고려대학교 ALPS 2021

C언어 스터디

Week 0x04

함수, 포인터,
이차원 배열, 변수 범위

Contents

0x01 2D Array

이차원 배열

저번 시간에 배열을 이용해서 데이터들을 1자, 즉 일차원으로 저장하는 방법을 배웠습니다.
그런데 배열을 묶어서 이차원 배열 (행렬과 비슷한 형태)을 선언할 수 있으며,
이 개념에서는 포인터가 조금 헷갈릴 수도 있습니다.
또한, 2차원 뿐만 아니라 여러 개를 더 묶어 다차원 (n-차원) 배열을 선언하는 것도 가능합니다.

```
int arr2D[2][3] = {1, 2, 3, 4, 5, 6};
int arr2D[2][3] = {
    {1, 2, 3},
    {4, 5, 6}};
char str2D[2][6] = {
    "Hello",
    "World"};
```

2차원 배열도, 다차원 배열도 일차원 배열과 마찬가지로,
메모리 상에서 일자로 공간을 차지하므로,
첫번째 예시처럼 초기화할 수도 있고,
두번째 예시처럼 배열의 배열(int[3]을 2개 모은 배열)로 생각하셔도 됩니다.

문자열 역시 배열이므로, 2차원 배열을 이용하면 문자열 배열 또한 만들 수 있습니다.

0x01 2D Array

이차원 배열

이차원 배열 역시 인덱스를 사용하여 원소에 접근할 수 있습니다.

행렬과 형태가 비슷하다고 볼 수 있으므로,
선형대수학의 행렬을 생각해보면, 첫 인덱스를 행, 두번째 인덱스를 열로 생각하면 쉽습니다.

```
int arr2D[2][3] = {
    {1, 2, 3},
    {4, 5, 6}};
```

$$== \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

인덱스의 추가를 차원을 추가하는 것이라고 생각해보시면,
n개의 index를 이용하면 n차원 배열을 만들 수 있습니다.

원소에 접근하는 것 역시 일차원 배열과 마찬가지로 인덱스를 이용하시면 됩니다.
또한 중첩 반복문을 사용하시면 이차원 배열을 효율적으로 활용할 수 있겠죠?

```
int arr2D[2][3] = {
    {1, 2, 3},
    {4, 5, 6}};
int sum = 0;
for (int i = 0; i < 2; ++i)
  for (int j = 0; j < 3; ++j)
    sum += arr2D[i][j];
```

이 이상의 n차원 배열 역시 마찬가지입니당.

0x01 2D Array

이차원 배열의 포인터

저번 주에 포인터와 배열의 관계를 배우면서,
배열의 이름은 첫번째 원소의 메모리 주소와 같다는 것을 알 수 있었습니다.

그렇다면 이차원 배열은 어떨까요?

이차원 배열은 행렬로 생각하면 쉽지만,
메모리 상, 기본적으로는 '배열'의 '배열' 입니다.

그러므로, 배열의 원소 또한 배열이 됩니다.
오른쪽 예시로 생각해보자면,

arr2D의 첫 원소는, {1, 2, 3}, 총 3개의 int 값을
원소로 가지는 배열이라고 할 수 있겠죠.

```
int arr2D[2][3] = {
    {1, 2, 3},
    {4, 5, 6}};
```

0x01 2D Array

이차원 배열의 포인터

포인터 개념에서도 이차원 배열은 배열의 배열이라는 사실이 적용됩니다.

그러므로 이차원 배열의 이름은 배열의 첫 원소를 가리키는데
그 첫 원소가 배열이므로, 배열을 가리키는 배열 포인터가 됩니다.

