고려대학교 ALPS 2021

C언어 스터디

Week 0x03

배열, 포인터, 문자열

Contents

0x01 Array

배열이란

먼저 “배열” 부터 알아보도록 합시다.
배열은 int, double, char 등의 한 ' 자료형 ' 을 모아놓은 집합이라고 할 수 있습니다.
배열을 사용하는 이유가 뭘까요?

다음처럼 학생 30명의 성적 평균을 계산한다고 생각해봅시다.

```
float score1 = 80;
float score2 = 84;
// ... (생략)
float score29 = 72;
float score30 = 91;

float sum = score1 + score2 + ... (생략) ... + score29 + score30;
float average = sum / 30;
```

이런 식으로 코드를 짜면 반복문을 사용하기도 껄끄럽고 힘들겠죠?
만약에 학생 수가 3천명이라고 하면 더 어렵겠죵

배열은 이런 반복되는 작업이나, 자료 저장을 반복문과 결합하여 더욱 효율적이고 편하게 합니다.

0x01 Array

배열이란

배열은 변수를 선언하는 것과 비슷한 방식으로 쉽게 선언할 수 있습니다.
아까 배열은 한 자료형을 묶어둔 집합이라고 했었죠, 그러므로 배열을 선언할 때는
변수 선언처럼 배열의 **이름**과 **자료형**, 배열의 **크기**만 있으면 됩니다.

```
// 자료형 배열이름[크기];
int arr1[10];
int arr2[2] = {1, 2};
int arr3[10] = {0,};
int arr4[] = {1, 2};
```

왼쪽과 같이 배열을 선언할 수 있는데,
첫번째는 선언만 한 예시이고, 그 아래는 변수처럼 초기화를 한 예시
선언에는 **[대괄호]**, 초기화에는 집합처럼 **{중괄호}**가 필요합니다.
모든 원소를 초기화 할 수도, 일부만 초기화 할 수도 있습니다. 일부만 초기화 하면 나머지 요소는 모두 0으로 초기화됩니다.

4번째 예시처럼 대괄호 안에 배열 크기를 넣지 않고 초기화를 하면 자동으로 배열의 크기는 초기화 한 원소의 개수(2)가 됩니다.

이렇게 선언한 배열에 접근할 때는, 선언과 마찬가지로 **[대괄호]**를 이용합니다.

arr2[1] 이런 식으로 값에 접근할 수 있는데, 중요한 점은
배열의 원소는 **0부터** 시작한다는 점입니다. 즉 arr2[1]은 1이 아니라 2를 값으로 저장하고 있습니다.
새로운 값 할당 또한 대괄호를 이용해서 arr1[1] = 10; 과 같이 하시면 됩니다.

0x01 Array

Zero Based Numbering & Half-Opened Interval

이와 같은 배열의 요소 번호를 **인덱스** ***index*** 라고 합니다.
방금도 봤듯이 배열 번호, 인덱스는 0부터 시작합니다.
이것을 ***zero-based numbering*** 이라고 합니다.

More
Dijkstra: Why numbering should start at zero
https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

가장 큰 이유는 이따가 배열과 포인터의 관계를 다룰 때 다루도록 하고,

다른 이유로는 관습적인 이유가 있습니다.

제가 저번 시간에 써드린 반복문 예시나, 많은 C언어 서적들, 컴프 수업 등에서 보시는 반복문 예시에서는 N번 반복한다고 하면, for(int i=0;i<N;++i) 이런 식으로 쓰는 걸 볼 수 있습니다.

이것은 **반 닫힌 구간** ***Half-open interval*** 이라고 하는데, 이 개념은 수학 시간에 보신 적이 있을 겁니다. 위의 반복문 구간(0<=i<N)을 [0,N) 이런 식으로 표현하는 걸 배우셨을 거에요.

이런 것들도 C의 배열 인덱스가 0부터 시작하는 이유 중 하나입니다.
반 닫힌 구간을 쓰는 이유를 다음 슬라이드에서 살펴보도록 하겠습니다.

0x01 Array

Zero Based Numbering & Half-Opened Interval

반 닫힌 구간이 좋은 이유로,

1) 마지막 수와 시작 수의 차가 집합의 크기와 같습니다.
N번의 반복을 한다고 할 때,
1<=i<=N [1,N] 보다는 0<=i<N [0,N) 이 더 깔끔하고 쉽게 표현할 수 있겠죠.
1<=i<=N은 반복문으로 표현하면 for(int i=1;i<=N;++i) 인 반면
0<=i<N은 반복문으로 표현하면 for(int i=0;i<N;++i)로 좀 더 간단하고,
N-0은 N이라는 점에서 반복 횟수를 바로 짐작할 수 있습니다.

