

3장 프로그래밍 실습_C 코드

[프로그래밍 실습 2]

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <io.h>
#include <fcntl.h>

#pragma warning(disable: 4996) // 비주얼 스튜디오 2019에서의 4996 워닝 해제
#define EXP_SIZE 50           // 합성명제의 길이를 위한 상수
#define OPERATOR_NUMBER 10    // 명제 변수의 개수를 위한 상수
#define MAX_SIZE_TABLE 26     // 논리표의 크기를 위한 상수

int size = 0;                // 연산자의 크기를 위한 변수
char keys[MAX_SIZE_TABLE];   // 연산자를 저장하기 위한 문자형 배열
int values[MAX_SIZE_TABLE];   // 비연산자를 저장하기 위한 배열 (0으로 초기화)
int NUM_OF_OPERAND;          // 비연산자의 개수를 위한 변수
int NUM_OF_OPERATOR;         // 연산자의 개수를 위한 변수

int** result_table;          // 결과 논리표를 저장하기 위한 메모리 공간
char** expression_result;    // 계산된 합성명제를 저장하기 위한 메모리 공간

// 연산자를 위한 구조체
typedef struct {
    char op[EXP_SIZE];
    int top;
}Operator;

// 계산될 논리값을 위한 구조체
typedef struct {
    char exp[EXP_SIZE][EXP_SIZE];
    int value[EXP_SIZE][1 << OPERATOR_NUMBER];
    int top;
}Value;

// 정의된 함수들에 대한 함수 원형
bool check_and_despace(char, int);
int getIndex(char);
void insert(char, int);
int get(char);
char* addBinary(char*, char*);
int** create_initial_table(int);
void push_op(Operator*, char);
void push_val(Value*, char);
char pop_op(Operator*);
Value* pop_val(Value*);
char peek_op(Operator* stack);
void main_operation(char*);
void display_and_writeOnFile();

bool check_and_despace(char exp[EXP_SIZE], int num) {
    int len = strlen(exp) - 1;
```

```

int i = len;
while (exp[i--] == ' '); // 끝에서 연속 공백을 건너뛰
int hash[26] = { 0 };
len = ++i;
while (i >= 0) {
    if (exp[i] == ' ') {
        int j = i;
        int k = len;
        while (j < strlen(exp)) {
            exp[j] = exp[k];
            k++, j++;
        }
        len = j;
    }
    else if (!(exp[i] >= 65 && exp[i] <= 90) || exp[i] == '|' || exp[i] == '&' || exp[i]
== '!' || exp[i] == '>' || exp[i] == '(' || exp[i] == ')')) {
        return false;
    }
    else if ((exp[i] >= 65 && exp[i] <= 90) && !hash[exp[i] - 65]) {
        hash[exp[i] - 65] = 1;
        num--;
    }
    else if (exp[i] == '|' || exp[i] == '&' || exp[i] == '!' || exp[i] == '>') {
        NUM_OF_OPERATOR++;
    }

    len = i;
    i--;
}
if (num == 0)
    return true;
return false;
}

// 배열에서 해당 키의 인덱스를 반환하는 함수
int getIndex(char key)
{
    for (int i = 0; i < size; i++) {
        if (keys[i] == key) {
            return i;
        }
    }
    return -1; // 키가 없을 때
}

// 특정 키에 값을 매핑하는 함수
void insert(char key, int value)
{
    int index = getIndex(key);
    if (index == -1) { // 키가 없으면
        keys[size] = key;
        values[size] = value;
        size++;
    }
    else { // 키가 있을 때 값을 반환
        values[index] = value;
    }
}

```

```

// 맵(map)에서 해당 키의 값을 반환하는 함수
int get(char key[])
{
    int index = getIndex(key);
    if (index == -1) {    // 키가 없으면
        return -1;
    }
    else {    // 키가 있을 때 값을 반환
        return values[index];
    }
}

char* addBinary(char* a, char* b) {
    int len1 = strlen(a);
    int len2 = strlen(b);
    int carry = 0, i = len1 - 1, j = len2 - 1;
    int maxLen = len1 > len2 ? len1 : len2;
    char* result = (char*)malloc((maxLen + 1) * sizeof(char)); // 결과 문자열을 위한 공간을
    할당
    result[maxLen] = '\0';