즉 오른쪽 예시에 따르면, arr2D 라는 이름은 int 3개짜리 배열을 가리키는
배열 포인터가 되는 것입니다.

```
int arr2D[2][3] = {
    {1, 2, 3},
    {4, 5, 6}};
```

배열 포인터 변수의 선언은 아래와 같습니다.

```
int arr[3][4];
int (*arrPtr)[4];
arrPtr = arr;
```

0x01 2D Array

배열 포인터 주의점

배열 포인터를 선언할 때는 괄호를 꼭 넣어서 선언해야 합니다.
괄호 없이 `int *ptrArr[4];` 와 같이 선언한다면,
이것은 배열 포인터가 아닌, int 포인터 변수가 모인 포인터들의 배열이 됩니다.
(array of pointer)

또한, int 포인터와 같은 일반 포인터 변수는 배열 포인터를 담지 못합니다.
(일차원 배열은 담을 수 있음, 일차원 배열 이름은 배열 포인터가 아니라,
일반 포인터이기 때문에)

```
int arr[3][4];
int *ptr = arr; // 불가능!
```

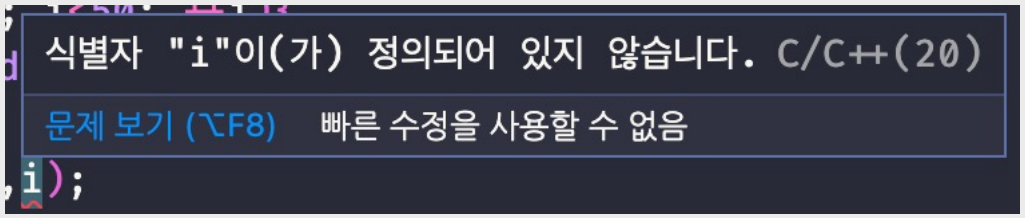
0x02 Variable Scope

변수의 범위와 수명

이렇게 짠 코드를 보시면 반복문 다음에 나오는 i 아래에
빨간 줄, 즉 에러가 발생한 것이 보이실 겁니다.

분명히 for문의 초기식에서 i를 선언했는데 왜 아래와 같은 에러가 발생할까요?

```c
#include <stdio.h>
int main(void)
{
  for(int i=0; i<50; ++i){
    printf("%d",i);
  }
  printf("%d",i);
  return 0;
}
```



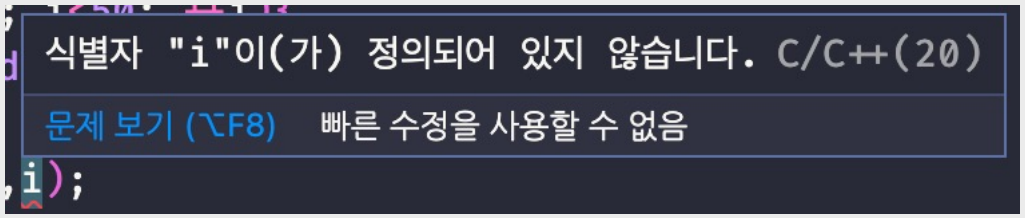
0x02 Variable Scope

변수의 범위와 수명

이렇게 짠 코드를 보시면 반복문 다음에 나오는 i 아래에
빨간 줄, 즉 에러가 발생한 것이 보이실 겁니다.

분명히 for문의 초기식에서 i를 선언했는데 왜 아래와 같은 에러가 발생할까요?

```c
#include <stdio.h>
int main(void)
{
  for(int i=0; i<50; ++i){
    printf("%d",i);
  }
  printf("%d",i);
  return 0;
}
```



0x02 Variable Scope

변수의 범위와 수명

이전 슬라이드의 코드가 에러가 나는 이유는 바로 변수의 범위 때문입니다.
변수의 수명, 즉 언제까지 존재하는지와 변수의 범위, 어디서 사용가능한지에 따라 나뉩니다.

변수는 지역 변수 Local Variable 와 전역 변수 Global Variable 로 나눌 수 있습니다.
지금까지 우리가 C언어를 공부하면서 사용했던 변수들은 모두 지역 변수입니다.

