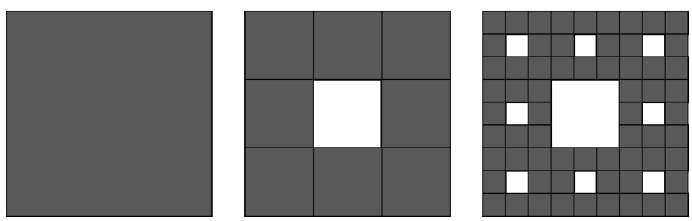
프로젝트 01

시에르핀스키 사각형(1권)

✓핵심 키워드

시에르핀스키 사각형, 재귀 함수, 분할 정복

여기서는 무얼 배울까



1권 챕터9 '함수'의 연습 문제에서 재귀적으로 정의된 프랙탈 도형인 시에르핀스키 사각형에 대해 알아보았다. 시에르핀스키 사각형처럼 재귀적으로 정의된 문제는 흔히 볼 수 있고, 이러한 문제를 푸는 가장 좋은 방법은 분할 정복의 아이디어이다. 이번 프로젝트에서는 시에르핀스키 사각형의 규칙을 따르는 별 찍기 프로그램을 만들어 보고, 분할 정복을 익혀 보자.

문제 파악하기

어떤 문제를 풀 때 가장 중요한 것은 그 문제를 제대로 파악하는 것이다. 해결하고자 하는 문제가 쉬운 문제일지라도 문제가 요구하는 것은 무엇인지, 문제를 풀기 위해서 무엇이 필요한지 정리하는 습관은 늘 도움이 된다.

문제 정의

입력

0 이상의 정수 N을 입력받는다.

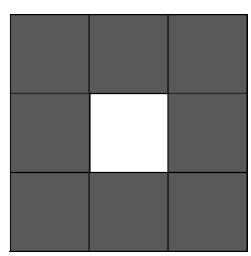
출력

N단계 시에르핀스키 사각형을 출력한다.
사각형의 한 변의 길이는 3^N이다.

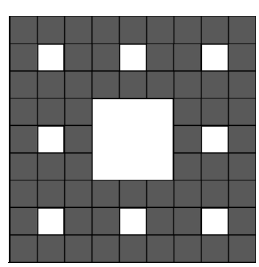
이번 프로젝트에서 만들 프로그램은 다음과 같이 간단한 입력과 출력으로 정리할 수 있다. 다음으로 필요한 단계는 출력에서 필요로 하는 시에르핀스키 사각형의 규칙을 찾는 것이다. 시에르핀스키 사각형의 규칙은 앞선 연습 문제에서 설명했지만, 여기서는 한 단계씩 진행하며 규칙의 흐름을 파악할 것이다.

문제 규칙

0단계 시에르핀스키 사각형은 완전히 칠해진 정사각형이다. 문제에서 출력하고자 하는 0단계 시에르핀스키는 1*1 크기이므로, 실제 출력의 결과는 별 하나가 될 것이다.



1단계 시에르핀스키 사각형은 가운데가 비어 있는 정사각형이다. 1단계는 단순히 가운데가 비어 있는 정사각형으로 생각할 수 있지만, 실제로는 0단계 시에르핀스키 8개가 뭉친 것이다. 1단계의 크기가 3*3이므로, 실제 출력의 결과도 8개의 0단계인 총 8개의 별로 출력됨을 예상할 수 있다.

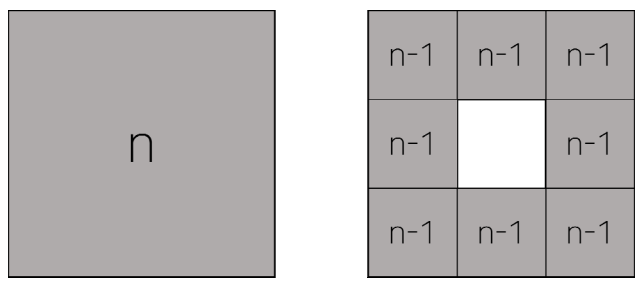


2단계 시에르핀스키 사각형은 2단계가 8개 뭉친 모습이며, 크기가 9*9이므로 앞서 본 1단계 시에르핀스키 사각형 8개가 합친 크기와 동일하다. n단계 시에르핀스키 사각형의 크기가 3^n * 3^n으로 정해져 있으므로, 우리는 여기서 n단계 시에르핀스키 사각형의 크기와 n+1단계를 이루는 n단계 시에르핀스키 사각형의 크기가 완전히 동일함을 알 수 있다.

분할 정복

n단계 시에르핀스키 사각형은 8개의 n−1단계 시에르핀스키 사각형으로 구성된다. 그리고 n단계를 구성하는 n−1단계와, 독립적인 n−1단계는 그 크기가 같다. 우리는 이 사실을 바탕으로 n단계 시에르핀스키 사각형을 출력하는 아이디어를 구상하고자 한다.

부분 문제로 나누기



최종적으로 출력하고자 하는 시에르핀스키 사각형은 n단계이지만, 우리는 n단계를 n−1단계로 나눌 수 있음을 안다. 만약 곧바로 100단계를 출력하고자 한다면 매우 복잡한 과정일 수 있지만, 시에르핀스키 사각형의 규칙을 통해 100단계를 출력하는 것이 아닌 99단계를 8개 출력하는 것으로 전체 문제를 더 작게 쪼갤 수 있다.

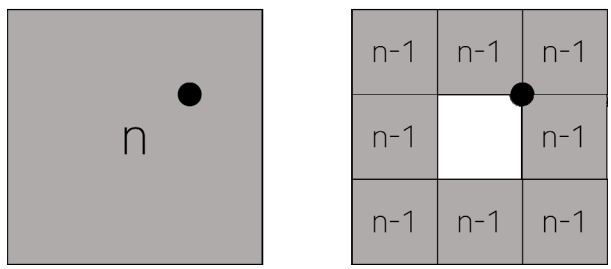
어떤 큰 문제를 보다 작은 문제 여럿으로 나누는 방법을 분할 정복이라고 한다. 대다수의 큰 문제는 한 번에 풀려고 시도하면 어렵지만, 더 작은 문제로 나누어 풀면 더 쉽게 풀 수 있다. 시에르핀스키 사각형도 역시 마찬가지이다. 100단계보다는 99단계를 출력하기가 더 쉽고, 99단계보다는 98단계를 출력하는 것이 더 쉽다. 분할 정복의 아이디어는 재귀적인 관계에서 더 빛을 발하는데, n단계를 n−1단계로 나눌 수 있다는 규칙만으로 우리는 100단계를 0단계까지 한 번에 쪼갤 수 있다는 것이다. 잘게 쪼개진 문제가 많은 것을 걱정할 필요는 없다. 문제를 쪼개는 관계식을 잘 정의하면 나머지는 재귀 함수의 마법으로 해결되기 때문이다.

```
draw(n) = draw(n-1)을 8개 그린다.
draw(0) = 별 하나를 그린다.
```

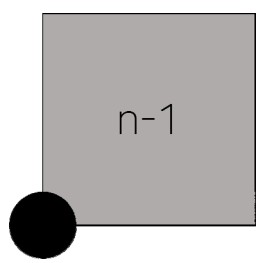
우리는 결과적으로 다음과 같은 알고리즘을 생각할 수 있다. n의 크기와는 관계없이 항상 이 방법으로 문제를 해결할 수 있다. 이것이 분할 정복과 재귀 함수의 힘이다.