    // 이진수를 오른쪽부터 왼쪽까지 더하는 것
    while (i >= 0 || j >= 0) {
        int sum = carry;
        if (i >= 0) sum += a[i--] - '0';
        if (j >= 0) sum += b[j--] - '0';

        carry = sum >> 1;    // 캐리는 합계가 2보다 크거나 같으면 1이고, 그렇지 않으면 0임
        result[--maxLen] = (sum & 1) + '0';    // 결과의 가장 오른쪽 비트를 합계의 최하위 비트
    로 설정
    }

    if (carry) {    // 만일 최종 캐리가 있다면 결과의 앞에 추가
        result = (char*)realloc(result, (strlen(result) + 2) * sizeof(char));    // 더 큰 결과
        문자열을 위해 메모리를 재할당
        memmove(result + 1, result, strlen(result) + 1);    // 결과 문자열을 1만큼 오른쪽으로
        이동
        result[0] = '1';    // 가장 왼쪽 비트를 1로 설정
    }
    return result;
}

// 입력된 명제 변수를 기반으로 빈 논리표를 2차원 배열로 할당하고 반환
int** create_initial_table(int num, char* letters) {

    for (int i = 0; i < num; i++) {
        insert(letters[i], i);
    }

    int total_row = 1 << num;
    int** arr = malloc(sizeof(int*) * total_row);

    for (int i = 0; i < total_row; i++) {
        arr[i] = malloc(sizeof(int) * (NUM_OF_OPERATOR + NUM_OF_OPERAND));
    }

    char* binary = (char*)malloc(sizeof(char) * num);

    memset(binary, '0', sizeof(char) * (num + 1));

```

```

        binary[num] = '\0';
        for (int i = 0; i < total_row; i++) {
            for (int j = 0; j < num; j++) {
                if (binary[j] == '1') {
                    arr[i][j] = 1;
                }
                else {
                    arr[i][j] = 0;
                }
            }
            binary = addBinary(binary, "1");
        }
        free(binary);
        return arr;
    }

void push_op(Operator* stack, char c) {
    if (stack->top + 1 == EXP_SIZE) {
        printf("Stack is full\n");
        exit(0);
    }
    stack->op[++(stack->top)] = c;
}

char pop_op(Operator* stack) {
    if (stack->top == -1) {
        printf("Stack is empty\n");
        exit(0);
    }
    return stack->op[(stack->top)--];
}

char peek_op(Operator* stack) {
    if (stack->top == -1) {
        //printf("Stack is empty\n");
        return '\0';
    }
    return stack->op[stack->top];
}

void push_val(Value* stack, char* c, int* value) {
    if (stack->top + 1 == EXP_SIZE) {
        printf("Stack is full\n");
        exit(0);
    }
    strcpy(stack->exp[++(stack->top)], c);
    for (int i = 0; i < (1 << NUM_OF_OPERAND); i++) {
        stack->value[(stack->top)][i] = value[i];
    }
}

Value* pop_val(Value* stack) {
    if (stack->top == -1) {
        printf("Stack is empty\n");
        return NULL;
    }
    (stack->top)--;
    return stack;
}

```

```

// 주 작업을 진행하는 메인 함수
void main_operation(char* expression) {

    // 우선순위표
    // ! - negation - 1
    // & - and - 2
    // > - implication - 3
    // | - or - 4
    // 0이 있는 곳에서 스택(열)에 있는 연산자의 우선순위가 높음
    // 1이 있는 곳에서 입력(행)에 있는 연산자의 우선순위가 높음

    int presedence[5][5] = {
        // ! & > |
        {1,1,1,1,1},
        {1,1,0,0,0},    // !
        {1,1,0,0,0},    // &
        {1,1,0,0,0},    // >
        {1,1,0,0,0}    // |
    };