C언어에서 중괄호로 묶은 부분을 블록이라고 합니다.
우리가 if, for, switch, while 등의 문법을 사용할 때 썼던 중괄호들이 모두 블록입니다.
(단, 배열 초기화 시 사용하는 그 중괄호 제외)
(문법 없이 중괄호만 사용해도 블록 취급)

이 중괄호로 이루어진 블록 안에서 선언된 변수들을 바로 지역 변수라고 합니다.
지역 변수들은 그 변수들이 선언된 블록 안에서만 사용이 가능합니다.
이전 슬라이드 예시에서, int i는 for 문 안에서 선언되었으므로,
for문 블록 안에서만 사용가능했기에 아래의 printf에서 오류가 발생했던 겁니다.

0x02 Variable Scope

변수의 범위와 수명

```c
#include <stdio.h>
int main(void)
{
    int a = 1;
    {
        int b = 2;
    }

    do
    {
        int c = 3;
    } while (0);

    printf("%d %d %d", a, b, c);
}
```

왼쪽 예시 코드를 보시면,

b는 그냥 블럭 안, c는 do-while 블럭 안,
a는 int main 안에 있습니다.

그런데 b, c와 다르게 a는 printf와 같은 블럭에
있으므로, 에러가 발생하지 않고,

선언 된 블럭 밖에서 호출된
b와 c만 에러가 발생한 것을 볼 수 있습니당.

0x02 Variable Scope

변수의 범위와 수명

C > C ex.c > main(void)

```c
#include <stdio.h>
int main(void)
{
    int a = 1;
    {
        printf("%d\n", a);
    }
}
```

문제 출력 디버그 콘솔 터미널

```
clang: warning: treating 'c' input as
1
```

또한 블럭 구조를 중첩시킨다고 하면,
상위 블럭에서 선언한 변수는 하위 블럭에서도
사용 가능합니다.

우리가 int main 블럭 안에서 선언한 지역 변수들을
if, for 등의 블럭 안에서 사용할 수 있었던 이유와
같습니다.

0x02 Variable Scope

변수의 범위와 수명

변수는 지역 변수와 전역 변수로 나눌 수 있다고 했었죠.

그렇다면 전역 변수는 어떻게 선언할 수 있을까요?

간단합니다. int main 바깥에서 선언하면 됩니다.
main도 하나의 블럭을 구성하므로 main을 나타내는 블럭 밖에서 선언하면,
프로그램 내 모든 부분에서 사용 가능해집니다.

이전 슬라이드에서 설명했던,
상위 블럭의 지역 변수를 하위 블럭에서 사용하는 것과 비슷하다고 생각하시면 됩니다.

전역 변수에 대해서 더 자세한 사항들은
함수에서 설명하겠습니다.

0x02 Variable Scope

변수의 범위와 수명

```c
#include <stdio.h>
int main(void)
{
    int a = 1;
    {
        int a = 2;
        printf("%d\n", a);
    }
    printf("%d\n", a);
}
```

문제 출력 디버그 콘솔 터미널

```
clang: warning: treating 'c' input as '
2
1
```

하위 블럭 안에서, 상위 블럭과 이름이 똑같은
새로운 변수를 선언하면 상위 변수가
하위 변수에 의해 가려집니다.

즉, 블럭 안에서 변수를 사용할 때,
이름이 같은 변수가 있다고 하면,
가장 가까운 변수가 우선적으로 사용된다는 것입니다.

이것을 변수가 은폐된다고 합니다.

예시를 봅시다.

왼쪽 코드를 보시면, main 블럭에서 int a가 1로 초기화되면서
동시에 선언되는데, 그 이후 안쪽 블럭에서,
새로운 a가 2로 초기화되면서 선언됩니다.
안쪽 변수가 사용되기에 블럭안에서는 2가 출력되고,
블럭을 벗어나면서 2인 a는 소멸되고 다시 바깥의 1인 a가 사용되어
블럭 밖에서는 1이 출력됩니다.

0x02 Variable Scope

변수의 범위와 수명

```c
#include <stdio.h>
int main(void)
{
    int a = 1;
    {
        a = 2;
        printf("%d\n", a);
    }
    printf("%d\n", a);
}
```

문제 출력 디버그 콘솔 터미널

```
clang: warning: treating 'c' input as
2
2
```

이번 새로운 예시를 보면,
이전 예시와 다르게, 블럭 안에서 a를 사용하기는 하지만,
새롭게 선언하지는 않습니다.

