C Programming II
2024 Spring
Homework 01

Instructor: Po-Wen Chi

Due: 2024.03.26 PM 11:59

Policies:

- **Zero tolerance** for late submission.
- **Plagiarism is not allowed.** Both source and copycat will be **zero**.
- You need to prepare a README file about how to make and run your program. Moreover, you need to provide your name and your student ID in the README file.
  - Your Name and Your ID.
  - The functional description for each code.
  - Anything special.
- Please pack all your submissions in one zip file.
- For convenience, your executable programs must be named following the rule hwXXYY, where the red part is the homework number and the blue part is the problem number. For example, hw0102 is the executable program for homework #1 problem 2.
- I only accept **PDF**. MS Word is not allowed.
- Do not forget your Makefile. For convenience, each assignment needs only one Makefile.

1 My String Library (20 pts)

In this class, I have shown you some standard string functions. Of course, you should use them when coding. However, sometimes you may need some functions that are not included in the standard string library. Do not worry, you can implement on your own. For your practice, I want you to implement some existing string functions in your own way.

```c
char *mystrchr(const char *s, int c);
char *mystrrchr(const char *s, int c);
size_t mystrspn(const char *s, const char *accept);
size_t mystrcspn(const char *s, const char *reject);
char *mystrpbrk(const char *s, const char *accept);
char *mystrstr(const char *haystack, const char *needle);
char *mystrtok(char *str, const char *delim);
```

The usage of these functions should be the same with the standard version, including their return values. All requirements are in the manual. You need to prepare **mystring.h**, and TA will prepare **hw0101.c**. Of course, Makefile is your own business. Do not forget to make **hw0101.c** in your Makefile.

You cannot call the corresponding standard functions directly in your implementation.

2 String Calculator (20 pts)

Please develop a function where given an arithmetic expression and a base, the function should generate the arithmetic result according to the given base.

```c
// Input:
//   char *pExpr: the arithmetic expression.
//   int32_t base: the base that is used to show the arithmetic result. (2-16)
// Output:
//   char **ppResult: the arithmetic result string.
// Return:
//   0: Success; -1: Error input
int32_t calculate( char *pExpr, int32_t base, char **ppResult );
```

Some notes:

- The input string format will be as follows:
  <Operand 1> (Operator 1) <Operand 2> (Operator 2) <Operand 3> ...
  - Operand format: digits_base. Example: "123_10", "BEEF_16".
  - Operator: +, -, *
  - For your simplicity, you do NOT need to consider parentheses.
  - If the format is not valid, return -1.
- Example:
  - *pExpr: "ABC_16 + 00001_2"
  - base = 10
  - Output string: "2749_10"
- For your simplicity, if the result is negative, you just print '-' followed by the positive number.
  Example: -2_10
- Do not forget to **allocate** the output memory.

You need to prepare **mycal.h**, and TA will prepare **hw0102.c**. Of course, Makefile is your own business. Do not forget to make **hw0102.c** in your Makefile.

3 Chain Rule (20 pts)

You all know what **chain rule** is, right? In calculus, the chain rule is a formula that expresses the derivative of the composition of two differentiable functions. If a variable z depends on the variable y, which itself depends on the variable x, the chain rule is expressed as

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

In this problem, I want you to develop a function to the derivative result. Please see the following codes.

```
#include <stdint.h>

// Polynomial structures
// Example:
//    f(x) = x^3-2x+1
//    size = 3
//    pPowers: [3,1,0]
//    pCoefficients: [1,-2,1]
typedef struct _sPoly
{
  uint32_t size;

  uint32_t *pPowers;
  int32_t  *pCoefficients;
} sPoly;

// Input:
//    *pFy: z = f1(y);
//    *pFx: y = f2(x);
//    Note that pPower can be in any orders.
// Output:
//    *pResult = dz/dx
//    Note that pPower should be in the descending order.
// Return:
//    0: Success; -1: Error input
int32_t chain_rule( sPoly *pResult, const sPoly *pFy, const sPoly *pFx );
```

There is a simple example.

$$z = f_1(y) = y^2, y = f_2(x) = 4x - 3, \frac{dz}{dx} = 2(4x-3)^1 \cdot 4 = 32x - 24.$$

```
// *pFy:
//    size = 1
//    pPowers: [2]
//    pCoefficients: [1]
// *pFx:
```

