit conversion gives less undesired behaviours. It also improves readability as we have to explicitly mention the enum class name.

## Friend

In realistic settings, only `Technicians` are trained to use `Diagnosis_Tool`. Other people cannot use them even if they have a `Diagnosis_Tool`. A `Diagnosis_Tool` provides functions to access the models and status of different kinds of `Electronics`, so that a `Technician` can understand which component is defective and what model of `CPU` or `MEM` he/she should replace. Also, only `Technicians` are allowed to disassemble `Computers`.

We introduced many private data members or member functions for each class in the overview session. Of course, Technicians can't repair Computers if they cannot access the private data members or member functions in other classes. You may have already thought of using the concept of `friend` to finish the system. Indeed, we will use friends intensively in this lab. The following graph will explain in detail.



Recall a friend relationship is directional, which means that if A is a friend of B, then B is not necessarily a friend of A. In the graph, each red arrow, starts at a class/function and ends at another class, defining one friend relationship. The starting class/function of each arrow should be able to access the private data member/method of the ending class. in detail:

1. `Technician` should be able to access the private member functions of `Diagnosis_Tool` so that he/she can retrieve information about the `Computer`.
2. `Technician` should be able to access the private data members of `Computer` so that he/she can replace the defective `CPU` or `MEM` in the `Computer`.
3. The `diagnose(Electronics*)` function in `Diagnosis_Tool` should be able to access the private data members of `Electronics`.
4. The `get_CPU_Model(CPU*)` function in `Diagnosis_Tool` should be able to access the private data members of `CPU`.
5. The `get_MEM_Model(MEM*)` function in `Diagnosis_Tool` should be able to access the private data members of `MEM`.

In total, there are 5 friend relationship.

## Static

Now we will focus on report generation. Your boss give you a sample report that you should work on.

```
----------------------
REPORT FOR TECHNICIANS
----------------------
NAME                ORDERS_COMPLETED
Harry               2
Ron                 1
Hagrid              0
Hermoine            0
----------------------
REPORT FOR ELECTRONICS
----------------------
Total number of CPU used: 6
Number of Defected CPU: 3

Total number of MEM used: 5
Number of Defected MEM: 2
----------------------
--- END OF REPORT ----
----------------------
```

This report will be generated by a helper function that resides in `util.cpp`:

```
void print_report();
```

In the helper function, we will call other static functions to help us generate the report. There are 3 static functions in this lab.

```
void MEM::print_report();
void CPU::print_report();
void Technician::print_report();
```

The `print_report` helper function will call these static functions to generate the report. We have already implemented all the static functions for you.

Several static variables will help us generate the report:

- Static variable defined in class `Technician`:

  ```
  static Technician* technician_list[]
  ```

  `technician_list` is an array of `Technician` pointers that will keep track of all `Technicians` you manage. It will be updated when a new `Technician` is created. `Technicians` in this list will be sorted by their creation time.

  ```
  static int num_technician;
  ```

  `num_technician` keeps track of the current number of `Technician`(s) recruited.

- Static variable defined in `CPU`:

  ```
  static int manufacture_count;
  ```

  `manufacture_count` keeps track of the current number of `CPU`s produced. The initial value will be 0. It will increase by 1 when a `CPU` is created in the system.

  ```
  static int defect_count;
  ```

  `defect_count` keeps track of the current number of defective `CPU`s. The initial value will be 0. It will increase by 1 only when a `Technician` finds out a `CPU` is defective.

Class `MEM` has the same set of static data members as class `CPU`. Their functionalities are exactly the same. Just replace the `CPU` with `MEM` in the above explanation.

## Lab tasks

### Task 0: Read the .h/.cpp files

You should read the .h/.cpp files to get familiar with each class.
Suggested reading sequence:

1. `technician.h/.cpp`: class `Technician` and class `Diagnosis_Tool`
2. `electronics.h/.cpp, model.h`: class `Electronics`, class `CPU`, class `MEM` and enum class `CPU_Model` and `MEM_Model`.
3. `computer.h/.cpp`: class `Computer`
4. `util.h/.cpp`: `print_report` function
5. `main.cpp`: the sequence of function calls to produce the sample output.

*Notes:*
*In `main.cpp`, you will find that all `Diagnosis_Tool`, `CPU` and `MEM` are dynamically allocated. Be careful when you have to deal with their creations or destructions.*

*In `computer.h/.cpp` and `electronics.h/.cpp`, you will encounter functions with the prefix "check": `check_computer`, `check_healthy` and `check_model`. They are functions that will judge the correctness of your solution to this lab. You may think of it as a Quality Assurance (QA) step. You should not use them in your tasks or else you will introduce extra lines of output, unless you want to use them for debugging.*

### Task 1: Finish all friendship relationship declarations

There are 5 TODO friends relationship that you should declare in the following classes:

- `Diagnosis_Tool`
- `Electronics`
- `CPU`
- `MEM`
- `Computer`

You should follow the 5 requirements that are defined in the Friend session.
Tips: You may try both friend functions/friend classes and observe their effects. Why do some work and some do not work?