사실 지금까지 사용해왔던 방식과 같습니다.
Main에서 선언된 a가 계속해서 사용되는 것이기에,

블럭 안에서 값이 2로 바뀌면 블럭 밖에서도 적용이 되어
2, 2가 출력됩니다.

a(1)이 계속해서 사용됨

이런 변수 은폐는 흐름을 잘 파악하지 않으면
실수하기 쉬우므로 문제로 자주 출제되며
실제로 코드 짤 때에는 변수
이름을 구분하는게 좋아용

0x02 Variable Scope

변수의 범위와 수명

변수는 또한 프로그램이 실행될 때 어디에 저장되는가에 따라서도 나눌 수 있습니다.
이것을 기억 부류 storage class 라고 하는데, 이는 변수의 범위, 수명과도 관계가 있습니다.

키워드	저장 장소	범위	초깃값	수명
extern	data / BSS	program	0	program
auto	stack	block	uninitialized	block
static	data / BSS	block or file	0	program
register	CPU register	block	uninitialized	block

기억 부류를 지정하려면, 선언할 때 int 등의 자료형 전에 위의 기억 부류 지정자 4가지 중 하나를 사용하면 되는데, 이 중 auto와 static이 기억해줄 만 합니다.

auto는 우리가 선언했던 모든 변수들입니다.
자동 변수 automatic variable 은 지역 변수와 비슷한데,
선언할 때 변수의 저장 공간에 자동으로 들어가 그 변수의 범위를 벗어나면
자동으로 해제되는 그런 변수라고 생각하시면 됩니다.
굳이 붙이지 않아도 기본적으로 설정되는 변수이며, 표의 수명 란을 보면 block입니다.

0x02 Variable Scope

변수의 범위와 수명

그럼 static 변수는 무엇이냐면,

초기화가 단 한번만 가능한 변수입니다.

오른쪽 예시를 보시면
autoCount는 반복마다 초기화되므로
각 반복 시 0으로 초기화되어
00000이 출력됩니다.

하지만 staticCount는 단 한번만
초기화되므로,
0에서 시작하여 계속 증가해서
01234가 출력됩니다.

```c
#include <stdio.h>
int main(void)
{
    for (int i = 0; i < 5; ++i)
    {
        int autoCount = 0;
        printf("%d", autoCount++);
    }
    printf("\n");
    for (int i = 0; i < 5; ++i)
    {
        static int staticCount = 0;
        printf("%d", staticCount++);
    }
}
```

```
문제  출력  디버그 콘솔  터미널
clang: warning: treating 'c' input as 'c++' when
00000
01234
```

0x03 Function

함수

오늘 세번째로 배울 내용은 바로 함수입니다.
함수는 프로그램이 담당하는 기능 중 하나를 따로 빼내어 구현하고자 사용합니다.

예를 들어서, 계산기 프로그램을 만든다고 하면,
수를 입력받는 함수, 계산을 하는 함수, 출력을 하는 함수 세가지로 나눠
프로그램을 짤 수 있겠죠.

이렇게 함수를 사용하는 것은 기계에서 부품을 만드는 것과 같습니다.
즉, 프로그램을 짜면서 틀린 부분을 바꾸거나,
유지 보수를 위해 수정을 하거나, 다른 프로그램에 같은 기능을 넣을 때 이점이 있습니다.

우리가 지금까지 코드를 짜면서 본 int main 이라는 함수가 바로 프로그램의 몸체가 되는
함수인데, 그 main에 모든 기능을 넣어두면, 마치
모든 부품이 일체형인 기계처럼 불편함이 많습니다.

예를 들면, 일체형 배터리는 고장이 나면 휴대폰 전체를 수리해야 하는 것처럼요.

0x03 Function

함수

함수도 변수와 마찬가지로 선언을 해야합니다.

함수 선언 전에, 수학에서의 함수를 생각해 봅시다.

