VG101 Mid 2 Review Part3

Author: Lu Haorong

Pretend we also have a cat!

Pioneers

Usually in part 3 we will discuss the solution to exam questions from previous years, but in this year please forget everything about the past. You are the creators of VG101 history!

Take It Easy

Three computer-based C/C++ exams I experienced in JI,

- FA2018 VG101 Final, Mean = 38/100,
- SU2020 VE280 Mid, Mean = 51 / 120,
- SU2020 VE280 Final, Mean = 71.5 / 120.

But this time,

- **FA2021 VG101 Mid2, Mean = 80/100 (Estimated by Jigang)**

So the problems should be easy, don't worry too much. Some boring but always proper reminders,

- Read carefully through the problem description,
- Think about the boundary/extreme cases (Especially if you want to get 95+),
- Always compile and test your program!

There are still 6 days before mid2, I think we have some time to go back to one month ago...

```
Time time = current_time();
time.month--;
```

Lab 3

Tutorial

- Basic data types and operators in C.
- I think you are all familiar with them now.

Mathsh

Before starting writing some codes, it's better to think about the **skeleton and procedure**.

- Print the prompt, and keep waiting for user inputs
  - `while(1)`, and `printf` something at the beginning of each iteration
- Read the user input, and execute according to the command name
  - Input format: `max 3 1 2 3`, `min 2 1 2`, `fib 30`, ...
  - As the number of arguments can be accurately predicted, and all arguments are split by one space, it's proper to use a `for` loop plus `scanf()` to read the arguments.

```
/* Skeleton of Mathsh */
char command[5];          // Command name
int n;                    // Number of arguments
while(1) {                // Keep reading user inputs
    printf("mashth $ ");  // Print the prompt
    scanf("%s", command); // Read the command name
    if (!strcmp(command, "exit")) {
        // ...
        break;            // The only valid exit in this program
    }
    else if (!strcmp(command, "max")) {
        scanf("%d", &n);       // Read number of arguments
        int num;               // Actual integer
        for(int i = 0;i < n;++i) {
            scanf("%d", &num); // Read each integer
            // calculate ...
```

```
            }
        }
        // Other commands ...
    }
```

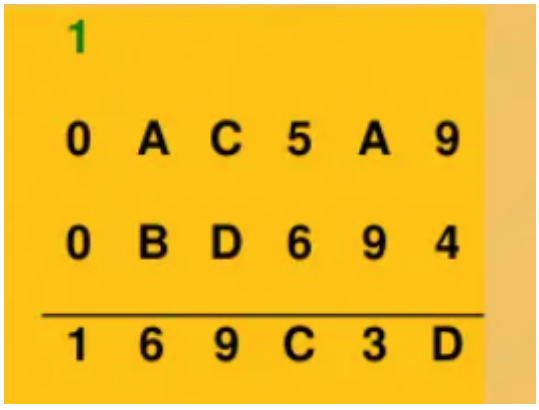
- How to calculate the result without an array?

```c
// Basic calculation of max
int max = -1000000, num;
for (int i = 0; i < n; ++i) {
    scanf("%d", &num);
    max = (max < num) ? num : max;
}
printf("max: %d\n", max);

// Use sum to calculate avg
double sum = 0, num;
for (int i = 0; i < n; ++i) {
    scanf("%lf", &num); // Use %lf to read a double in scanf
    sum += num;
}
printf("avg: %f\n", sum / (double)n);
```

Hw 4

- Intuition: Convert hex numbers into decimal form, calculate the sum, and convert it back to hex form.
- Feasibility: "The number of digits is less than 255", namely at most $2^{255} - 2$
  - The upper limit of integer type in C: `unsigned long long`, at most $2^{64} - 1$
  - Therefore, it's not feasible!
- Another approach: **Simulate hexadecimal columnar addition**



- Pseudocode, assume the lengths of two num are equivalent:

```
int i = 0, inc = 0;
while (i < digit) {
    sum[i++] = (inc + num1[i] + num2[i]) % 16;
    inc = (inc + num1[i] + num2[i]) / 16;
}
if (inc != 0) {
    sum[i] = inc;
}
```