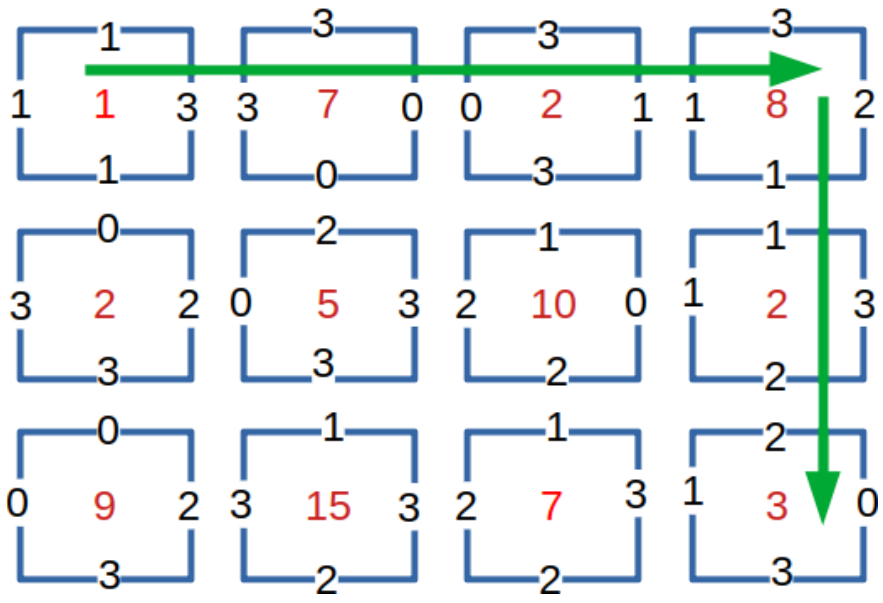

Figure 1: The minimum cost is 23.

```
//     size = 2
//     pPowers: [1,0]
//     pCoefficients: [4,-3]
// *pResult:
//     size = 2
//     pPowers: [1,0]
//     pCoefficients: [32,-24]
```

For your simplicity, I promise there will be no overflow issue in the power and the coefficient. You need to prepare **mychain.h** which includes the above codes, and TA will prepare **hw0103.c**. Of course, Makefile is your own business. Do not forget to make **hw0103.c** in your Makefile.

4 Maze (20 pts)

Given a maze as follows. The maze is composed of many rooms and you can go from one room to another if and only if the corresponding doors of these two rooms are equal. Figure 1 is an example. Please find a path with the least cost from the start point to the end.

```
#include <stdint.h>

typedef struct _sRoom
{
  uint32_t cost;
  uint8_t  doors; // 8 bits: aabbccdd
                  // aa: the up door number 0-3
                  // bb: the right door number 0-3
                  // cc: the down number 0-3
                  // dd: the left door number 0-3
} sRoom;
```

```c

typedef struct _sPoint
{
  uint32_t row;
  uint32_t col;
} sPoint;

typedef struct _sPath
{
  uint32_t  length;   // Path length.
  uint32_t  cost;   // Cost
  sPoint    *pPath;   // An array of all points in order.
} sPath;

// The start point is pMaze[0][0] and the exit point is pMaze[row-1][col-1]
// If there is no path, return 0; If there is any errors in inputs, return -1;
    otherwise, return 1;
int32_t find_min_path( const sRoom *pMaze, const uint8_t row, const uint8_t
   col, sPath *pMinPath );
```

For your simplicity, I promise there will be only one min path. You need to prepare **mymaze.h** which includes the above codes, and TA will prepare **hw0104.c**. Of course, Makefile is your own business. Do not forget to make **hw0104.c** in your Makefile.

5 Taiko Music Generator (20 pts)

Have you ever heard of or played the game Taiko no Tatsujin(太鼓の達人)? Here is how to game playlink. It's a rhythm game where players hit taiko-notes by striking the center or rim of a taiko drum. You might have seen taiko arcade outside, such as the ones near our school in Gongguan. But did you know that this game is also available on various platforms like Nintendo, PS, mobile apps (RHYTHM CONNECTlink), and even on computers through simulators (Taiko-san Daijiro)? Notably, simulators allow users to play their own taiko music by creating custom chart(譜面) and music.

In the simulator, the game is run by reading in .tja files, which are considered charts controlling the taiko-notes. However, the contents of tja files are not very intuitive. Therefore, this time, I want you to develop program which can convert tja files into a json file that represents the music and taiko-notes's timeline using timestamp notation. In other words, the goal is to map the taiko-notes to specific seconds on the music timeline.