2) 구간을 합칠 때 편리합니다.
[0,5), [5,10) 처럼 반 닫힌 구간을 합치면 중복되거나 비는 부분 없이 깔끔하게
[0,10) 처럼 나타낼 수 있으며, 구간을 나눌 때도 깔끔하게 나눠집니다.

3) 시작 인덱스를 포함합니다.
(0,10), (0,10]처럼 시작구간을 포함하지 않으면 구간을 표현할 때 최소 인덱스보다 작은 수를 써야합니다. -1<i<10 처럼 말이죠.

위와 같은 이유로 반 닫힌 구간을 사용하는데, [1,N+1) 보다는 당연히 [0,N) 이 깔끔하므로,
zero-based numbering과 함께 쓰는 것이죠. 이런 이유들에서 for문도 0부터 시작하면 좋겠죠??
재밌지 않나요 ㅎㅎㅎ.. 😄

0x01 Array

문자열

배열에서도 특히 char로 이루어진 배열을 문자열이라고 합니다.
이것은 저번 시간에 배운 아스키 코드, 즉 문자를 배열 형태로 저장하여 나타냅니다.
아스키 코드를 저정하므로, 문자열 중간에 '\n' 같은 이스케이프 시퀀스를 넣으면
중간에 개행도 가능합니다.

이것도 배열의 한 종류일 뿐인데 굳이 왜 따로 설명하냐면,
문자열을 string이라고 하여 특별하게 다루는 언어가 많고, (java 등의 객체 지향 언어)

C에도 문자열만을 다루는 라이브러리가 있으며,
문자열에는 배열과 다르게 문자열만이 가지고 있는 특징이 있기 때문입니다.

그 특징은 바로 문자열은 마지막에 '\0'이라는 문자를 가집니다.
이것은 아스키 코드로 0이며, NULL 문자라고 합니다.

NULL 문자는 문자열의 끝을 구분하는 역할을 합니다.
이러한 NULL 문자 덕에, 문자열의 길이를 구하거나, 복사하는 등의 함수가 라이브러리에 정의되어
있으며, 문자열을 다루는 함수는 보통 NULL 문자를 활용합니다.

0x01 Array

문자열

예로 들어 “Hello, c!”라는 문자열을 저장해본다고 합시다.

인덱스	0	1	2	3	4	5	6	7	8	9
문자	h	e	l	l	o	,		C	!	\0
10진수	104	101	108	108	111	44	32	67	33	0

왼쪽과 같이 문자열 마지막에 NULL 문자가 포함되어 있으며, 공백이나 쉼표 같은 문자 또한 저장할 수 있습니다.

또한, 문자열은 초기화 시 중괄호가 아니라, 큰 따옴표로 구성해도 됩니다.
다시 말해, `char string[] = "Hello World!";` 와 같이 초기화 할 수 있으며
가장 끝에는 NULL 문자가 자동으로 들어갑니다.

이런 초기화 방식을 보면, printf(“Hello World!”); 의 그것과 비슷하죠.
우리가 지금까지 쓰던 printf 속 큰 따옴표도 사실은 문자열이랍니당.

위에 적힌 “Hello World!”와 같이 변수나 배열에 들어 있지 않고 코딩되어 있는 문자열을
“문자열 상수”라고 합니다.

이러한 문자열 상수들은 코드 실행 시 정적 영역에 저장되어 있는데, 이로 인한 문제들은
포인터와 함께 다뤄보도록 합시다.

이런 문자열들은 단어로 의미를 이룰 것이므로, 문자열의 길이는 각각 다를 수 가 있겠죠.
이런 상황에서 길이가 계속 달라지는 문자열의 '끝’이자 구분 역할을 해 주는게 바로 NULL 입니다.

0x02 Pointer

포인터

C언어의 난관으로 악명 높은 포인터.
왜 필요한지, 그리고 정확히 무엇을 말하는지만 알면 쉽게 배울 수 있다고 생각합니다.

C에서 데이터들은 메모리라는 공간에 저장되며,
그 메모리는 일자로 존재한다고 배웠었죠.

데이터가 저장된 메모리 주소를 나타내는 것이 바로 **포인터**입니다.
'가리킨다'라는 point의 뜻에 주목해보면, 메모리 주소를 가리킨다는 의미에서 포인터란 이름이 붙었다고 볼 수 있겠습니다.

즉 포인터는 메모리 주소를 저장하고 있는 변수라고 할 수 있겠습니다.

한 변수가 차지하는 메모리 공간 나타내는 것인데, 각 변수의 자료형 마다
1 byte, 4 byte, 8 byte 등 크기가 다르다고 했었죠.

이처럼 크기가 다른 변수들을 나타내기 위하여 포인터도 자료형을 선언해야합니다.
다음 슬라이드에서 더 알아보도록 하겠습니다.

0x02 Pointer

포인터

포인터에서 쓰이는 특별한 연산자로 &, *(애스터리스크 asterisk)가 있습니다.