한 칸씩 생각하기

원하는 위치에 별을 그리는 방법이 있다면 위에서 찾은 방법으로 문제를 해결할 수 있지만 공교롭게도 우리는 아직 그러한 방법을 배우지 않았다. 따라서 한 단계를 더 거쳐서 문제를 해결해 보자.



큰 사각형을 한 번에 출력할 수 없기에, 이제 별을 하나씩 출력하는 과정을 생각해 보자. 만약 지금 출력해야 할 위치가 n단계의 검은 원의 위치라고 한다면, n−1단계 8개로 나눴을 때에도 똑같이 검은 원의 위치이다. 단계를 하나 낮췄으므로 우리는 n단계를 생각하지 않고 대신 n−1단계를 생각할 수 있다.



출력할 위치의 n−1단계를 확대하면 이런 형태이다. 특정 위치의 별을 출력하기 위해서 우리가 알아야 하는 것은 그 위치에 출력할 것이 별이냐 빈칸이냐이다. 그러므로 n단계의 특정 위치가 별인지 아닌지를 알아내는 것은, 그것에 대응되는 n−1단계의 위치에서 별인지 아닌지 알아내는 것과 동일하다. 이 규칙은 모든 위치의 점에서 똑같이 적용되고, 모든 단계는 순서대로 내려가 최종적으로 0단계에 도달한다. 0단계에 도달하기 전에 빈 공간이라는 사실을 밝혀내면 그 위치는 빈 공간이고, 0단계에 도달한다면 그 위치는 별이었던 것이다.

```
isStar(x, y, n) = 별이 아니다. { (x, y)가 n단계의 중심에 위치할 때 }
isStar(x, y, n) = isStar(x', y', n-1) { 그렇지 않을 때 }
isStar(x, y, n) = 별이다. { n = 0 }
```

위 아이디어를 바탕으로 (x, y) 위치가 별인지 아닌지를 알아내는 함수를 정의했다. 이제 남은 것은 (x, y)가 n단계의 중심인지를 확인하는 것과 x', y'을 계산하는 것이다.

수식 정리

지금부터 사용할 좌표는 왼쪽 위가 (0, 0)이고, 가장 끝은 $(3^{(n-1)}, 3^{(n-1)})$를 가진다. n단계의 시에르핀스키 사각형은 가로와 세로를 정확히 3등분하여 그중 가운데는 모두 빈칸, 나머지는 n−1단계로 넘긴다. 그러므로 가운데의 정사각형은 3^n의 중간 지점에 해당한다.

```
k = 3^(n-1)
isStar(x, y, n) = 별이 아니다. { k <= x, y < 2k }
```

별이 아닌 경우에 대한 조건을 찾았으니 다음은 n−1단계로 내려간 후의 (x', y')이다. 이에 대한 규칙은 생각보다 간단한데, 3^n개의 좌표에서 $3^{(n-1)}$개 3개가 같은 규칙으로 반복하고 있으므로 나머지 연산을 이용해 준다.

```
k = 3^(n-1)
isStar(x, y, n) = 별이 아니다. { k <= x, y < 2k }
isStar(x, y, n) = isStar(x % k, y % k, n-1) { 그렇지 않을 때 }
isStar(x, y, n) = 별이다. { n = 0 }
```

2단계로 예를 들자면, 2단계를 구성하는 x 좌표는 [0, 1, 2, 3, 4, 5, 6, 7, 8]이다. 이를 3개로 나누면 각각에 대한 x 좌표는 [0, 1, 2], [0, 1, 2], [0, 1, 2]이고, 이는 2단계를 구성하는 좌표에서 3으로 나눈 나머지를 구한 것과 같다.

구현하기

시에르핀스키 사각형을 출력하기 위한 모든 규칙과 패턴을 찾았다. 이제 남은 것은 이를 코드로 구현하여 실제 결과를 보는 것이다.

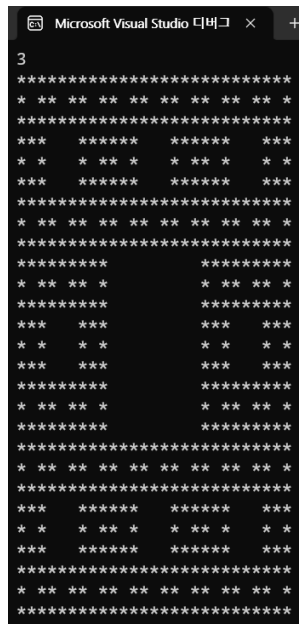
```
int isStar(int x, int y, int n)
{
    if (n == 0)
        return 1;

    int k = 1;
    for (int i = 0; i < n - 1; i++)
        k *= 3;
```

```
    if (k <= x && x < k * 2
        && k <= y && y < k * 2)
        return 0;

    return isStar(x % k, y % k, n - 1);
}
```

n단계의 특정 위치가 별인지 아닌지를 계산하는 재귀 함수이다. 위에서 구한 재귀 호출과 종료 조건, 그리고 수식 계산을 그대로 함수로 만든 것이다.

```
int main()
{
    int n, n3 = 1;
    scanf("%d", &n);

    for (int i = 0; i < n; i++)
        n3 *= 3;

    for (int x = 0; x < n3; x++)
    {
        for (int y = 0; y < n3; y++)
        {
        if (isStar(x, y, n))
            printf("*");
        else
            printf(" ");
        }
        printf("\n");
    }
}
```

main 함수에서는 출력할 시에르핀스키 사각형의 단계인 n을 입력받고서 isStar()를 이용해 사각형을 출력한다. n단계는 크기가 3^n이므로 n3=3^n을 먼저 구하고, 각 좌표에 대해 isStar(x, y, n)을 호출하여 별과 빈칸을 출력한다.



완성된 프로그램으로 3단계 시에르핀스키 사각형을 출력한 결과이다. 큰 문제를 보다 작은 문제로 나누어 문제를 푸는 전략인 분할 정복은 앞으로 만날 많은 문제를 해결할 수 있는 강력한 무기이고, 특히 재귀 함수와 만났을 때 그 효과는 더 커진다. 앞으로 만날 문제에 대해서 항상 이러한 방법으로 풀 수 있는 것은 아니고, 또 풀 수 있다 해도 그 풀이를 바로 찾을 수는 없겠지만, 그럼에도 큰 문제를 만났을 때 작은 문제로 나누어 생각하는 것은 문제의 본질을 찾고 답을 향해 나아가는 데 도움이 될 것임은 분명하다.