5.1 tja file

(See this websitelink and download samplelink tja file) To make it easier for you to analyze tja files, I also provide a simple explanation. This is the typical structure of a tja file:

```
// ================================================
//                   Header Area
//  Header area set the tja global and inital setting
// ================================================
TITLE
```

Figure 2: Taiko no Tatsujin game

```
.
.
.
// ====================================================

// ====================================================
//                    Body Area 1
//           Body area set the music chart value
// ====================================================
COURSE
// Music and Chart Setting in this course
.
#START
// Chart Contents
.
.
#END
// ====================================================

// ====================================================
//                    Body Area n
//         Body amount is be decide according to
//               how many course does it have
//                 Range from 1 ~ 5 totally
// ====================================================
COURSE
// Music and Chart Setting in this course
.
#START
// Chart Contents
```

```
.
.
#END
// ================================================
```

You only need to get the values corresponding to the following key (ignore the others):

- BPM:[float]
- OFFSET:[float]
- COURSE:[string|uint] // Easy:0, Normal:1, Hard:2, Oni:3, Edit:4
- #START
- #MEASURE beat[uint]/note[uint]
- #BPMCHANGE [float]
- #END
- xxxxxxxxx, // chart content

5.2 Output Timeline Chart json

To output as a chart json, here is the following is the json format structure:

```
{
  "data": [
        {
            "course": "taiko difficulty number [uint]",
            "chart": [
                ["taiko-note number [uint]", "timestamp [float]"],
                                ...
            ]
        },
        {
            "course": ...
            "chart": [
                ...
            ]
        },
        ...
    ]
}
```

You just need to detect these taiko-notes (Figure 3):

- DON(Red): 1
- KA(Blue): 2
- BIGDON(Big Red): 3

- BIGKA(Big Blue): 4
- Blank(Nothing): 0 (5, 6, 7, 8, 9)

Figure 3: Taiko Notes

5.3 Convert tja file into Timeline Chart

After understanding which values to extract from the tja files and knowing their principles, in order to map taiko-notes to the music timeline, you need to understand a formula algorithm:

$$duration = \frac{60}{bpm} * \frac{beat}{Length(chart)} * \frac{4}{note}$$

You must track the BPM, beat, and note value in the chart to determine the duration. There's tja file example:

```
// 怪物.tja
BPM:170
OFFSET:-1.646
COURSE:Normal

#START
#MEASURE 4/4
1011, // duration = (60/170)*(4/4)*(4/4) ~= 0.352
```

Then it looks like Figure 4[link].

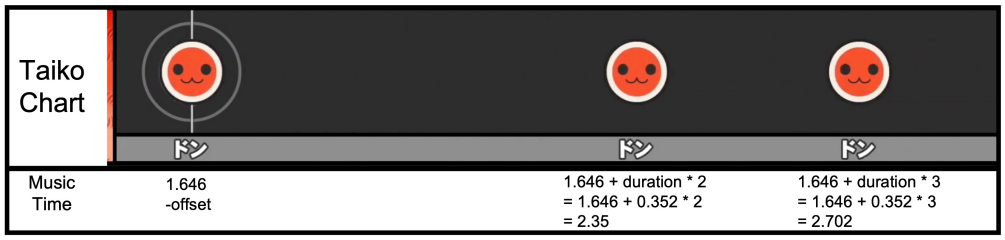

Figure 4: Timeline Chart Example

And chart.json looks like:

```
{
  "data": [
        {
            "course": 1,
            "chart": [
                [1, 1.646],
                [1, 2.35],
                [1, 2.702]
            ]
        }
    ]
}
```

Before calculating duration, beware that if $Length(chart) < beat$, then you have to extend the length of chart content to $lcm(Length(chart), beat)$. But if $Length(chart) = 0$, extend 0's number of beat:

```
#MEASURE 6/4
1, -> 100000, duration = 0.4
, -> 000000, duration = 0.4
02, -> 000200, duration = 0.4
102, -> 100020, duration = 0.4
2401, -> 200400000100, duration = 0.2
```

So if bpm = 150 and offset = 0 currently. And that is to be:

```
"chart": [
    [1, 0],
    [2, 6],
    [1, 7.2],
    [2, 8.8],
    [2, 9.6],
    [4, 10.2],
    [1, 11.4]
]
```

5.4 Generate Taiko Music

You can go in this website[link] website and generate taiko music by inputting the chart json file and the music ogg file. You can see the Figure 5 example.