    // 특정 연산자가 쉽게 접근하기 위한 해시 구조를 정의
    // 해시 구조에 따라 연산자 문자의 ANCSI 값으로 빠르고 쉽게 키값을 받음
    char operator_string[] = { '!', '&', '>', '|' };

    int operator_hash[128] = { 0 };
    operator_hash['!'] = 0;
    operator_hash['&'] = 1;
    operator_hash['>'] = 2;
    operator_hash['|'] = 3;

    int index = 0;
    int count = 0;

    // 명제 변수를 위한 문자열 정의
    char* letters = malloc(sizeof(char) * (NUM_OF_OPERAND * 2));
    int hash[26] = { 0 };    // 영어 알파벳 대문자의 총 개수인 26을 크기로 갖는 해시

    letters[count] = '\0';

    // 줄의 개수를 계산하면서 대입
    int total_row = (1 << NUM_OF_OPERAND);
    // 계산될 논리값들을 위한 배열 할당
    int* value_vector = malloc(sizeof(int) * total_row);

    // 반복문으로 특정 연산자에 해당되는 값을 매핑
    for (int i = 0; i < (sizeof(operator_string) / sizeof(operator_string[0])); i++) {
        insert(operator_string[i], i + 1);
    }

    // 명제를 읽고 명제 변수들만 letters에 대입
    while (expression[index] != '\0') {
        if (isalpha(expression[index]) && hash[expression[index] - 65] == 0) {
            letters[count] = expression[index];
            hash[expression[index] - 65] = 1;
            count++;
        }
        index++;
    }
}

```

```

// 결과를 위한 빈 논리표를 정의하고 result_table에 대입
result_table = create_initial_table(NUM_OF_OPERAND, letters);

// 연산자 스택과 비연산자 스택을 위한 구조체 변수 정의
Operator op_stack;
Value val_stack;
Value popped_value;
op_stack.top = -1;
val_stack.top = -1;
int result_index = NUM_OF_OPERAND; // 결과를 넣기 위한 테이블 인덱스 변수

// 스택에서 팝(pop)시킬 때 반환될 값을 저장하기 위한 배열 정의
int* popped_1 = malloc(sizeof(int) * (1 << NUM_OF_OPERAND));
int* popped_2 = malloc(sizeof(int) * (1 << NUM_OF_OPERAND));
char* exp_1 = malloc(sizeof(char) * EXP_SIZE);
char* exp_2 = malloc(sizeof(char) * EXP_SIZE);

// 명제 변수(들)을 결과 문자열에 대입
for (int i = 0; i < NUM_OF_OPERAND; i++) {
    expression_result[i][0] = letters[i];
    expression_result[i][1] = '\0';
}

index = 0;

// 명제를 처음부터 끝까지 읽어서 필요할 때 논릿값을 계산하면서 스택에 저장
while (index < strlen(expression)) {
    char current = expression[index];

    if (isalpha(current)) { // 해당 문자가 명제 변수일 경우
        for (int i = 0; i < total_row; i++) {
            value_vector[i] = result_table[i][getIndex(current) - 4]; // 4 = | { !, &,
>, | } |
        }
        char temp_string[2];
        temp_string[0] = current;
        temp_string[1] = '\0';
        push_val(&val_stack, temp_string, value_vector);
    }
    else { // 해당 문자가 논리 연산자일 경우
        if (current == '!') { // '!'일 경우
            current = expression[++index]; // '!' 뒤에 올 명제 변수를 읽음

            for (int i = 0; i < total_row; i++) {
                value_vector[i] = result_table[i][getIndex(current) - 4]; // 4 = | { !,
&, >, | } |

                value_vector[i] = (value_vector[i] == 1) ? 0 : 1; // 논릿값을 부정
                result_table[i][result_index] = value_vector[i];
            }

            char temp_string[3] = "!";
            temp_string[1] = current;
            temp_string[2] = '\0';
            strcpy(expression_result[result_index++], temp_string);
            push_val(&val_stack, temp_string, value_vector);
        }
        // 스택에 있는 연산자의 우선순위가 높을 때
        else if (presedence[getIndex(peek_op(&op_stack)) + 1][getIndex(current) + 1] ==

```