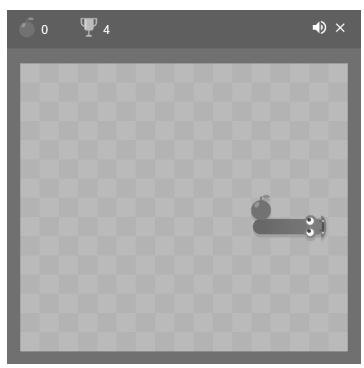
02 스네이크 게임(2권)

✓핵심 키워드

스네이크 게임, 콘솔, time.h, stdlib.h, 랜덤, 시간

리눅스용 설명(PDF 제공)

여기서는 무얼 배울까



스네이크 게임은 매우 유명한 콘솔 게임 중 하나이다. 무작위로 생성되는 사과를 먹으면 뱀이 길어지고, 벽이나 자신의 몸에 닿으면 게임에서 패배한다. 게임의 목표는 뱀의 길이를 최대한 길게 만드는 것이다. 이번 프로젝트에선 2권 챕터14 '콘솔 프로그래밍'에서 배운 콘솔 제어와 C 언어의 표준 함수들을 이용하여 스네이크 게임을 직접 구현해 보자.

문제 파악하기

스네이크 게임의 규칙은 간단하지만 이를 구현하기 위한 요소들을 파악해 보면 생각보다 그 수가 많다는 것을 느낄 수 있을 것이다.

요소 분할

1. 뱀
2. 사과
3. 벽

게임을 이루는 요소는 뱀과 사과, 그리고 벽이 있다. 이 세 요소는 콘솔에서 보이면서 게임에 영향을 준다. 그렇기에 스네이크 게임은 이 요소들을 중심으로 구성되어야 한다.

```
이동
길이 증가
패배
사과 랜덤 생성
```

세 요소로부터 얻을 수 있는 핵심적인 기능은 이동과 길이 증가, 사과 랜덤 생성, 그리고 패배가 있다. 열거된 네 기능이 가장 중요한 기능이므로 앞으로의 설계는 네 기능을 중심으로 하되, 필요에 따라 함수를 더 나누어 구현한다.

함수 분할

요소에 대한 기능은 곧 하나의 함수가 된다. 위에서 정리한 기능을 바탕으로 기본적인 함수를 나열하면 다음과 같다.

```
이동()
길이_증가()
패배()
사과_생성()
```

```
출력()
입력()
```

벽과 과일, 뱀을 출력하는 기본적인 함수를 구상하고, 여기에 추가로 현재 뱀의 길이를 숫자로 확인할 수 있도록 점수 출력 함수를 추가한다. 입력 함수는 키보드로부터 방향키 입력을 받고 가장 최근에 입력한 방향을 저장하는 용도로 사용될 것이다.

설계하기

게임을 구성하는 기본적인 함수를 구상했으니, 이제 이 함수를 연결하여 프로그램의 전체적인 틀을 만들 차례이다.

게임 루프

```
게임이 끝날 때까지 반복한다
    입력
    업데이트
    렌더
```

일반적인 게임 루프는 입력과 업데이트, 렌더를 반복한다. 뼈대는 이 게임 루프를 바탕으로 하겠으나, 우리는 여기서 효과적인 처리를 위해 약간의 변화를 추가할 것이다.

```
게임이 끝날 때까지 반복한다
    입력
    업데이트 시간이 되었다면?
        업데이트
        렌더
```

스네이크 게임은 컴퓨터의 연산량을 100% 사용할 필요가 없다. 뱀이 1초에 한 칸씩 움직인다면 업데이트와 렌더를 1초에 한 번만 수행하면 되기 때문이다. 이를 위해 이전 업데이트에서부터 지금까지 시간이 얼마나 흘렀는지를 계산하고, 뱀의 이동 속도에 맞춰서 업데이트의 횟수를 제한한다.

함수 설계

상당수의 함수는 간단한 기능을 하지만 일부 함수는 다른 함수와 연계하여 연산이 이루어지는 경우가 있다.

```
이동
이동할 위치를 계산한다.
이동할 위치에 몸 또는 벽이 있다면?
    패배한다.
이동할 위치에 사과가 있다면?
    사과를 먹는다.
이동한다.
```

뱀의 이동은 패배하거나 사과를 먹는 함수와 연계되어 있다. 뱀이 이동할 위치에 무엇이 있느냐에 따라 각 연산으로 분기하기 때문이다.

```
사과 먹기
길이를 증가시킨다.
사과를 생성한다.
```

사과를 먹으면 항상 길이가 증가하고 새로운 사과가 생성된다. 여기서 사과 먹기와 길이 증가는 항상 붙어 있으므로 이 둘을 묶어 하나의 함수로 만든다.

```
사과 생성
반복한다:
    랜덤한 위치에 사과를 생성한다.
    그 위치에 뱀의 몸이 존재하지 않는다면?
        종료한다.
```

사과가 뱀의 몸에 생성되면 안 되므로 뱀의 몸이 아닌 위치에 생성될 때까지 반복한다. 운이 나쁘다면 하나의 사과를 생성하기 위해서 많은 반복이 있을 수 있지만, 확률적으로 성능에 문제가 있을 만큼 반복이 될 가능성은 없고, 이 방법이 구현이 더 간단하다.

구현하기

만들어야 할 함수들을 나열하고 그 함수들 사이의 관계를 정했으면, 이제 본격적으로 프로그램을 만들 차례이다. 여기서는 프로그램을 5개의 소스 파일과 4개의 헤더 파일로 나누어서 만들었으며 각 파일 사이의 값을 전달해야 하는 것은 전역 변수로 만들었다. 위에서 설명한 함수 외에 편의를 위한 함수도 포함되어 있다.

winUtil

winUtil.h

```c
#include <Windows.h>
#include <conio.h>

typedef enum color {
    BLACK, DARK_BLUE, DARK_GREEN, DARK_SKY_BLUE,
    DARK_RED, DARK_PURPLE, DARK_YELLOW, GRAY,
    DARK_GRAY, BLUE, GREEN, SKY_BlUE,
    RED, PURPLE, YELLOW, WHITE
```

```
} color;

void setColor(color background, color text); // 콘솔의 색을 제어
void gotoxy(int x, int y); // 커서의 위치를 제어
void setCursor(int size, BOOL visible); // 커서 설정
```

winUtil에는 Windows.h의 함수를 사용하기 쉽도록 만든 함수들이 포함된다. 여기에는 색 설정, 커서 이동, 커서 설정이 있다.

winUtil.c

```
#include "winUtil.h"

void setColor(color background, color text)
{
    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), background * 16 + text);
}

void gotoxy(int x, int y)
{
    COORD pos = { x, y };
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), pos);
}

void setCursor(int size, BOOL visible)
{
    CONSOLE_CURSOR_INFO cursorInfo;
    cursorInfo.dwSize = size;
    cursorInfo.bVisible = visible;
    SetConsoleCursorInfo(GetStdHandle(STD_OUTPUT_HANDLE), &cursorInfo);
}
```