5.5 Example by Running Program

Again, the purpose of your program is to convert the contents of tja files into output the contents of chart json files. There's the input and output example following below:

5.5.1 Input

```
TITLE:
SUBTITLE:
BPM:170
WAVE:
```

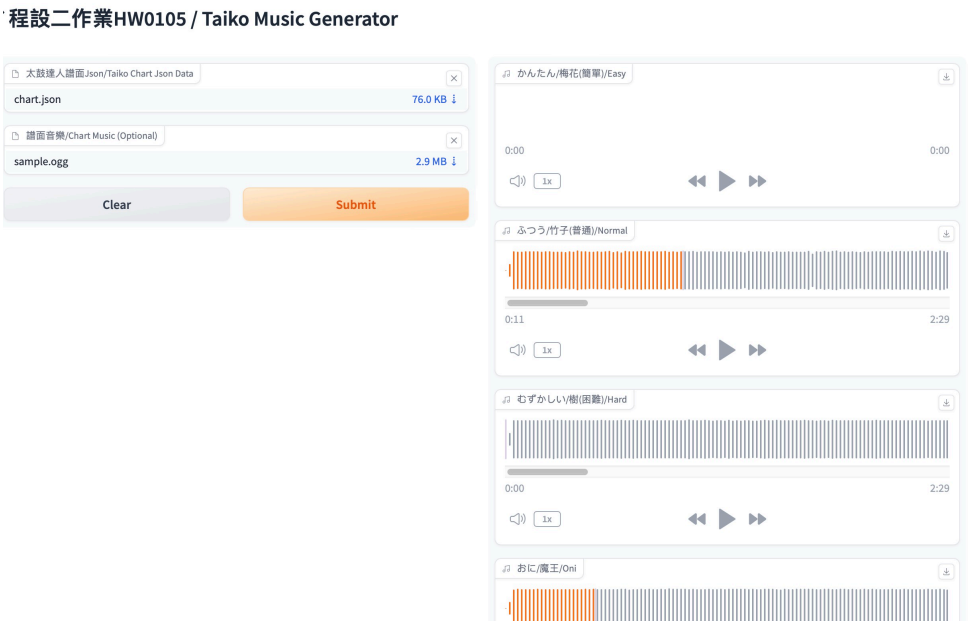

Figure 5: Taiko Music Generator Website

```
OFFSET:-1.646
DEMOSTART:35.15
SONGVOL:100
SEVOL:100


COURSE:Oni
LEVEL:7
BALLOON:2
SCOREINIT:
SCOREDIFF:
#START
11211122,
,
#END


COURSE:Hard
LEVEL:6
BALLOON:2,24
SCOREINIT:
SCOREDIFF:
#START
10101212,
#END

COURSE:Normal
LEVEL:5
BALLOON:14,7,18,7
SCOREINIT:930,4870
```

```
SCOREDIFF:305

#START
1011,
10010101,
,
#END


COURSE:Easy
LEVEL:3
BALLOON:11,5,14,5
SCOREINIT:
SCOREDIFF:
#START
1,
1,
1,
12,
,
#END
```

5.5.2 Output

```
{
  "data": [
    {
      "course": 3,
      "chart": [
        [1, 1.646000],
        [1, 1.822471],
        [2, 1.998941],
        [1, 2.175412],
        [1, 2.351882],
        [1, 2.528353],
        [2, 2.704823],
        [2, 2.881294]
      ]
    },
    {
      "course": 2,
      "chart": [
        [1, 1.646000],
        [1, 1.998941],
        [1, 2.351882],
        [2, 2.528353],
        [1, 2.704823],
        [2, 2.881294]
      ]
    },
    {
      "course": 1,
      "chart": [
```

```
        [1, 1.646000],
        [1, 2.351882],
        [1, 2.704824],
        [1, 3.057765],
        [1, 3.587177],
        [1, 3.940118],
        [1, 4.293059]
      ]
    },
    {
      "course": 0,
      "chart": [
        [1, 1.646000],
        [1, 3.057765],
        [1, 4.469530],
        [1, 5.881294],
        [2, 6.587176]
      ]
    }
  ]
}
```

If there is an error issue occurred, you have to output nothing and exit the program. For your convenience, I'll provide some files[link] for your development and testing:

- taiko.h (optional to use)
- sample tja and ogg files
- sample chart json files
- urls of people playing Taiko no Tatsujin on YouTube (for validating the taiko music that you generate)

6 Bonus: _BitInt(N) (5 pts)

In C23, there are new types, **_BitInt(N)** and **unsigned _BitInt(N)**, for bit-precise integers. Please answer the following questions with small example codes for your answer verification.

- The **sizeof** result for these new types.
- What will happen if you assign a value to an integer which is over the given size?
- Is it possible to do arithmetic operations between two operands with different bits?
- Is it possible to do arithmetic operations between **_BitInt(N)** and **unsigned _BitInt(N)**? If yes, the result is **_BitInt(N)** or **unsigned _BitInt(N)**?
- What will happen if N is greater than BITINT_MAXWIDTH?