National University of Singapore
School of Continuing and Lifelong Education
TIC1002: Introduction to Computing and Programming II
Semester II, 2019/2020

Problem Set 1
Structure, Nested Loop, Composite Data Structures

Release date: 16 Jan 2020
Due: 2 Feb 2020, 23:59

Task 1: Building Pyramids [6 marks]

You are the chief monument architect for the Pharaoh Rameses II. Unbeknown to him and the people of Egypt, you are secretly a time traveler with a vital skill: C++ programming (and a tiny portable laptop hidden in your bed chamber). Help bring glory to the people of Egypt by designing the blue print of the various type of pyramids. With C++…. of course. ☺

Part A: Solid Slanted Pyramid

Deliverable: **`void print_solid_slanted_pyramid( int height );`**
We will indicate the main deliverable for each parts, usually one or more functions. You need to follow exactly the definition (name, parameter and return type) for testing purpose. Note that you are allowed (and encouraged) to write addition helper function(s) to modularize your code. Just remember to copy and paste all relevant functions into submission box on Coursemology.

Given an integer **`height`** **(>= 1)**, print a pyramid blueprint with **`height`** **rows of** "#" on screen. Below is a sample output for **`height = 4`**. Newlines are explicitly shown as "⏎" so that you can follow the format closely.

```
#⏎
##⏎
###⏎
####⏎
```

Part B: Solid Symmetric Pyramid

Deliverable: `void print_solid_symmetric_pyramid( int height );`

The slanted pyramid is easy to design, but not very nice to look at. Upgrade the design to make it symmetrical. Below is a sample output for `height = 4`. Spaces are explicitly shown as "▫".

```
▫▫▫#⏎
▫▫###⏎
▫#####⏎
#######⏎
```

Part C: Hollow Symmetric Pyramid

Deliverable:
```
void print_hollow_symmetric_pyramid(
         int height, char outer, char inner );
```

Pharaoh is impressed but not very happy as the solid pyramid has no space for him to inter the sarcophagus (and those tons of treasure)! Well, he is your boss and the boss is always right…..

We will now consider the design of the outer wall `outer`, and the design of inlaid `inner`. Both represented as a character.

Below is a sample output for `height = 4, outer = '#', inner = '$'`.

```
▫▫▫#⏎
▫▫#$#⏎
▫#$$$#⏎
#######⏎
```

Note that the last level of the pyramid is "solid", i.e. only using the outer wall design.

For ease of auto-testing on Coursemology (and reduce your frustration level), note:

1. Follow the output format **exactly**. There is a reason why we show every spaces / newline explicitly.
2. The program template is very simple. You may want to change the main() to read user input using (e.g. using cin or scanf()) for ease of testing.
3. Input restrictions (e.g. "height >= 1") will **always be respected during the auto-testing.** i.e. there is no need to perform input validation (whether input is correct) in all cases.
4. There is **one public test case and two private test cases** per part for this question.

Task 2: I am kinda busy here…..[12 marks]

Nowadays, it is getting harder to live without an online calendar / task manager / to do list or something similar. We will build a (very much) simplified "Appointment Helper" program for this task.

Pause and think
For checking purpose, we have split the problem into separate components for you. Instead of just following the next few parts, take a moment to use top-down design to break down the problem. You can then compare your design with the suggested approach below.

Part A. Today is 2034-01-23! Date Functionalities

Since date consists of **year, month** and **day**, it is a prime candidate for using **C++ structure.** Design and implement the following functions.

Deliverable: A structure for representing a Date. You are free to name the fields and choose the type, but the structure **must be named** `Date`, i.e.

```
struct Date {
    ……
};
```

Deliverable:
```
void make_date( Date& newDate,
                int year, int month, int day);
```

Function initialize the date structure reference `newDate`. The fields are initialized with the parameters (year, month, day) given. You can assume the parameters are always correct (i.e. no negative value; day and month are valid, etc).

Deliverable:
```
int compare_date( Date& dateA, Date& dateB);
```

Function returns an integer to represent the comparison result between `dateA` and `dateB`.