### Task 2: Static variables initialization

1. In `technician.cpp`:
   - Initialize `technician_list` in class `Technician` to be an array of length `MAX_NUM_TECHNICIAN`, and every element inside `technician_list` to be `nullptr`.
     *Note:* *(You should know how to do this after Lab 2.)*
   - Initialize `num_technician` in class `Technician` to be 0.
2. In `electronics.cpp`:
   - Initialize `manufacture_count` in class `CPU`, and class `MEM` to be 0.
   - Initialize `defect_count` in class `CPU`, and class `MEM` to be 0.

### Task 3: Implement constructor in class Technician

```
Technician::Technician(std::string name, Diagnosis_Tool* diagnosis_tool);
```

Assign the arguments of the constructor to the `Technician` object. You may want to refer to `main.cpp` to understand how this constructor is called. Remember to take care of the corresponding static variable of this class.
Think: how can you get the pointer of the object and assign it to `technician_list`?

## Task 4: Implement repair function in class Technician

```
void Technician::repair(Computer* computer);
```

This function will find out the defective CPU/MEM of the computer and replace the defective one with a functional one with the correct model.
You will need to use functions from `Diagnosis_Tool`, and 2 member functions in class `Technician`:

```
bool Diagnosis_Tool::diagnose(Electronics* electronics);
CPU_Model Diagnosis_Tool::get_CPU_Model(CPU* cpu);
MEM_Model Diagnosis_Tool::get_MEM_Model(MEM* mem);
CPU* Technician::acquire_CPU(CPU_Model model) const;
MEM* Technician::acquire_MEM(MEM_Model model) const;
```

Ensure that you also take care of the corresponding static variables from all classes, and the data member variables for the `Technician`.
This task heavily depends on the correctness of the friend relationship to be correctly defined above. If you encounter any difficulties, you may want to revise Task 1.

Hint: Make sure you prevent memory leaks in Task 4!

## Task 5: Implement print_report function in util.cpp

```
void print_report();
```

Call the `print_report()` static function of class Technician, CPU, MEM.

## Test Case

After you have finish the tasks, you may run make to generate the executable. Run `./lab6.exe` and compare it with the following output.

```
QA starts checking the Computer
QA checks for healthy for electronics...
It is working great!
QA checks for healthy for electronics...
It is working great!
The Computer is fully repaired!
QA checks for the CPU model...
The CPU model is correct!
QA checks for the MEM model...
The MEM model is correct!
The models are well-matched!
QA finished checking the Computer.

QA starts checking the Computer
QA checks for healthy for electronics...
It is working great!
QA checks for healthy for electronics...
It is working great!
The Computer is fully repaired!
QA checks for the CPU model...
The CPU model is correct!
QA checks for the MEM model...
The MEM model is correct!
The models are well-matched!
QA finished checking the Computer.

QA starts checking the Computer
QA checks for healthy for electronics...
It is working great!
QA checks for healthy for electronics...
It is working great!
The Computer is fully repaired!
QA checks for the CPU model...
The CPU model is correct!
QA checks for the MEM model...
The MEM model is correct!
The models are well-matched!
QA finished checking the Computer.

----------------------
REPORT FOR TECHNICIANS
----------------------
NAME                ORDERS_COMPLETED
Harry               2
Ron                 1
Hagrid              0
Hermoine            0
----------------------
REPORT FOR ELECTRONICS
----------------------
Total number of CPU used: 6
Number of defective CPU: 3

Total number of MEM used: 5
Number of defective MEM: 2
----------------------
--- END OF REPORT ----
----------------------
```

The QA part will determine whether you implement your task 4 correctly.
The Report part will determine whether you deal with all static variables correctly.

We have 2 test cases on Zinc for this lab exercise. Both of them will run the program and produce output as the sample output, except test case 2 will run with memory leak check.

## Submission Deadline and Guidelines:

The lab assignment is due 10 minutes after the end of your lab session.
We will use an online grading system ZINC, to grade your lab work. Therefore, you are supposed to upload the following files in a zip file to ZINC:

```
- computer.h
- electronics.cpp
- electronics.h
- util.cpp
- technician.cpp
- technician.h
```

You can submit your codes multiple times to ZINC before the deadline. Only the last submission will be graded.