수학에서의 함수는 보통,
$y = f(x)$로 나타냅니다.
이 함수를 보면 x에는 정의역이 필요하겠고, y에는 공역,
f라는 함수 자체에는 그 함수의 계산이 필요하겠죠.

프로그래밍에서의 함수도 비슷하게 이러한 사항들이 정의되어야 합니다.

선언 방법은 다음과 같습니다.
main 함수 위에서,

```
반환형 함수이름 (매개변수) {
        // 함수 연산
}
```

0x03 Function

함수

함수는 전달인자 Argument를 입력받아 연산을 한 뒤, 특정 값을 반환 Return 하고 함수를 종료
(함수 값을 반환하지 않을 수도 있습니다.)

방금 본 선언에서 반환형이 바로 반환하는 결과값의 자료형을 말하는 것이며,
매개 변수는 입력 받는 값의 목록입니다.

수학에서 f(x)=x+1을 계산하는 함수를 생각해봅시다.
이 함수는 실수를 입력 받아서 1을 더해 반환하는 함수이죠.
그러므로 정의역, 공역은 실수라고 할 수 있겠고, 함수 연산은 +1 이라고 할 수 있습니다.

이것을 C 언어로 바꿔 생각해보면, 반환형은 double형이며,
매개 변수 parameter 는 실수형 즉 double 형인 x를 하나 입력 받는다고 할 수 있겠죠.

```c
double f(double x)
{
  return x + 1;
}
```

double 형이 함수의 반환형 (x+1의 자료형)
f는 함수의 이름
(double x)는 입력 받을 매개변수의 자료형과 이름(여러 개 가능)
return은 함수 값 반환

0x03 Function

함수

그러면 수학적으로가 아니라 프로그래밍의 사고에서 함수에 대해 생각해봅시다.

int 변수 두가지를 더하여 반환하는 함수를 생각해봅시다.

```
int add(int a, int b)
{
  int result = a + b;
  return result;
}
```

수학에서 f(x)를 사용하듯이 함수 값을 사용하면 됩니다.

```
printf("2 + 3 = %d", add(2, 3));
```

반환형을 'void'라고 하면 반환을 하지 않아도 되고,
입력 매개 변수는 전달받지 않아도 됩니다.
콘솔 창에서 변수를 입력받아 더해서 출력하는 함수를 생각해봅시다.

오른쪽 함수는 매개변수와 반환형 없이
단순히 정수 값 두 개를 입력 받아 더하여 출력하는
'기능'만을 담당합니다.

```
void compute()
{
  int num, result = 0;
  for (int i = 0; i < 2; ++i)
  {
    scanf("%d", &num);
    result += num;
  }
  printf("%d", num);
}
```

0x03 Function

함수 프로토 타입

main 함수 위에 여러 함수들을 선언해두면 지저분해지겠죠.
하지만 그렇다고 해서
main 함수 아래에 함수를 선언한다면,
c 언어는 위에서부터 아래로 코드를 해석하므로 컴파일러가 코드를 이해할 수 없습니다.

이것을 해결하기 위한 방법이 바로 함수 프로토 타입 Function prototype입니다.

프로토 타입 이라는 말 들어보신 적이 있나요?
쉽게 말하면 '기초 형태'를 프로토타입이라고 합니다.

즉 함수의 기초 형태, 함수 연산이 정의되는 블록을 제외하고,
반환형, 함수 이름, 매개 변수 자료형만 정의를 해 두는 것입니다.

이전 슬라이드의 add 함수를 예로 들자면,
int add(int a,int b) 또는 int add(int, int) 이렇게만 main 위에 선언해두고
main 함수가 끝난 뒤
함수의 정의를 자세하게 적어두는 겁니다.

0x03 Function

메인 함수

메인 함수는 프로그램의 뼈대가 됩니다.

프로그램 실행 시 main 함수가 실행되는 것이고,
main 함수로부터 다른 함수를 호출하고, 여러 변수를 사용합니다.

main 함수의 반환형을 보시면 int입니다.

