It is what we think we know already
that often prevents us from learning.
~ *Claude Bernard*

CS1010 Programming Methodology
Week 13: Structures

To students:

- Your DL may not cover all the questions here or he/she might give you other questions.
- Some programs for this discussion are on the CS1010 website, under the "Discussion" page.
- Questions 5 to 7 are mounted on CodeCrunch.

I. Structures (Basics)

The following 4 operators are used in this discussion.

(i) The dot operator '.' (also known as structure member operator)
(ii) The arrow operator '->' (also known as structure pointer operator)
(iii) The pointer dereference operator '*'
(iv) The address operator '&'

Note that '.' and '->' have left-to-right associativity and they are of higher precedence than '*' and '&' which have right-to-left associativity.

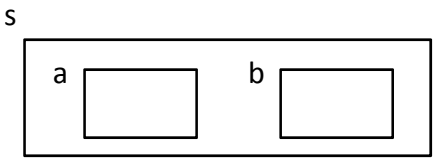
1. Without running **q1.c** on the computer, what output would you expect to get?

```
#include <stdio.h>

typedef struct {
   int p;
   float q;
} one_t;

typedef struct {
   int p;
   float q;
} two_t;

int main(void) {
   one_t one = { 1, 2.3 };
   two_t two;

   two = one;
   printf("%d %f\n", one.p, one.q);
   printf("%d %f\n", two.p, two.q);

   return 0;
}
```

(Note: This program, q1.c, is available on the CS1010 website under the "Discussion" page.)

2. Given the following structure definition and a variable declaration:

```
typedef struct {
   int a;
   float b;
} s_t;

s_t s;
```



The variable **s** is depicted in the diagram on the right above.

If you want to read a value into the member **a** of variable **s**, which of the following statement is/are correct, and why?

(i) scanf("%d", &(s.a));
(ii) scanf("%d", s.&a);
(iii) scanf("%d", (&s).a);
(iv) scanf("%d", &s.a);

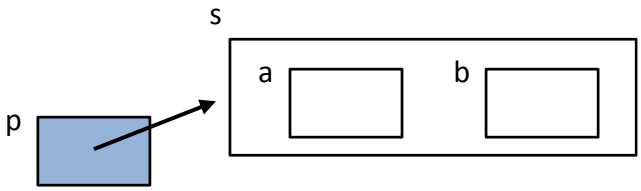
3. Re-attempt questions 3 and 4 of Week 9 Discussion. However, this time round, use structures to return 2 or more outputs from a function instead of pointers.

II. More on Structures

4. (a) Given the following structure definition and 3 additional statements:

```
typedef struct {
   int a;
   float b;
} s_t;
```

```
s_t s;
s_t *p;
p = &s;
```



The variable **s** and **p** are depicted in the diagram on the right above.

If you want to read a value into the member **a** of variable **s**, through the pointer **p**, which of the following statement is/are correct, and why?

(i) scanf("%d", &p.a);
(ii) scanf("%d", &(p.a));
(iii) scanf("%d", *p.a);
(iv) scanf("%d", p.a);
(v) scanf("%d", (*p).a);
(vi) scanf("%d", p->a);
(vii) scanf("%d", &(*p).a);
(viii) scanf("%d", &(*p.a));
(ix) scanf("%d", &(p->a));

5. Write a program **tiles.c** to read in an integer (greater than 1) indicating the number of tiles, followed by the tiles' data (length, width and price per square metre). A structure called **tile_t** should be created and the tiles' data should be stored in an array of such structure. The program then computes and outputs the difference in cost between the cheapest tile and the most expensive tile.

   (Actually, to get the answer there is no need to store the data in an array. This is done just for you to practise using array of structures.)

   The length and width are integers in metres, while the price, in dollars, is of type **float**. You may assume that there are at least 2 tiles and at most 20 tiles.

   You should write a modular program with the following functions:

```
// To read tiles' data into array tiles
// Return the number of tiles read
int scan_tiles(tile_t tiles[]);

// Return the difference in cost between cheapest
// tile and most expensive tile in the array tiles
float difference(tile_t tiles[], int size);
```

   A skeleton program **tiles_skeleton.c** is given. A sample run is shown below.

```
Enter number of tiles: 5
Enter data for 5 tiles:
5 8 0.20
3 5 0.18
6 10 0.31
4 6 0.27
2 4 0.38
Largest difference = $15.90
```

6. **Cumulative Average Point (CAP)**

Write a program **cap.c** that makes use of these structures:

- **result_t** that contains 3 members: a 7-character module code, the grade obtained by the student, and the number of modular credits (MCs) of that module; and
- **student_t** that contains the student's name (at most 30 characters), and an array of result_t structures. You may assume that a student can take at most 50 modules.

Your program should read in a student's name, the number of modules he has taken, and for each module, the module code, the grade obtained, and the number of modular credits. All these data should be stored in a **student_t** variable. Your program should then compute the student's CAP, based on this formula:

CAP = Σ (MCs × Grade Point) / Σ (MCs)

The table below shows the grade point corresponding to each grade:

Grade	A+ or A	A-	B+	B	B-	C+	C	D+	D	F
Grade Point	5.0	4.5	4.0	3.5	3.0	2.5	2.0	1.5	1.0	0

For example, if Brusco Beh's has taken 5 modules and his results (module code, grade obtained, and number of MCs) are as follows:

```
CS1010 A+ 4
CS1231 B 4
MA1101R B+ 4
GEM1211 A- 3
PH2001 C 4
```

then his CAP is calculated as follows:

(5.0×4 + 3.5×4 + 4.0×4 + 4.5×3 + 2.0×4) / (4 + 4 + 4 + 3 + 4) = 71.5/19 = **3.76**

A skeleton program **cap_skeleton.c** is given . A sample run is shown below.

```
Enter student's name: Brusco Beh
Enter number of modules taken: 5
Enter results of 5 modules:
CS1010 A+ 4
CS1231 B 4
MA1101R B+ 4
GEM1211 A- 3
PH2001 C 4
CAP = 3.76
```

7. [Past-year's exam question] **Class Schedule**
   You are to write a program **schedule.c** that uses the **interval_t** structure which contains two integer members: **start** and **finish** of an interval. The program is to read the following data:

   - The first integer contains a positive value *n* representing the number of lessons. You may assume there are at most 20 lessons.
   - This is followed by data for the *n* lessons. For each lesson, two non-negative integers *start* and *finish*, where *start* < *finish*, represent the start time and finish time of the lesson. You may assume that the latest finish time is 1000.