함수의 구현은 2권 챕터14 '콘솔 프로그래밍'에서 본 것과 동일하다.

input

input.h

```
typedef enum dir {
    RIGHT, LEFT, UP, DOWN
} dir;

void input(); // 입력 처리 함수
```

input은 키보드 입력을 담당한다.

input.c

```c
#include <conio.h>
#include "input.h"

// 뱀이 이동할 방향을 나타내는 전역 변수
dir SNAKE_DIR = RIGHT;

void input()
{
    if (_kbhit())
    {
        int code = _getch();
        if (code == 0xE0)
        {
            int code2 = _getch();
            if (code2 == 72) // 위쪽 방향키
                SNAKE_DIR = UP;
            else if (code2 == 80) // 아래쪽 방향키
                SNAKE_DIR = DOWN;
            else if (code2 == 75) // 왼쪽 방향키
                SNAKE_DIR = LEFT;
            else if (code2 == 77) // 오른쪽 방향키
                SNAKE_DIR = RIGHT;
        }
    }
}
```

input()은 kbhit()과 getch()를 이용하여 방향키 입력을 받는다. 스네이크 게임에서 한 번 입력한 방향은 다른 방향을 누르기 전까지 지속되므로, 이전의 입력을 저장하는 변수 SNAKE_DIR이 있다.

render

render.h

```c
void initRender(); // 시작 렌더 함수
void renderMove(); // 이동 렌더 함수
void renderEat(); // 사과를 먹었을 때의 렌더 함수
```

render는 콘솔에 대한 렌더 함수들이 모여 있다. 렌더는 3종류로 시작했을 때, 이동했을 때, 그리고 사과를 먹었을 때이다.

render.c ... 1

```c
#include <stdio.h>
#include "render.h"
#include "winUtil.h"

extern const int MAP_X;
extern const int MAP_Y;
extern const int START_X;
extern const int START_Y;
extern int SNAKE_LEN;
extern int APPLE_X;
extern int APPLE_Y;
extern int SNAKE_POS[128][2];
extern int PRE_POS[2];

void initRender()
{
    // 커서 제거
    setCursor(1, FALSE);

    // 벽 렌더
    setColor(WHITE, WHITE);
    for (int i = 0; i < MAP_X + 2; i++)
    {
        gotoxy(i * 2, 0);
        printf("  ");

        gotoxy(i * 2, MAP_Y + 1);
        printf("  ");
    }
    for (int i = 0; i < MAP_Y + 2; i++)
    {
        gotoxy(0, i);
        printf("  ");

        gotoxy((MAP_X + 1) * 2, i);
        printf("  ");
    }
```

```
    // 뱀 렌더
    setColor(BLUE, BLUE);
    for (int i = 0; i < SNAKE_LEN; i++)
    {
        gotoxy(2 * (START_X + i), START_Y);
        printf("  ");
    }

    // 사과 렌더
    setColor(RED, RED);
    gotoxy(APPLE_X * 2, APPLE_Y);
    printf("  ");

    // 점수 렌더
    setColor(BLACK, WHITE);
    gotoxy(2, MAP_Y + 3);
    printf("score:%d", SNAKE_LEN);
}
```

render는 다른 파일의 여러 값들을 바탕으로 렌더링한다. initRender()에선 게임이 시작되고 나서 필요한 벽, 사과, 뱀, 점수를 그려 내는 역할을 수행한다.

render.c ... 2

```
void renderMove()
{
    // 꼬리 제거
    setColor(BLACK, BLACK);
    gotoxy(PRE_POS[0] * 2, PRE_POS[1]);
    printf("  ");

    // 머리 추가
    setColor(BLUE, BLUE);
    gotoxy(SNAKE_POS[SNAKE_LEN - 1][0] * 2, SNAKE_POS[SNAKE_LEN - 1][1]);
    printf("  ");
}
```

renderMove()는 사과를 먹지 않고 이동했을 때를 렌더링하는 함수이다. 사과를 먹지 않았다면 뱀의 길이는 변하지 않고, 뱀의 전체 모습을 보면 꼬리가 한 칸 줄고 머리가 한 칸 늘어난다. 변하지 않는 다른 모든 몸을 다시 그리는 것은 비효율적이므로 여기서는 꼬리를 제거하고 머리를 추가하는 두 부분만 다시 그린다.

render.c ... 3

```
void renderEat()
{
    // 머리 추가
    setColor(BLUE, BLUE);
    gotoxy(SNAKE_POS[SNAKE_LEN - 1][0] * 2, SNAKE_POS[SNAKE_LEN - 1][1]);
    printf("  ");

    // 사과 렌더
    setColor(RED, RED);
    gotoxy(APPLE_X * 2, APPLE_Y);
    printf("  ");

    // 점수 렌더
    setColor(BLACK, WHITE);
    gotoxy(2, MAP_Y + 3);
    printf("score:%d", SNAKE_LEN);
}
```

반대로 사과를 먹었다면 길이가 하나 증가하여 꼬리가 없어지지 않는다. 따라서 이 경우 꼬리를 제거하지 않고 머리를 추가한 후 사과의 위치와 점수를 업데이트하도록 한다.

snake

snake.h

```
typedef enum moveResult {
   EAT, MOVE, FAIL
} moveResult;

void init(); // 뱀 시작 설정 함수
moveResult move(); // 이동 함수
void newApple(); // 사과 생성 함수
void eat(int x, int y); // 사과 섭취 함수
```

스네이크 게임의 핵심 기능인 snake이다. 앞선 설계에서 설명한 함수가 그대로 있는데, 한 가지 특징으로 렌더링 자체가 사과를 먹는 경우와 먹지 않는 경우로 나뉘므로 이를 알아낼 수 있도록 이동의 결과를 반환하도록 하였다.

snake.c ... 1

```
#include <stdlib.h>
#include <time.h>
#include "snake.h"
#include "input.h"

const int START_X = 3; // 뱀의 시작 위치 x
const int START_Y = 8; // 뱀의 시작 위치 y
int SNAKE_LEN = 3; // 뱀의 길이
int SNAKE_POS[128][2];
int PRE_POS[2];

int APPLE_X = 12; // 사과의 위치 x
int APPLE_Y = 8; // 사과의 위치 y

extern dir SNAKE_DIR;
extern const int MAP_X;
extern const int MAP_Y;

void init()
{
    srand(time(NULL)); // 랜덤 함수의 시드값을 설정
    for (int i = 0; i < SNAKE_LEN; i++)
    {
        // 뱀 몸 위치 설정
        SNAKE_POS[i][0] = START_X + i;
        SNAKE_POS[i][1] = START_Y;
    }
}
```

snake.c에는 뱀의 길이와 몸의 위치, 사과의 위치를 나타내는 전역 변수가 있다. init()은 뱀의 길이와 시작 위치에 맞게 몸 위치 배열을 초기화하는 역할을 한다. init()의 상단에 있는 srand()는 stdlib.h에 있는 함수로, 무작위 값을 뽑아내기 위한 시드를 설정한다. 시드가 같다면 랜덤 함수의 결과가 똑같이 나오기 때문에 시드 값을 현재 시간으로 설정하여 매 게임마다 다른 위치에 사과가 생성되도록 유도한다.