이는 프로그램이 정상적으로 종료될 시 0, 아니면 0이 아닌 수를 반환하는 것입니다.
예시로 에러 발생 시 return 1을 실행하면 됩니다.
그렇게 중요한 얘기는 아니구 혹시 궁금하실 까봐.. 넣어뒀습니다.

참고로 return 시 값을 반환하는 것이므로 함수는 종료됩니다.

0x03 Function

함수 호출하기

int 변수를 매개 변수를 받아 3을 더해주는 함수를
생각해봅시다.
그렇다면 오른쪽과 같이 코드를 짤 수 있을 텐데,
결과 값을 보시면,
03 이 아니라 00이 출력됩니다.

왜 그럴까요?

이런 문제를 이해하기 위해서는,
함수의 호출 방식에 대한 이해가 필요합니다.

함수 호출 방식은
Call by value
Call by reference
두가지로 나눌 수 있습니다.

다음 슬라이드에서 자세히 알아봅시다.

```c
#include<stdio.h>

void add3(int num)
{
    num += 3;
}
int main(void)
{
    int a = 0;
    printf("%d\n", a);
    add3(a);
    printf("%d", a);
}
```

문제 출력 디버그 콘솔 터미널

```
clang: warning: treating 'c' input as 'c
0
0
```

0x03 Function

함수 호출하기

일단 call by value는 인자를 값으로 전달해 주는겁니다.
변수 자체의 주소와는 상관 없이 값을 복사해서 함수 안에서 처리하게 됩니다.

즉, 함수 안에서 새로운 지역 변수를 선언하고,
그 변수에 인자로 받아온 변수의 값을 대입하여 사용한다고 생각하면 됩니다.

그러므로 함수 안에서 받은 인자에 연산을 하더라도 함수가 끝나고
다시 돌아와 값을 확인해보면 값이 달라지지 않는 것입니다.

이전 슬라이드의 예시가 바로 call-by-value 방식입니다.

주로 변수의 값 자체만을 사용하고자 할 때 사용합니다.

0x03 Function

함수 호출하기

call by reference 또한
기본적 원리는 call by value와 같습니다.

대신 변수의 값을 가져와 사용하는 것이 아니라,
매개 변수로 포인터 변수를 받아와,
사용할 변수의 메모리 주소를 복사하여 포인터 변수에 대입하여 이용하는 것입니다.

함수 안에서 함수 바깥에 저장된 변수의 메모리에 접근할 수 있으므로,
함수 안에서도 바깥의 변수의 값을 변경할 수가 있습니다.

변수 두 가지의 값을 바꾸는 함수의 예시를 봅시다.

```
void swap(int *a, int *b)
{
  int tmp = *b; // 역참조
  *b = *a;
  *a = tmp;
}
```

오른쪽 예시를 보시면
매개 변수로 int 포인터 a,b를 받아오고,
역참조를 이용해 함수 바깥의 변수의 값을 조정합니다.

이것 또한 포인터가 중요한 이유라고 볼 수 있습니다.

0x03 Function

배열 인자로 받기

배열을 함수의 인자로 받을 수도 있습니다.

일단 1차원 배열은, 배열 이름이 포인터 자체라고 했으니, 그냥 포인터 형식으로 받으면 되고,
2차원 배열은 배열 이름이 배열을 가리키는 포인터라고 했으므로
약간 다른 포인터 형식으로 입력받으면 됩니다.

1차원 배열
function(int *arr) 또는, function(int arr[]) //
배열을 나타내는 대괄호를 이용해도 됩니다.

2차원 배열
function(int (*arr)[4]) 또는 function(int arr[][4])
배열의 원소 개수에 따라 다르게 선언하면 됩니다.

0x03 Function

함수와 변수

전역 변수는 main 함수뿐만 아니라 다른 함수에서도 사용이 가능합니다.

```c
#include <stdio.h>

int cnt = 3;
int add(int a, int b)
{
  cnt++;
  return a + b;
}

int main(void)
{
  printf("%d %d", add(2, 3), cnt);
}
```

문제 출력 디버그 콘솔 터미널

```
clang: warning: treating 'c' input as 'c++' when in
5 4
```

0x03 Function

함수와 변수

static 변수는 한번만 초기화되고,
이전의 표를 보시면 수명이 program이라고 했으므로,
함수에서 선언되는 static 변수는 함수가 종료되어도
사라지지 않고 메모리에 남아있습니다.