-1: if `dateA` is **earlier** than `dateB`
0: if `dateA` is **the same as** `dateB`
1: if `dateA` is **later** than `dateB`

Part B. Submit at 23:59! Time Functionalities

Similarly, design a structure and related functions to manipulate **time** information. We use a 24-hour format with only hour and minute for simplicity. i.e. 9am = 09 hour 00 min, 9pm = 21 hour 00 min. The functions are very similar to part A.

Deliverable: A structure for representing a Time. You are free to name the fields and choose the type, but the structure **must be named `Time`**, i.e.

```
struct Time {
    ……
};
```

Deliverable: `void make_time( Time& newTime, int hour, int minute );`

Function initialize the time structure reference **`newTime`**. The fields are initialized with the parameters (hour, minute) given. You can assume the parameters are always correct (i.e. no negative value; hour is between 0 to 23, minute is between 0 to 59 etc).

Deliverable: `int compare_time( Time& timeA, Time& timeB );`

Function returns an integer to represent the comparison result between **`timeA`** and **`timeB`**.

-1: if **`timeA`** is **earlier** than **`timeB`**
0: if **`timeA`** is **the same as `timeB`**
1: if **`timeA`** is **later** than **`timeB`**

Part C. Need to meet my client for lunch tomorrow, argh!

Observe a typical appointment: " *'Meet Kylo for lunch' on 2018-01-23 at 11:45, should be done by 13:30*".

Design a structure to hold the information for appointment. **You must reuse Part A and Part B** as much as possible. You can assume a) the description of the appointment (e.g. 'Meet Kylo for lunch') takes at most 80 characters; and b) the start and end time of the appointment falls in the same day.

Deliverable: A structure for representing an Appointment. You are free to name the fields and choose the type, but the structure **must be named `Appointment`**, i.e.

```
struct Appointment {
    ……
};
```

Deliverable:
```
void make_appointment(Appointment& newApt,
        string description,
        int year, int month, int day,
        int startHour, int startMinute,
        int endHour, int endMinute );
```

Function initialize the date structure reference **`newApt`**. The fields are initialized with the parameters given. You can assume the parameters are always correct.

Example: to make the appointment mentioned in the question
```
make_appointment( myApt, "Meet Kylo for Lunch",
                  2018, 1, 23, 11, 45, 13, 30);
```

Deliverable: **`bool has_conflict(Appointment& aptA, Appointment& aptB);`**

Function check whether the appointment **`aptB`** with the appointment **`aptA`**. Put simply, conflicts mean the two appointments cannot be scheduled together (they partially or fully overlaps).

Function returns:

true: there is a conflict between **`aptA`** and **`aptB`**
false: there is **no conflict**

Further Exploration:
We stop short of implementing the last step (a ***schedule*** that includes multiple appointments). You can try to design and implement for your own understanding. ☺

Task 3: Alien in Space! [12 marks]

In this task, we are going to simulate how alien evolve on a planet! The alien planet is represented as an **20 x 20, 2D character array**. Each location (cell) represents whether there is a living alien lifeform on that spot: **'X'** represents a live alien, **'O'** means the location is empty.

Lifeform Evolution on Alien Planet

Each cell in the world can contain a live or a dead (i.e. empty) lifeform. To evolve from one generation **G** to the next **G+1** generation, each cell will interacts with its eight *neighbour cells*, which are the cells that are horizontally, vertically, or diagonally adjacent. The following rules[1] are used to determine the status of the cell **in the next generation**:

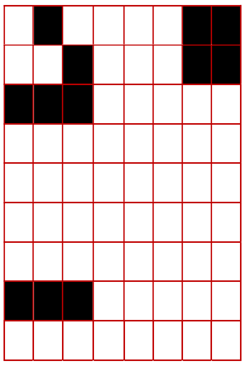
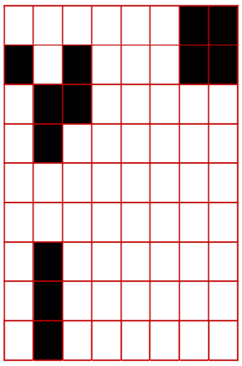
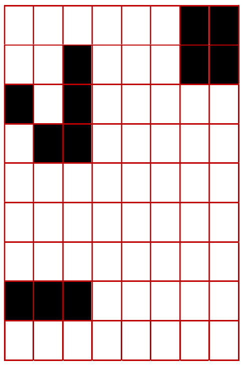
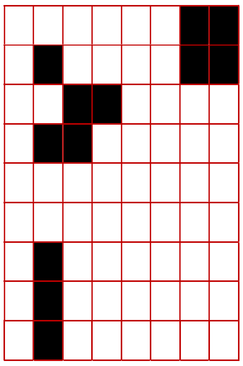
1. [**Under-population**] Any live cell with fewer than two live neighbours dies.
2. [**Survive**] Any live cell with two or three live neighbours lives on to the next generation.
3. [**Overpopulation**] Any live cell with more than three live neighbours dies.
4. [**Reproduction**] Any dead cell with exactly three live neighbours becomes a live cell.

For example, the middle cell (shaded red) can change according to the rules from one generation to the next (Shaded cell = alive, Non-shaded cell = dead (empty)):

	Rule 1	**Rule 2**	**Rule 3**	**Rule 4**
Generation G	🟫⬜⬜ ⬜🟥⬜ ⬜⬜⬜	🟫⬜⬜ ⬜🟥🟫 ⬜⬜⬜	🟫⬜⬜ ⬜🟥🟫 🟫⬜🟫	🟫⬜⬜ 🟫⬜⬜ ⬜🟫⬜
Generation G+1	🟫⬜⬜ ⬜⬜⬜ ⬜⬜⬜	🟫⬜⬜ ⬜🟥🟫 ⬜⬜⬜	🟫⬜⬜ ⬜⬜🟫 🟫⬜🟫	🟫⬜⬜ 🟫🟥⬜ ⬜🟫⬜

[1] This is essentially **Conway's Game of Life** ruleset

Note that the rules are applied to every single cell in the world. For example, below showed 3 rounds of evolutions of a 9 x 8 world:

Generation 0	**Generation 1**	**Generation 2**	**Generation 3**
			

Given function: `void init_alien_planet( const char filename[],`
`char alienPlanet[][20]);`

Function to open and read from a text file with `filename` to initialize the alien planet (the 2D array). The file contains **20 lines where each line contain 20 characters ('O' or 'X').** The information represent the "Generation 0" of the planet.

For example, `init_alien_planet( "planet_sample.txt", myPlanet);` should read from the sample file "`planet_sample.txt`" and initialize `myPlanet`, which should be declared as 20 x 20, 2D character array.

This function has already been implemented for you. Use it to review / learn C++ file stream so that you can implement the `save_alien_planet()` function below.

Deliverable: `void evolve_alien_planet(`
`char alienPlanet[][20], int nGeneration);`

Function evolve the planet according to the rules for `nGeneration` generations.

For example, `evolve_alien_planet( myPlanet, 3);` should evolve the planet for **3 rounds, (i.e. generation 0 → generation 3).**

Deliverable: `void save_alien_planet( const char filename[],`
`char alienPlanet[][20]););`

Function save (print out) the alien planet information into a file with `filename`. The output file format should be the same as the input file, i.e. 20 lines of 20 characters each.

Note: We will check this part by using the output file generated via `save_alien_planet`(), so make sure it is implemented **correctly with exact formatting**.

You are strongly encouraged to write additional helper functions, e.g. printing function for checking this part. The size 20 x 20 is chosen so that the planet readable on screen.

We have generated the first 3 generation of the planet for your reference. They are named " "`planet_sample_gen_1.txt`", "`planet_sample_gen_2.txt`"and "`planet_sample_gen_3.txt`"

**~~~ End of Problem Set ~~~**