snake.c ... 2

```
moveResult move()
{
    int headX = SNAKE_POS[SNAKE_LEN - 1][0], headY = SNAKE_POS[SNAKE_LEN -
        1][1];
```

```
    if (SNAKE_DIR == RIGHT)
        headX++;
    else if (SNAKE_DIR == LEFT)
        headX--;
    else if (SNAKE_DIR == DOWN)
        headY++;
    else if (SNAKE_DIR == UP)
        headY--;

    // 머리가 벽에 닿았다면 패배
    if (headX == 0 || headY == 0 || headX == MAP_X + 1 || headY == MAP_Y + 1)
        return FAIL;

    // 머리가 몸에 닿았다면 패배
      for (int i = 1; i < SNAKE_LEN; i++)
          if (SNAKE_POS[i][0] == headX && SNAKE_POS[i][1] == headY)
              return FAIL;

    // 머리가 사과에 닿았다면 먹음
    if (APPLE_X == headX && APPLE_Y == headY)
    {
        eat(headX, headY);
        return EAT;
    }

    // 제거될 꼬리를 설정하고, 모든 몸을 한 칸 옮기고, 머리를 추가
    PRE_POS[0] = SNAKE_POS[0][0];
    PRE_POS[1] = SNAKE_POS[0][1];
    for (int i = 1; i < SNAKE_LEN; i++)
    {
        SNAKE_POS[i - 1][0] = SNAKE_POS[i][0];
        SNAKE_POS[i - 1][1] = SNAKE_POS[i][1];
    }
    SNAKE_POS[SNAKE_LEN - 1][0] = headX;
    SNAKE_POS[SNAKE_LEN - 1][1] = headY;
    return MOVE;
}
```

게임의 핵심 기능인 move()는 input()의 결과를 통해 다음으로 이동할 위치를 구하고, 그 위치가 몸 또는 벽이라면 FAIL, 사과라면 EAT, 무엇도 아니라면 MOVE를 반환한다. 만약 사과라면 사과를 먹었을 때의 함수인 eat()을 호출하며 그냥 이동이라면 뱀 몸 위치 배열의 값을 갱신한다.

snake.c ... 3

```
void newApple()
{
    int appleX, appleY, isCreated = 0;

    while (!isCreated)
    {
        // 랜덤한 값으로 사과의 위치를 설정
        appleX = rand() % MAP_X + 1;
        appleY = rand() % MAP_Y + 1;
        isCreated = 1;

        // 사과의 위치에 뱀의 몸이 있다면 다시 시도
        for (int i = 0; i < SNAKE_LEN; i++)
        {
            if (SNAKE_POS[i][0] == appleX && SNAKE_POS[i][1] == appleY)
            {
                isCreated = 0;
                break;
            }
        }
    }

    // 뱀의 몸이 없는 위치에 사과 생성
    APPLE_X = appleX;
    APPLE_Y = appleY;
}

void eat(int x, int y)
{
    // 사과의 위치에 새로운 몸을 추가
    SNAKE_POS[SNAKE_LEN][0] = x;
    SNAKE_POS[SNAKE_LEN][1] = y;
    SNAKE_LEN++;

    // 사과 생성
    newApple();
}
```

newApple()은 사과를 생성하는 기능, eat()은 길이를 증가시키고 newApple()은 사과를 생성한다. rand()는 srand()로 설정된 시드를 바탕으로 무작위 값을 뽑아내는데, 여기서 나머지 연산을 사용하면 0부터 내가 원하는 값까지의 무작위 값을 얻을 수 있다. 여기서 설정된 무작위 위

치에 뱀의 몸이 있다면 다시 위치를 설정하고, 뱀의 몸이 없다면 사과의 위치를 설정한 후 종료한다.

main

main.c ... 1

```c
#include <stdio.h>
#include <time.h>
#include "winUtil.h"
#include "snake.h"
#include "input.h"
#include "render.h"
#pragma warning(disable: 4996)

const int MSPT = 100; // 1틱에 해당하는 시간
const int MAP_X = 15; // 맵의 X 크기
const int MAP_Y = 15; // 맵의 Y 크기

void gameLoop();

int main()
{
    init(); // 뱀 기본값 설정
    initRender(); // 시작하기 전의 렌더
    gameLoop(); // 게임 루프
}
```

main 함수가 있는 main.c는 게임을 시작하기 전 설정 함수들을 호출한 다음 게임 루프를 시작한다.

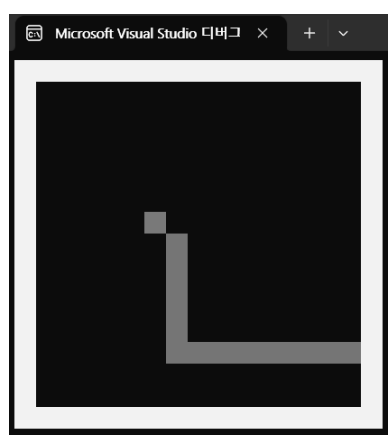
main.c ... 2

```c
void gameLoop()
{
    clock_t pre = clock(); // 이전 틱의 시간
    while (1)
    {
        input(); // MSPT와 관계없이 입력을 받음

        clock_t now = clock(); // 현재 시간
        if (now - pre >= MSPT * CLOCKS_PER_SEC / 1000) // 이전 틱으로부터 지금
            까지 MSPT 이상 지났을 경우
```

```
        {
            moveResult result = move();
            if (result == FAIL) // 게임 패배
                break;
            else if (result == EAT) // 사과를 먹었을 때
                renderEat();
            else if (result == MOVE) // 먹지 않았을 때
                renderMove();
            pre = now;
        }
    }

    // 프로그램 종료 메시지 출력을 위한 함수 호출
    setColor(BLACK, WHITE);
    gotoxy(0, MAP_Y + 4);
}
```

위에서 정의된 MSPT는 1틱에 몇 초의 시간이 소요되는지를 나타내는 변수이다. time.h의 clock()은 현재 시간을 클락 단위로 얻을 수 있는데, 이 값을 CLOCKS_PER_SEC으로 나누면 초 단위로 바꿀 수 있다. 이전의 한 시점에서 저장한 clock()과 지금 시간의 clock()을 계산하여 몇 밀리초가 지났는지를 알 수 있다. 이를 이용해 MSPT밀리초가 지날 때마다 move()와 렌더를 호출하도록 구성하였다.