오른쪽 예시를 보면
함수가 여러번 호출되어도
한번만 초기화되고,

함수가 종료되어도 값이 남아있음을 알 수 있습니다.

```c
#include <stdio.h>

void func()
{
    static int cnt = 0;
    printf("%d\n", cnt++);
}

int main(void)
{
    func();
    func();
    func();
}
```

```
문제  출력  디버그 콘솔  터미널

clang: warning: treating 'c' input as 'c
0
1
2
```

0x03 Function

재귀 함수

함수에서 함수를 호출하는게 가능합니다..

이게 무슨 소리냐.

> 어느 한 컴퓨터공학과 학생이 유명한 교수님을 찾아가 물었다.
> "재귀함수가 뭔가요?"
> "잘 들어보게. 옛날옛날 한 산 꼭대기에 이세상 모든 지식을 통달한 선인이 있었어. 마을 사람들은 모두 그 선인에게 수많은 질문을 했고, 모두 지혜롭게 대답해 주었지. 그의 답은 대부분 옳았다고 하네.
> 그런데 어느 날, 그 선인에게 한 선비가 찾아와서 물었어.
> **"재귀함수가 뭔가요?"**
> "잘 들어보게. 옛날옛날 한 산 꼭대기에 이세상 모든 지식을...

이런겁니다…
솔직히 이해가 안되실텐데,

수학적으로 생각해보시면
base condition이라고 부르는 종료 조건에 이르기까지 반복해서 함수를 호출함으로써
문제를 해결하는 것입니다.

0x03 Function

재귀 함수

피보나치 수열로 간단하게 예시를 보고, 넘어갑시다.

좀 더 익숙해지면, 알고리즘을 공부하면서 재귀 함수를 많이 보실거에요.

i번째 피보나치 수열을 fibo(i)라고 하면,

피보나치 수열은
fibo(0) = 1, fibo(1) = 1

fibo(i) = fibo(i-1) + fibo(i-2)
로 정의되는 수열입니다.

피보나치 수열이 재귀로 구현하는 가장 간단한 예시라고 할 수 있습니다.

재귀가 지금은 이해가 잘 안가실 텐데,
나중에는 재귀로 하는게 더 이해가 편하실거에요

아무튼 코드를 봅시다

0x03 Function

재귀 함수

```
int fibo(int i)
{
  if (i == 0 || i == 1)
  {
    return 1;
  }
  else
    return fibo(i - 1) + fibo(i - 2);
}
```

마치 점화식 같은거죠.
fibo(5)를 호출한다고 하면,
5는 0이나 1이 아니므로, fibo(4)+fibo(3)을 반환합니다.

그리고 이 값을 계산하기 위해서는 fibo(4), fibo(3)을 호출해야 하죠.
그러면 다시 호출된 fibo(4)는 fibo(3), fibo(2)를 부르고…

각각 0,1(종료 조건)을 호출할 때 까지 같은 함수를 반복해서 호출하게 되는 겁니다.

이해하고 보니까 좀 더 간단하게 느껴지지 않나요..?
암튼.. 굿

0x05 Assignment

과제

https://www.acmicpc.net/problem/10870
https://www.acmicpc.net/problem/10872

https://www.acmicpc.net/problem/4673
https://www.acmicpc.net/problem/11729

2문제 이상/
위에 두개는 쉽고 밑에 두개는 좀 어렵습니당
이왕이면 저한테 마음껏 질문하시면서 다 풀어보세요!

0x04 Practice

소개

https://solved.ac/class
https://codeup.kr/problemset.php
https://www.acmicpc.net/step

질문 환영~~~

부족한 수업
들어 주시느라 수고 많으셨습니다!

들어주셔서 감사하고, 질문과 밥약은 언제든 환영이에요

다음 주에 봅시다~