"Hello medPing"

An Introduction to the `medPing API` (Application Programmer's Interface)

This lab will introduce you to Object-Oriented Programming (OOP) via a collection of classes that form the medPing API. The medPing API simulates access to an embedded medical device that collects real-time medical diagnostics such as a patient's vital sign. The patient "wears" an embedded medPing chip; the chip collects data and then "pings" your phone or watch. The *use* of the medPing API is up to you, that is, the medPing API is just the software that ships with the embedded medical device. *What* you build as an application is up to you, the software developer. In this lab, you will practice using the medPing API to collect and store medical data.

(0) Download the Starter Kit and unzip. On first glance, you should notice that you have many .cpp and .h files. Welcome to software engineering in the large.

(1) Create a console project. Add *all* the .h (header) and .cpp (source) files to your project. Compile and run the code. You should see the following output:

```
Hello medPing patient ...
```

(2) Edit `medPing_Main.cpp`.

(3) Ask the medPing chip to provide you with some vital sign data, e.g.,

```
// create a medPing object
medPing mP;

double temp, bpm;

temp = mP.getBodyTemperature_F();
bpm  = mP.getRespirationRate_BPM();
```

(4) Open (but don't edit) **`medPing.h`**. Review the data the medPing chip sends you:

```
                                        // NORMAL RANGES
double  getBodyTemperature_F();         // 97.8 - 99.0 F

short   getPulseRate_BPM();             // 60 - 100 beats per minute

short   getRespirationRate_BPM();       // 15 - 20 breaths per minute

void    getBloodPressure_mmHg(short& systolic,  short& diastolic);
                                        // systolic  < 120 mm Hg
                                        // diastolic <  80 mm Hg

short   getGlucoseLevel_mgdL();
```

(5) Collect and print a full suite of the vital signs you can collect. Make sure you ask the medPing object to print your output; that is, use `mP.CELL_PrintF()`, not `cout`.

Call me over to see your progress.

(6) Write a loop to collect five (5) sets of vital signs. At the bottom of the loop, sleep for a brief time, e.g., two seconds. Look up the `sleep()` function. Note: `sleep()` is different for MacOS and Windows.

(7) Now, suppose you need to save *a history* of the vital signs you have collected. One solution would be to declare **parallel arrays** as your data structure, e.g.:

```
double allTemps[MAX];
short  allPulseRates[MAX]:
 :
       etc.
 :
long hmr;  // number of sets of vital signs so far:  0 ≤ hmr ≤ MAX-1
```

As discussed in class, parallel arrays are *typically not* the best way to go since all the data for one set of vital signs is spread over multiple arrays. A better solution is to store a complete set of vital signs collected at one time in one place: one **`struct`**ure. In general, a **struct** is called a "**record**".

Declare a **struct** like shown below.

```
struct dataFrame
{
        double temp;
        double bpm;
          :
          :
};
```

Make an array of **structs**. Within your loop that simulates the "ping" for vital signs, **store each set of vital signs in the next available struct in your array**.

After collecting a set of vital signs, **write a function to print the entire history of vital signs out** to stdout. Make this function print a *neat, professional* looking report.

Call me over to see your progress.

(8) Write another function to print your history of vital signs to an external file called: **medicalHistory.xls**. Print your headings and data in **c**omma-**s**eparated **v**alue fashion, obviously naming your output file with a **.csv** file extension.