- But in reality, there are more things you need to deal with:
- How to **convert "hex in `char`" to "decimal in `int`"**?
    - A possible approach: A helper function,
    - ```
      int hex2dec (char digit) {
          if (digit >= '0' && digit <= '9') {
              return digit - '0';
          } else if (digit >= 'A' && digit <= 'F') {
              return digit - 'A' + 10;
          }
          return -1;
      }
      ```
- How to **convert "decimal in `int`" to "hex in `char`"**?
    - A possible approach: A char array (wordbank)
    - ```
      char hex[17] = "0123456789ABCDEF";
      ```
- How to deal with the **overflow part and reverse order**?
    - It's hard to illustrate... Just read the code!
    - ```
      int n1 = strlen(hex1) - 1, n2 = strlen(hex2) - 1; // Start from
      the end of two numbers
      int i = 0, inc = 0;

      while (n1 >= 0 && n2 >= 0) { // The align part of two numbers
          int digitSum = hex2dec(num1[n1]) + hex2dec(num2[n2]) + inc;
          tmpsum[i++] = hex[digitSum % 16]
          inc = digitSum / 16;
          --n1;
          --n2;
      }
      while (n1 >= 0) { // If num1 is longer than num2
          int digitSum = hex2dec(num1[n1]) + inc;
          tmpsum[i++] = hex[digitSum % 16]
          inc = digitSum / 16;
          --n1;
      }
      while (n2 >= 0) { // If num2 is longer than num1
      ```
      ```

```
        int digitSum = hex2dec(num2[n2]) + inc;
        tmpsum[i++] = hex[digitSum % 16]
        inc = digitSum / 16;
        --n2;
    }
    if (inc != 0) {  // The remaining carry digit
        tmpsum[i++] = hex[inc];
    }
    // Reverse the final answer
    // Here I use a new array for simplicity
    for (int j = 0; j < i; ++j) {
        hexsum[j] = tmpsum[i - 1 - j];
    }
```

- Some reminders left:
    - When creating a fixed-length array, it's better to reserve some spaces to prevent potential overflow.
    - And do not forget to initialize.
    - ```
      #define MAX_LENGTH 300 // Be generous!
      char num1[MAX_LENGTH], num2[MAX_LENGTH];
      char tmpsum[MAX_LENGTH] = {0}, hexsum[MAX_LENGTH] = {0}; //
      Remember to iniliaze the array!
      ```

Lab 4

- The most difficult lab so far 😨
- General procedure (Note that the second and third steps can be combined into one step)
    - Read from file (*easy*)
    - Parse the command into Infix Expression (*hard and tricky*)
    - Convert Infix to Reverse Polish Expression (*moderate*)
    - Calculate the result of Reverse Polish Expression (*moderate*)
    - Print result / Assign result to variables (*moderate ~ hard*)
- How to read from file?
- Intuition: Only variables may affect other lines, in the other cases we only need focus on one line, therefore we can initialize the variable data structure outside the loop, and then deal with the file line by line,
      ```

- ```
  // Skeleton
  int main() {
      char line[MAX_LENGTH];
      FILE *fp = fopen("commands.txt", "r");
      Variables vars = {.varNames = {{0}}, .varNums = 0}; // Here I
  put vars and varName into one struct

      while(fgets(line, MAX_LENGTH, fp)) {
          if(line[strlen(line) - 1] == '\n') {
              line[strlen(line) - 1] = '\0'; // Get rid of the
  annoying \n
          }
          // ...
      }
  }
  ```