   Assuming that the following data have been prepared by the user:

```
9
200 240
210 230
30 60
80 100
10 40
200 260
260 280
150 180
160 170
```

   Your program is to compute the following:

   - The duration of the longest lesson. In the above example, the longest lesson is from 200 to 260 with duration of **60**.
   - The number of free periods from the time the first lesson starts to the time the last lesson ends. In the above example, there are **3** free periods: 60 to 80, 100 to 150, and 180 to 200. (0 to 10, and 280 to 1000 are not considered free periods.)
   - The most number of concurrent lessons. In the above example, there are **3** concurrent lessons going on during the period 210 to 230.

   You may create additional array(s) and/or functions in your program if you wish.

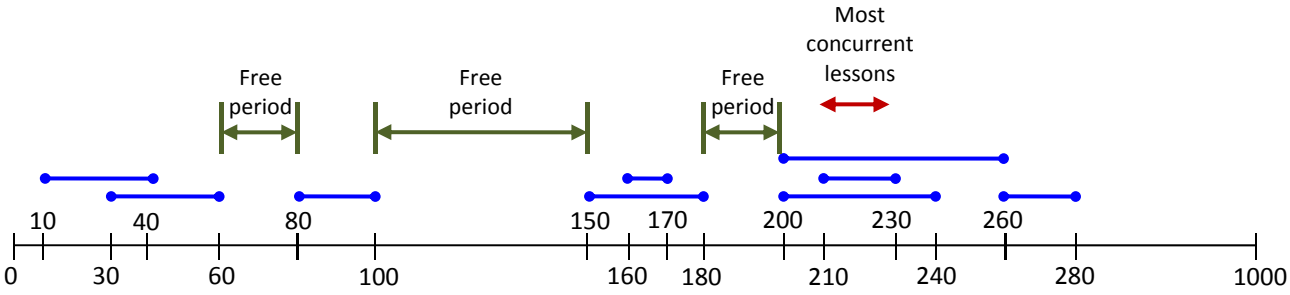
   A skeleton program **schedule_skeleton.c** is given. In it, a **print_intervals()** function is provided for checking purpose.

   The output of the program for the input data shown above is as follows:

```
Duration of longest lesson = 60
Number of free periods = 3
Most number of concurrent lessons = 3
```

Approach 1:

This is a simple approach. To understand this approach, we lay out the intervals read in the example above on a *timeline* as shown below (the timeline is not drawn to scale):



What we do is to translate the information of the intervals into an integer array **timeline**. Let timeline[*k*] stores the number of lessons going on in the period from time *k* to time *k*+1. Hence,

- timeline[0] through timeline[9] contain 0;
- timeline[10] through timeline[29] contain 1;
- timeline[30] through timeline[39] contain 2;
- timeline[40] through timeline[59] contain 1;
- timeline[60] through timeline[79] contain 0;
- etc.

Once this timeline array is set up properly, finding the number of free periods and the most number of concurrent lessons become quite straight-forward. There are 3 free periods as shown in the figure above, and 3 concurrent lessons are going on during the period 210 – 230.

Note that the period 0 – 10 is not considered a free period because it occurs before the first lesson. Likewise, the period 280 – 1000 is not a free period because it happens after the last lesson.

The skeleton program provided is based on this approach. Complete the program.

Approach 2 (optional):

Approach 1 above is straight-forward and easy to code. Under examination condition, this simple approach is acceptable.

For discussion here, we want to explore a better solution.

Suppose the latest finish time is not 1000 but a very large value, or the input data are not integers, approach 1 is not appropriate then. How do we get a more general algorithm?

One approach is to store the interval data in an array of interval structures. To do this, we define a new structure **endpoint_t** as follows:

```
typedef struct {
    int time;
    int type;
} endpoint_t;
```

Each interval is represented by two endpoints: its start endpoint and finish endpoint. Each endpoint is represented by the above structure, where **time** is the time of that endpoint and **type** indicates whether the endpoint is the start or end of the internal. We use 1 for type to represent the start of an interval, and -1 to represent the end of an interval. We may define two macros for readability:

```
#define START 1
#define FINISH -1
```

Hence, using the same example above, the first 2 intervals (200 – 240) and (210 – 230) will be stored in this array of endpoints, let's call this array **endpoints**, as follows:

- endpoints[0].time contains 200; endpoints[0].type contains START (or 1)
- endpoints[1].time contains 240; endpoints[1].type contains FINISH (or -1)
- endpoints[2].time contains 210; endpoints[2].type contains START (or 1)
- endpoints[3].time contains 230; endpoints[3].type contains FINISH (or -1)

Therefore, if there are *n* intervals, there will be 2*n* elements in this array **endpoints**.

With this approach, we can change the data type for **time** if it is no longer integer. It also does not depend on the latest finish time.

The input interval data are read into an array of endpoints. Sorting is required to sort this array in some order, after which we can compute the number of free periods and the most number of concurrent lessons.

Try out this approach.