```

0) {

    char temp_from_stack[2];
    temp_from_stack[0] = pop_op(&op_stack);
    temp_from_stack[1] = '\0';

    popped_value = *pop_val(&val_stack);
    strcpy(exp_2, popped_value.exp[val_stack.top + 1]);
    for (int i = 0; i < total_row; i++) {
        popped_2[i] = popped_value.value[val_stack.top + 1][i];
    }

    exp_2[(strlen(exp_2))] = '\0';
    popped_value = *pop_val(&val_stack);
    strcpy(exp_1, popped_value.exp[val_stack.top + 1]);
    for (int i = 0; i < total_row; i++) {
        popped_1[i] = popped_value.value[val_stack.top + 1][i];
    }
    strcat(exp_1, temp_from_stack);
    exp_1[(strlen(exp_1))] = '\0';
    strcat(exp_1, exp_2);
    exp_1[(strlen(exp_1))] = '\0';

    switch (temp_from_stack[0])
    {
    case '&':
        for (int i = 0; i < total_row; i++) {
            value_vector[i] = popped_1[i] & popped_2[i];
        }
        break;
    case '>':
        for (int i = 0; i < total_row; i++) {
            if (popped_1[i] == 1 && popped_2[i] == 0)
                value_vector[i] = 0;
            else
                value_vector[i] = 1;
        }
        break;
    case '|':
        for (int i = 0; i < total_row; i++) {
            value_vector[i] = popped_1[i] | popped_2[i];
        }
        break;
    default:
        break;
    }

    for (int i = 0; i < total_row; i++) {
        result_table[i][result_index] = value_vector[i];
    }
    strcpy(expression_result[result_index++], exp_1);

    push_val(&val_stack, exp_1, value_vector);

    push_op(&op_stack, current);

}
// 입력에 있는 연산자의 우선순위가 높을 때
else if (presedence[getIndex(peek_op(&op_stack)) + 1][getIndex(current) + 1] ==

```

```

1) {
    push_op(&op_stack, current);
}
}

index++;
}

//스택에 남아 있는 것을 팝(pop)시키면서 논리 연산을 진행
while (op_stack.top != -1) {
    char temp_from_stack[2];
    temp_from_stack[0] = pop_op(&op_stack);
    temp_from_stack[1] = '\0';

    popped_value = *pop_val(&val_stack);
    strcpy(exp_2, popped_value.exp[val_stack.top + 1]);
    for (int i = 0; i < total_row; i++) {
        popped_2[i] = popped_value.value[val_stack.top + 1][i];
    }

    exp_2[(strlen(exp_2))] = '\0';
    popped_value = *pop_val(&val_stack);
    strcpy(exp_1, popped_value.exp[val_stack.top + 1]);
    for (int i = 0; i < total_row; i++) {
        popped_1[i] = popped_value.value[val_stack.top + 1][i];
    }
    strcat(exp_1, temp_from_stack);
    exp_1[(strlen(exp_1))] = '\0';
    strcat(exp_1, exp_2);
    exp_1[(strlen(exp_1))] = '\0';

    switch (temp_from_stack[0])
    {
    case '&':
        for (int i = 0; i < total_row; i++) {
            value_vector[i] = popped_1[i] & popped_2[i];
        }
        break;
    case '>':
        for (int i = 0; i < total_row; i++) {
            if (popped_1[i] == 1 && popped_2[i] == 0)
                value_vector[i] = 0;
            else
                value_vector[i] = 1;
        }
        break;
    case '|':
        for (int i = 0; i < total_row; i++) {
            value_vector[i] = popped_1[i] | popped_2[i];
        }
        break;
    default:
        break;
    }

    for (int i = 0; i < total_row; i++) {
        result_table[i][result_index] = value_vector[i];
    }
    strcpy(expression_result[result_index++], exp_1);
}

```