- How to parse the command?
- First general problem: The annoying **SPACE**! How to deal with it?
- From special to general (Some approaches):
  - Delete all spaces!
  - ```
    char parLine[MAX_LENGTH];
    int index = 0;
    for (size_t i = 0;i < strlen(line);++i) {
        if(line[i] == ' ') {
            continue;
        }
        parLine[index++] = line[i];
    }
    ```
  - or Add space on both sides of each operator! (I choose this one)
  - ```
    char parLine[MAX_LENGTH * 2];
    int index = 0;
    bool isPrevOp = false;
    for (size_t i = 0; i < strlen(line); ++i) {
        // Special Case: '-' is minus or negative?
        // If the previous one is an operator, then this '-' must a
    negative sign
        // Or '-' is the first sign
        if (line[i] == '-' && (isPrevOp || i == 0)) {
            parLine[index++] = line[i];
        } else if (isOperator(line[i])) { // if line[i] is the
    opreator, add two spaces
            parLine[index++] = ' ';
            parLine[index++] = line[i];
            parLine[index++] = ' ';
            isPrevOp = true;
    ```
    ```

```
    } else {
        parLine[pi++] = line[i];
        // If this is the space, keep isPrevOp unchanged,
otherwise make isPrevOp false
        isPrevOp &= (line[i] == ' ');
    }
}
```

- A easy road...
  - If you split everything by at least one space (the second approach above), then it's easy to change it into infix
  - Just keep calling `strtok()`, and let it go to the correct place
  - ```
    // structure of infix expression
    typedef struct {
        double data[MAX_LENGTH];
        int state[MAX_LENGTH];
        int top;
    } InFix;

    char *token = strtok(parLine, " ");
    while (token != NULL) {
        if (isDigit(token[0]) || (token[0] == '-' && strlen(token)
    > 1)) { // a number
            infix.data[infix.top] = strtod(token, NULL); // convert
    it from string to double
            infix.state[infix.top++] = INFIX_DOUBLE;
        } else if (isOperator(token[0])) {               // an
    operator
            infix.data[infix.top] = token[0];            //
    implicit conversion from char to double
            infix.state[infix.top++] = INFIX_OPERATOR;
        } else {                                         // a
    variable
            infix.data[infix.top] =
    vars.varValues[getVarIndex(&vars, token)]; // get variable
    value
            infix.state[infix.top++] = INFIX_DOUBLE;
        }
        token = strtok(NULL, " ");
    }
    ```
  - Then just follow the procedure on the lab4 description, convert it to infix expression and calculate the result, life becomes easier!
- A hard road...
  - If you choose the first approach, then it still takes some trick to get everything out,
    ```

```
int i = 0, index = 0;
while (i < strlen(line)) {
    if(isDigit(line[i])) { // line[i] is a digit
        char* start = line + i; // record the start point
        while(isDigit(line[i])) { // keep i++ until line[i] is
not a digit
            ++i;
        }
        char* end = line + i - 1; // record the end point
        infix[index] = strtod(start, &end); // convert to
double (thanks strtod!)
        helper[index++] = INFIX_DOUBLE;
    } else if (isOperator(line[i])){ // line[i] is an operator
        infix[index] = line[i++];
        helper[index++] = INFIX_OPERATOR;
    } else if (isAlpha(line[i]) || line[i] == '_') { // line[i]
seems to be a variable!
        char substr[MAX_LENGTH] = {0};
        int subIndex = 0;
        while(isAlpha(line[i]) || line[i] == '_' ||
isDigit(line[i])) {
            // keep assign line[i] to substr and i++,
            // until line[i] is none of [a-z], [0-9], _
            substr[subIndex++] = line[i++];
        }
        infix[index] = vars.varValues[getVarIndex(&vars,
substr)]; // get value by substr
        helper[index++] = INFIX_DOUBLE;
    }
}
```

- Note that here all operations is based on `i`, be careful about the value of `i`, it's your only index on `line`!
- Then just follow the procedure on the lab4 description, convert it to infix expression and calculate the result, life becomes easier!

- The above part assumes there is no `fprintf` or `=`, but actually there are three cases:
  - No equal sign `=`: `2 *(1 + 4) / 2`, `cx + 9`
  - Assign Equation: `a = 10 * 8 + 6`, `cx = a + 7`, `cx = cx + 9`
  - `Fprintf`: `fprintf("I hate VG101!\n")`, `fprintf("I hate VG%f!\n", cx + 9)`

- For the second case, actually it's same as the first case, except for the `${variable name} = ...`
  - You can use `strchr` to determine the position of `=`, and treat everything on right as the `line` above

- For the left part, you need first to check if it is already created,