실행 결과



프로그램을 시작하면 흰 벽과 빨간 사과, 그리고 파란 뱀으로 이루어진 콘솔 게임이 시작된다. 방향키를 누르면 뱀의 이동 방향이 전환되고, 뱀이 사과를 먹으면 길이가 증가하여 점수가 오른다.

이번 프로젝트에서 만든 스네이크 게임은 콘솔 게임들 중에서는 구현이 쉬운 편에 속하지만, C 언어를 배우면서 활용한 많은 지식들이 한 번에 녹아들었음을 알 수 있었을 것이다. 완전히 새로운 작품을 만드는 것도 훌륭하지만, 다른 누군가가 만들어 낸 작품을 따라서 만드는 것 또한 여러분의 실력을 키우는 좋은 연습이 된다. 기회가 된다면 스네이크 게임을 넘어서 다양한 프로젝트에 도전해 보고, 어떻게 설계를 해야 더 나은 프로그램을 만들 수 있는지 고민하면서 연습하는 시간을 갖길 바란다.

03 계산기(2권)

✓핵심 키워드

윈도우 프로그램, 계산기, 후위 표기법, 스택

컴퓨터라는 단어의 뜻에 가장 걸맞은 프로그램은 단연 계산기일 것이다. 다양한 계산기 프로그램마다 많은 기능들이 포함되어 있고, 그러한 기능들이 계산기 프로그램의 특징을 가지고 있겠지만 계산기의 핵심은 계산 그 자체에 있다. 이번 프로젝트에서는 Win32 API를 이용하여 간단한 형태의 계산기를 만들어 본다. 단, 우리가 평소에 사용하는 중위 표기법은 구현에 난이도가 있기에 대신 후위 표기법을 채택한다.

문제 파악하기

계산기 프로그램을 만드는 것은 생각보다 난이도가 있는 작업이다. 그러므로 여기서는 문제에 제약을 두어 좀 더 쉬운 형태의 계산기를 만들 것이다.

1. 연산자는 +, −, *, / 를 사용한다.
2. 피연산자는 한 자리의 0과 자연수만 사용한다.
3. 후위 표기법을 사용한다.
4. 수식에 오류가 없음을 가정한다.

4개의 제약 조건 중에서 하나라도 빠지면 구현 난이도가 큰 폭으로 증가한다. 따라서 함께 만들 프로젝트에서는 제약 조건 모두를 갖고서 만들겠다. 나중에 프로그래밍 실력을 키운 후 제약 조건 일부를 풀어서 만들어 볼 계획이라면 순서대로 1 → 4 → 2 → 3으로 제약 조건을 없앨 것을 추천한다.

UI 컴포넌트

이 프로그램에선 다른 계산기와 마찬가지로 버튼을 이용할 것이다. 필요한 버튼으로는 0부터 9까지의 숫자, 연산자인 +, −, *, /, 계산 버튼인 =, 마지막으로 작성된 수식을 지우는 C 버튼이 있다.

수식			
+	7	8	9
−	4	5	6
*	1	2	3
/	C	0	=

윈도우 창에서는 16개의 버튼과 하나의 텍스트가 이와 같은 형태로 배치될 것이다.

후위 표기법

연산자의 표기 방법중 하나인 후위 표기법은 이름 그대로 연산자가 뒤에 위치하는 표기법이다.

```
2 * (2 + 2)
2 2 2 + *
```

위는 사람이 사용하는 중위 표기법으로 표현한 것이고, 아래는 후위 표기법으로 표현한 것이다. 후위 표기법은 사람이 읽고 해석하기에는 불편하지만 중위 표기법과는 달리 연산 우선 순위를 위해서 괄호가 필요하지 않으며, 컴퓨터가 계산하기에 훨씬 유리하다는 장점이 있다.

```
2 2 2 + * 4 / 1 +
= 2 4 * 4 / 1 +
= 8 4 / 1 +
= 2 1 +
= 3
```

후위 표기법은 왼쪽부터 차례로 보며 필요한 피연산자와 연산자가 순서대로 나열되어 있을 경우, 이를 계산하여 해당 자리에 다시 쓴다. 본래 식의 2 2 +는 피연산자 둘과 연산자가 나열되어 있기에 2+2를 계산하여 4를 쓴다. 계산 후에는 2 4 *를 계산하여 8을 쓰고, 8 4 /를 계산하여 2를 쓴다. 마지막으로 2 1 +의 결과인 3이 계산의 결과가 된다. 위 식을 중위 표기법으로 나타내면 (2*(2+2))/4+1이다.

설계하기

수식 문자열

윈도우 창에 있는 =와 C를 제외한 모든 버튼은 수식을 쓰는 버튼이다. 버튼을 누르면 수식 문자열에 버튼에 해당하는 글자가 추가되고, 다 쓴 수식에 =를 누르면 수식이 계산되어 결과가 나타난다.

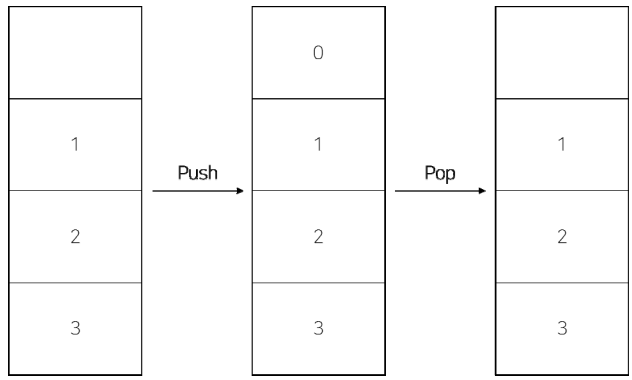
```
수식 문자열[길이] = 새 문자
수식 문자열[길이+1] = 널 문자
길이++
```

본래 존재하는 문자열의 뒤에 새로운 문자를 추가하려면, 기존 길이에 해당하는 위치에 새 문자를 넣은 후 그 뒤에 널 문자를 추가한다. 이를 모든 버튼에 적용하여 수식 문자열을 계속해서 늘리고, 수식 문자열에 변화가 있을 때마다 텍스트를 수정하도록 한다.

문제를 정의할 때 정한 제약 조건 중 하나는 입력으로 들어오는 수는 모두 한 자리 수로 가정한다고 했었다. 그렇기에 수식 문자열에 있는 0부터 9까지의 문자는 그 자체로 하나의 독립적인 수이다.

수식 계산

수식 문자열은 한 인덱스에 하나의 피연산자 또는 연산자가 있다. 남은 작업은 이 수식 문자열을 읽어서 그에 맞도록 계산을 수행하는 것이다. 후위 표기법으로 작성된 수식을 계산하는 쉽고 빠른 방법은 스택이라는 이름의 자료구조를 이용하는 것이다.