```

    push_val(&val_stack, exp_1, value_vector);
}

// 할당된 공간들을 해제
free(value_vector);
free(popped_1);
free(popped_2);
free(exp_1);
free(exp_2);
free(letters);
}

// 계산된 논리표를 화면에 출력하면서 output.txt라는 파일에 저장
void display_and_writeOnFile() {
    FILE* outputFile;
    int* spaces = malloc(sizeof(int) * (NUM_OF_OPERAND + NUM_OF_OPERATOR)); // 공백의 개수를
    위한 배열
    int total_row = (1 << NUM_OF_OPERAND); // 총 줄의 개수

    // output.txt 파일을 새로 만들
    outputFile = fopen("output.txt", "w");
    if (outputFile == NULL) {
        printf("File open error\n");
        exit(0);
    }

    // expression_result의 논리표를 화면에 출력하면서 파일에 저장
    // 논리표를 예쁘게 출력하기 위해 space 배열을 통해 공백도 잘 출력
    printf("\n");
    _setmode(_fileno(stdout), _O_U16TEXT); // _O_TEXT

    wprintf(L"\x250c");
    fwprintf(outputFile, L"\x250c", "string");

    for (int j = 0; j < (NUM_OF_OPERAND + NUM_OF_OPERATOR); j++) {
        spaces[j] = strlen(expression_result[j]) + 2;
        for (int i = 0; i < spaces[j]; i++) {
            wprintf(L"\x2500");
            fwprintf(outputFile, L"\x2500", "string");
        }
        if (j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1) {
            wprintf(L"\x252c");
            fwprintf(outputFile, L"\x252c", "string");
        }
    }

    wprintf(L"\x2510\n\x2502");
    fwprintf(outputFile, L"\x2510\n\x2502", "string");
    for (int j = 0; j < (NUM_OF_OPERAND + NUM_OF_OPERATOR); j++) {
        _setmode(_fileno(stdout), _O_TEXT); // _O_TEXT
        printf(" %s ", expression_result[j]);
        fprintf(outputFile, " %s ", expression_result[j]);
        _setmode(_fileno(stdout), _O_U16TEXT);
        wprintf(L"\x2502");
        fwprintf(outputFile, L"\x2502", "string");
    }

    wprintf(L"\n\x251c");
    fwprintf(outputFile, L"\n\x251c", "string");
}

```

```

for (int j = 0; j < (NUM_OF_OPERAND + NUM_OF_OPERATOR); j++) {
    for (int i = 0; i < spaces[j]; i++) {
        wprintf(L"\x2500");
        fwprintf(outputFile, L"\x2500", "string");
    }
    if (j != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1) {
        wprintf(L"\x253c");
        fwprintf(outputFile, L"\x253c", "string");
    }
}
wprintf(L"\x2524\n");
fwprintf(outputFile, L"\x2524", "string");

for (int i = 0; i < total_row; i++) {
    wprintf(L"\x2502");
    fwprintf(outputFile, L"\x2502", "string");
    for (int j = 0; j < (NUM_OF_OPERAND + NUM_OF_OPERATOR); j++) {
        if (spaces[j] % 2 == 1) {
            for (int i = 0; i < spaces[j] / 2; i++) {
                wprintf(L" ");
                fwprintf(outputFile, L" ", "string");
            }
            if (result_table[i][j] == 1) {
                wprintf(L"T");
                fwprintf(outputFile, L"T", "string");
            }
            else if (result_table[i][j] == 0) {
                wprintf(L"F");
                fwprintf(outputFile, L"F", "string");
            }
            for (int i = 0; i < spaces[j] / 2; i++) {
                wprintf(L" ");
                fwprintf(outputFile, L" ", "string");
            }
        }
        else {
            for (int i = 0; i < (spaces[j] / 2) - 1; i++) {
                wprintf(L" ");
                fwprintf(outputFile, L" ", "string");
            }
            if (result_table[i][j] == 1) {
                wprintf(L"T");
                fwprintf(outputFile, L"T", "string");
            }
            else if (result_table[i][j] == 0) {
                wprintf(L"F");
                fwprintf(outputFile, L"F", "string");
            }
            for (int i = 0; i < spaces[j] / 2; i++) {
                wprintf(L" ");
                fwprintf(outputFile, L" ", "string");
            }
        }
        wprintf(L"\x2502");
        fwprintf(outputFile, L"\x2502", "string");
    }