```
// Here we suppose `token` is the string of variable name
int varIndex = getVarIndex(&vars, token);
if (varIndex == -1) {                              // Never been
created
    varIndex = vars.varNums;                       // Create the
variable
    strcpy(vars.varNames[vars.varNums++], token); // Give it
name
}
```

- Then after the calculation, do not forget to assign the result to it,

```
vars.varValues[varIndex] = result;
```

Hw 5

- Somehow simpler than HW4, but be careful about the extreme/boundary cases.
- Just find the char-to-char relationship between the original string and the new string.
- Key point 1: `new_str[(i + shift) % n] = str[i];` (Shift to right, `n` is the length of `str`)
- Key point 2: `shift` can be negative, and it's absolute value can be larger than the length of `str`!
- Typical Issues:

```
char *strshift(char *str, int shift) {
    int n = strlen(str);
    for (int i = 0;i < shift;++i) {
        // ...
        // In the worst case, this loop will iterates 2^31 times,
which is unacceptable
    }
    // ...
}

char *strshift(char *str, int shift) {
    // If shift is -2^31, while the upper limit of int is 2^31 - 1,
what will happen?
    if (shift < 0) {
        shift = -shift;
    }
    // or
    shift = abs(shift);
```

```
}
```

- Is there a way to not distinguish between the direction (left or right) of `shift`?
- Yes! The idea is, **given a string of length 4, "moving it to left by 1" equals to "moving it to right by 3"**
    - Implementation:
    - ```
      int n = strlen(str);
      shift %= n;
      if (shift < 0) {
          shift += n; // (negative % postive) will give a negative value in C, like -7 % 3 == -1
      }

      // You can combine those into one equation
      shift = ((shift % n) + n) % n;

      // Then you only to consider shifting the string to right!
      ```
- Challenge: How to deal with this problem in $O(1)$ space? (i.e. do not create a new array)
    - A problem that my roommate just encountered in his Bytedance interview
    - Just for fun :)

Hw 6

- Although its deadline has not passed, most of you have done a good job!
- **Do not copy the codes here, TA is watching you :)**
- Thanks for kind Jigang, this time all the testcases are gentle (i.e. at most 100 numbers, no negative number, no numbers are identical...)
- But there are a lot of approaches, which one is better?
- Simple approach: sort the array in ascending order, and then return the second element,
      ```

- ```
  int cmp(const void *lhs, const void *rhs) { // A sample compare function for int array
      return (*(int*)lhs) > (*(int*)rhs);
  }

  int SecondSmallest(int array[], int n) {
      /* find the second-smallest number in the array, where n is the number of elements */
      qsort(array, n, sizeof(int), cmp); // A sample qsort() call for int array
      return arr[1];
  }

  // Complexity: O(nlogn), or O(n^2) if you use bubble/selection/insertion sort
  ```

- Better approach: Find the minimum of the array, delete it, and then find the minimum again

- ```
  int SecondSmallest(int array[], int n) {
      /* find the second-smallest number in the array, where n is the number of elements */
      int min = -1, index = -1;
      for(int i = 0;i < n;++i) {
          if(array[i] < min) {
              min = array[i];
              index = i;
          }
      }
      array[index] = INT_MAX; // delete the minimum by change it to INT_MAX
      min = -1; // reset min
      for(int i = 0;i < n;++i) {
          if(array[i] < min) {
              min = array[i];
          }
      }
      return min;
  }

  // Complexity: O(n), traverse the array twice
  ```

- Much Better Approach: **Maintain the smallest and second smallest numbers so far in one traverse.**
- Complexity: $O(n)$, only one traverse
- The following is paid content 😁 You can check it free after Nov.18!

Lab 5

The following is paid content 😁 You can check it free after Nov.18!