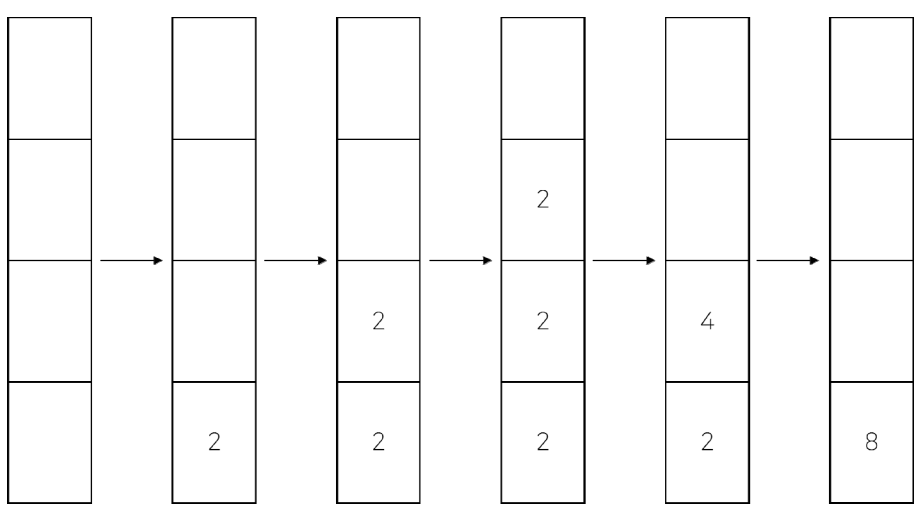


스택은 푸시와 팝이라는 두 연산이 정의되는 자료구조이다. 푸시는 스택에 데이터를 하나 집어넣고, 팝은 데이터를 제거하여 가져온다. 여기서 팝은 스택에 존재하는 데이터들 중에서 가장 최근에 추가된 데이터를 가져오는데, 이러한 특성으로, 스택은 LIFO, Last In First Out인 자료구조이다.

스택은 배열로 쉽게 구현할 수 있다. 그림에서 3이 저장된 최하단의 0번 인덱스가 가장 오래된 데이터, 그리고 인덱스가 증가될수록 최근의 데이터를 저장한다. 푸시는 마지막 인덱스에 데이터를 추가하고, 팝은 마지막 인덱스에서 데이터를 가져오게 구현하면 스택이 완성된다.

```
2 2 2 + *
```

후위 표기법의 계산은 스택 하나로 쉽게 구현할 수 있다. 빈 스택을 하나 준비한 다음, 수식 문자열의 가장 왼쪽부터 순서대로 읽으면서 숫자가 있다면 스택에 넣는다. 숫자가 아닌 연산자라면 스택에 저장된 두 데이터를 빼내어 계산한 후 다시 스택에 넣는다.



위의 간단한 후위 표기법으로 작성된 식을 계산하면 스택은 이렇게 변한다. 맨 처음 3개의 2개 연달아서 있기에 이를 모두 스택에 넣는다. 그리고 + 연산자를 만나면 스택의 맨 위 2개를 더하여 스택에 넣고, * 연산자를 만나 맨 위의 2개를 곱하여 스택에 넣는다. 모든 연산이 끝나면 최종 결과가 스택에 유일하게 남게 된다. 따라서 위 식의 계산 결과는 8이다.

구현하기

이제 Win32 API와 수식 문자열, 스택을 이용한 계산을 모두 모아 하나의 프로그램을 만들 차례이다.

win32util

win32util.h

```
#include <Windows.h>

// 메시지 처리 콜백 함수
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
    lParam);

// 윈도우 창 생성
HWND createWindow(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
     int nCmdShow, int x, int y, int width, int height);

// 버튼 생성
HWND createButton(HWND parent, int x, int y, int width, int height, const
      wchar_t* text, int id);

// 텍스트 생성
HWND createText(HWND parent, int x, int y, int width, int height, const
      wchar_t* text);
```

win32util에는 Win32 API의 함수를 모아서 쉽게 호출할 수 있도록 만든 함수들인 윈도우 창, 텍스트, 버튼 생성, 콜백 함수가 있다.

win32util.c ... 1

```
#include "win32util.h"

void buttonEvt(int id);
```

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
        buttonEvt(LOWORD(wParam));
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

buttonEvt()는 버튼 메시지를 처리하는 함수로 calc.c에서 정의되어 있다. 콜백 함수는 버튼 클릭 메시지를 받으면 버튼의 아이디를 추출하여 buttonEvt()를 호출한다.

win32util.c ... 2

```
HWND createWindow(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow, int x, int y, int width, int height)
{
    WNDCLASS wc = { 0 };
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hbrBackground = (HBRUSH)(COLOR_BACKGROUND);
    wc.lpszClassName = TEXT("MyWindowClass");
    RegisterClass(&wc);

    HWND hWnd = CreateWindow(
        TEXT("MyWindowClass"),
        TEXT("Window Title"),
        WS_OVERLAPPEDWINDOW,
        x,
        y,
        width,
        height,
        NULL,
        NULL,
        hInstance,
        NULL
```

```
    );
    ShowWindow(hWnd, nCmdShow);

    return hWnd;
}

HWND createButton(HWND parent, int x, int y, int width, int height, const
    wchar_t* text, int id)
{
    HWND hWnd = CreateWindow(
        L"BUTTON",
        text,
        WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON,
        x,
        y,
        width,
        height,
        parent,
        id,
        (HINSTANCE)GetWindowLongPtr(parent, GWLP_HINSTANCE),
        NULL);
    ShowWindow(hWnd, 1);

    return hWnd;
}

HWND createText(HWND parent, int x, int y, int width, int height, const
    wchar_t* text)
{
    HWND hWnd = CreateWindow(
        L"STATIC",
        text,
        SS_CENTER | WS_CHILD,
        x,
        y,
        width,
        height,
        parent,
        NULL,
        (HINSTANCE)GetWindowLongPtr(parent, GWLP_HINSTANCE),
        NULL);
    ShowWindow(hWnd, 1);
```

```
    return hWnd;
}
```

나머지 win32util의 함수는 윈도우를 생성하는 역할을 수행하며, 대부분 2권 챕터15 '윈도우 프로그래밍'에서 소개한 함수들이다.

stack

stack.h

```
void push(int n); // 스택 푸시 함수
int pop(); // 스택 팝 함수
```

stack은 스택을 위한 최소한의 함수인 push()와 pop()이 있다.

stack.c

```
#include "stack.h"

static int stack[1024]; // 스택 배열
static int size = 0; // 스택의 현재 크기

void push(int n)
{
    stack[size] = n;
    size++;
}

int pop()
{
    size--;
    return stack[size];
}
```

스택은 전역 변수로 값을 관리하고, push()와 pop()을 이용해 배열의 값을 넣거나 뺀다. 스택은 실제 수식을 계산하는 부분에서 활용된다.

calc

calc.h

```
void calc(); // 계산 함수
void addExpr(wchar_t ch); // 수식 문자열 추가 함수
void buttonEvt(int id); // 버튼 클릭 메시지 처리 함수
void clear(); // 초기화 함수
```

calc는 버튼 클릭 메시지를 처리하여 수식 문자열을 만들고, 수식 문자열에 대한 결과를 계산하는 역할을 한다.