    if (i != total_row - 1) {
        wprintf(L"\n\x251c");
    }
}

```

```

        fprintf(outputFile, L"\n\x251c", "string");
        for (int k = 0; k < (NUM_OF_OPERAND + NUM_OF_OPERATOR); k++) {
            for (int m = 0; m < spaces[k]; m++) {
                wprintf(L"\x2500");
                fprintf(outputFile, L"\x2500", "string");
            }
            if (k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1) {
                wprintf(L"\x253c");
                fprintf(outputFile, L"\x253c", "string");
            }
        }
        wprintf(L"\x2524\n");
        fprintf(outputFile, L"\x2524\n", "string");
    }

}

wprintf(L"\n\x2514");
fprintf(outputFile, L"\n\x2514", "string");
for (int k = 0; k < (NUM_OF_OPERAND + NUM_OF_OPERATOR); k++) {
    for (int m = 0; m < spaces[k]; m++) {
        wprintf(L"\x2500");
        fprintf(outputFile, L"\x2500", "string");
    }
    if (k != (NUM_OF_OPERAND + NUM_OF_OPERATOR) - 1) {
        wprintf(L"\x2534");
        fprintf(outputFile, L"\x2534", "string");
    }
}

wprintf(L"\x2518\n");
fprintf(outputFile, L"\x2518", "string");

// 할당된 공간을 해제하고 파일을 닫음
free(spaces);
fclose(outputFile);
}

int main() {
    char expression[EXP_SIZE];

    // 사용자로부터 명제 변수의 개수와 명제를 입력받음
    printf("명제 변수의 개수를 입력하라.: ");
    scanf("%d", &NUM_OF_OPERAND);
    int total_row = (1 << NUM_OF_OPERAND);
    printf("명제를 입력하라.: ");
    char c;
    scanf("%c", &c);

    fgets(expression, EXP_SIZE, stdin);
    expression[strlen(expression) - 1] = '\0';
    // 명제의 형식이 맞는지 확인
    if (check_and_despace(expression, NUM_OF_OPERAND)) {
        printf("%s", expression);
    }
    else {
        printf("wrong expression!\a\n");
        return 0;
    }

    // 결과를 위해 충분한 공간을 할당

```

```

expression_result = malloc(sizeof(char*) * (NUM_OF_OPERAND + NUM_OF_OPERATOR));
for (int i = 0; i < (NUM_OF_OPERAND + NUM_OF_OPERATOR); i++) {
    expression_result[i] = malloc(sizeof(char) * EXP_SIZE);
}

main_operation(expression);    // 메인 함수 호출
display_and_writeOnFile();    // 결과표를 콘솔창에 출력하면서 텍스트 파일에 저장하는 함수

// 할당된 메모리 공간을 해제
for (int i = 0; i < (NUM_OF_OPERAND + NUM_OF_OPERATOR); i++) {
    free(expression_result[i]);
}
free(expression_result);

for (int i = 0; i < total_row; i++) {
    free(result_table[i]);
}
free(result_table);
return 0;
}

// 테스트하기 위한 예제 명제들(테스트할 때 명제 변수의 개수에 주의!):
// P & Q > !P | Q = 0
// P > !Q & R | !P = x
// P & Q > !P | Q = 0
// P > !Q & P | !P > Q | P = 0
// P > !Q & P | !P > Q | P & R > S = 0
// P & Q & S & T & R = 0
// P | Q & !R > P = 0
// P&Q>S!T!R&Q&T!U

```