calc.c ... 1

```
#include <Windows.h>
#include "stack.h"
#include "calc.h"

extern HWND text;
wchar_t expr[1024] = { 0, };
int len = 0;

void buttonEvt(int id)
{
    if (id < 10)
        addExpr(id + L'0');
    else if (id == 10)
        addExpr(L'+');
    else if (id == 11)
        addExpr(L'-');
    else if (id == 12)
        addExpr(L'*');
    else if (id == 13)
        addExpr(L'/');
    else if (id == 14)
        calc();
    else if (id == 15)
        clear();
}
```

buttonEvt()는 콜백 함수로부터 받은 아이디를 바탕으로 필요한 연산을 분기한다. 0부터 13까지는 수식 문자열에 문자를 추가하고, 14는 계산, 15는 초기화를 담당한다.

calc.c ... 2

```
void clear()
{
    // 수식 문자열을 지우고 텍스트 갱신
    expr[0] = L'\0';
    len = 0;
    SetWindowText(text, expr);
}

void addExpr(wchar_t ch)
{
    // 수식 문자열의 뒤에 새로운 문자 추가
    expr[len] = ch;
    expr[len + 1] = L'\0';
    len++;
    SetWindowText(text, expr);
}
```

clear()는 수식 문자열을 모두 비우고 텍스트를 갱신하는 기능을 하며 addExpr()은 매개변수로 받은 문자를 수식 문자열 뒤에 붙이는 기능을 한다.

calc.c ... 3

```
void calc()
{
    // 왼쪽부터 하나씩 값을 순회하며 계산
    for (wchar_t* iter = expr; *iter; iter++)
    {
        // 숫자라면 스택에 푸시
        if (L'0' <= *iter && *iter <= '9')
            push(*iter - '0');
        else
        {
            // 연산자라면 스택에서 팝한 후 값을 계산하여 다시 스택에 넣음
            int b = pop();
            int a = pop();
            if (*iter == L'+')
                push(a + b);
            else if (*iter == L'-')
                push(a - b);
            else if (*iter == L'*')
                push(a * b);
            else if (*iter == L'/')
```

```
                push(a / b);
            }
        }
        // 모든 계산이 끝난 후 스택에 남은 값이 수식의 결과
        len = 0;
        wsprintf(expr, L"%d", pop());
        SetWindowText(text, expr);
    }
```

가장 핵심 기능인 calc()는 수식 문자열의 문자를 하나씩 순회하며 값을 계산한다. 숫자에 해당하는 문자를 만나면 이를 숫자로 바꾸어 스택에 넣고, 연산자를 만나면 두 값을 빼내어 계산하고 다시 스택에 넣는다. 여기서 스택은 넣은 순서와 빼는 순서가 반대이기 때문에 빼낸 값을 뒤집어서 계산해 준다.

main

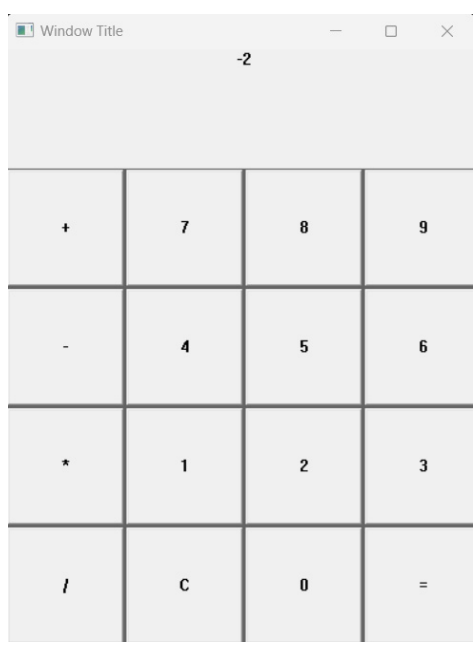
main.c

```
#include "win32util.h"
#pragma warning(disable: 4996)

HWND hWnd, text;
HWND btn[16];

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
    int nCmdShow)
{
    // 윈도우 생성
    hWnd = createWindow(hInstance, hPrevInstance, lpCmdLine, nCmdShow, 0, 0,
        410, 540);
    text = createText(hWnd, 0, 0, 400, 100, L"");
    btn[0] = createButton(hWnd, 0, 100, 100, 100, L"+", 10);
    btn[1] = createButton(hWnd, 100, 100, 100, 100, L"7", 7);
    btn[2] = createButton(hWnd, 200, 100, 100, 100, L"8", 8);
    btn[3] = createButton(hWnd, 300, 100, 100, 100, L"9", 9);
    btn[4] = createButton(hWnd, 0, 200, 100, 100, L"-", 11);
    btn[5] = createButton(hWnd, 100, 200, 100, 100, L"4", 4);
    btn[6] = createButton(hWnd, 200, 200, 100, 100, L"5", 5);
    btn[7] = createButton(hWnd, 300, 200, 100, 100, L"6", 6);
    btn[8] = createButton(hWnd, 0, 300, 100, 100, L"*", 12);
    btn[9] = createButton(hWnd, 100, 300, 100, 100, L"1", 1);
```

```
    btn[10] = createButton(hWnd, 200, 300, 100, 100, L"2", 2);
    btn[11] = createButton(hWnd, 300, 300, 100, 100, L"3", 3);
    btn[12] = createButton(hWnd, 0, 400, 100, 100, L"/", 13);
    btn[13] = createButton(hWnd, 100, 400, 100, 100, L"C", 15);
    btn[14] = createButton(hWnd, 200, 400, 100, 100, L"0", 0);
    btn[15] = createButton(hWnd, 300, 400, 100, 100, L"=", 14);

    // 메시지 루프
    MSG msg = { 0 };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return (int)msg.wParam;
}
```

마지막 main에서는 프로그램에 필요한 모든 윈도우를 생성하고 메시지 루프를 실행하는 것으로 마무리된다.

실행 결과



프로그램을 실행한 후 후위 표기법에 맞춰 수식을 작성한 후 등호를 누르면 계산 결과가 표시된다. 이번 프로젝트에서 알 수 있는 것은 CLI와 GUI는 입출력의 차이를 제외하고서는 크게 다른 것이 없다. 여기에서는 Win32 API를 이용해 버튼을 통해 수식을 작성하도록 했지만, 이를 그대로 CLI로 구현한다면 단순한 문자열 입력으로 처리할 수 있다. 그리고 그 문자열에 대한 연산은 이 계산기에서 사용된 로직을 그대로 사용하여 만들 수 있다.

CLI와 GUI, 둘 중 무엇이 더 낫다고는 말할 수 없다. 편한 입출력이 필요하다면 GUI를 택하고, 입출력에 영향을 받지 않는 프로그램이라면 CLI로 개발하는 것이 좋은 선택일 수 있다.