& 연산자는 scanf에서 본 적 있었죠. &는 주소 연산자로, 해당 변수가 저장되어 있는 메모리 주소를 반환합니다.
* 는 포인터 변수를 선언할 때 사용합니다.

포인터 변수를 선언하고 초기화하는 방법은 다음과 같습니다.

```
// 자료형 *포인터 이름;
// 포인터 = &변수;
int num = 5;
int* numPtr1 =  &num;
int * numPtr2 = &num;
int *numPtr3 = &num;
```

*는 자료형 바로 뒤, 자료형과 포인터 명 사이, 포인터명 바로 앞 세 군데 중 어디에 붙여도 되고, 보통은 포인터명 바로 앞에 붙입니다. (numPtr3)

numPtr1,2,3 이라는 포인터 변수에, num 변수가 저장되어 있는 메모리의 주소를 저장한다는 뜻입니다.

현재 numPtr1 변수에는 num의 메모리 주소가 저장되어 있는 상태입니다.
출력(%p)을 통해 값을 알아보면 다음과 같이 16진수로 보여지는 것을 알 수 있습니다.

```
int num = 5;
int *numPtr1 = &num;
printf("address: %p", numPtr1);
```

```
address: 0x7ffee2e4468c
```

0x02 Pointer

포인터

포인터에서 쓰이는 특별한 연산자로 &, *(애스터리스크 asterisk)가 있습니다.

& 연산자는 scanf에서 본 적 있었죠. &는 주소 연산자로, 해당 변수가 저장되어 있는 메모리 주소를 반환합니다.
* 는 포인터 변수를 선언할 때 사용합니다.

포인터 변수를 선언하고 초기화하는 방법은 다음과 같습니다.

```
// 자료형 *포인터 이름;
// 포인터 = &변수;
int num = 5;
int* numPtr1 =  &num;
int * numPtr2 = &num;
int *numPtr3 = &num;
```

*는 자료형 바로 뒤, 자료형과 포인터 명 사이, 포인터명 바로 앞 세 군데 중 어디에 붙여도 되고, 보통은 포인터명 바로 앞에 붙입니다. (numPtr3)

numPtr1,2,3 이라는 포인터 변수에, num 변수가 저장되어 있는 메모리의 주소를 저장한다는 뜻입니다.

현재 numPtr1 변수에는 num의 메모리 주소가 저장되어 있는 상태입니다.
출력(%p)을 통해 값을 알아보면 다음과 같이 16진수로 보여지는 것을 알 수 있습니다.

```
int num = 5;
int *numPtr1 = &num;
printf("address: %p", numPtr1);
```

```
address: 0x7ffee2e4468c
```

0x02 Pointer

포인터

*의 다른 용법으로, 포인터가 나타내고 있는 메모리 주소에
어떤 값이 저장되어 있는지 알 수 있습니다.
이것을 간접 참조 Dereference 라고 합니다.

포인터에 자료형이 필요한 이유도 이것 때문입니다.

자료형마다 크기가 다르므로, 간접 참조를 할 때 몇 byte만큼 메모리에 저장된 비트를 읽어들여야 할지 명시해야하기 때문입니다.

```c
int num = 5;
int *numPtr1 = &num;
printf("address: %p\n", numPtr1);
printf("value: %d", *numPtr1);
```

```
address: 0x7ffeec24b68c
value: 5
```

간접 참조 연산자를 이용하면 값을 바꾸는 것도 가능합니다.

```c
int num = 5;
int *numPtr1 = &num;
*numPtr1 = 10;
printf("address: %p\n", numPtr1);
printf("value: %d, %d", *numPtr1, num);
```

numPtr은 num의 메모리 주소와 같고, numPtr이 가리키고 있는 메모리 주소, 즉 num에 10을 저장하므로 num의 값도 10이 됩니다.

```
address: 0x7ffee4c3e68c
value: 10, 10
```

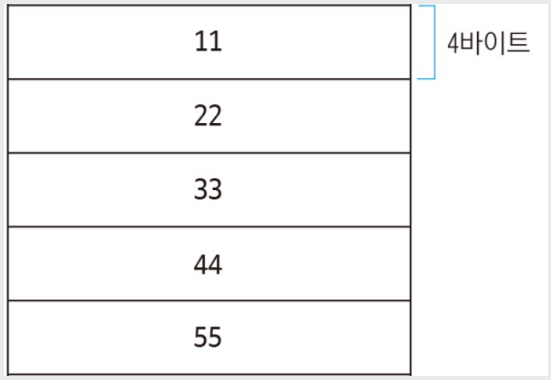
0x03 Array & Pointer

배열 포인터

배열과 포인터는 사실 거의 같은 개념입니다.
여러분이 배열을 선언하시면, 그 배열의 이름은 바로, 배열의 첫 원소를 담고 있는 포인터입니다.

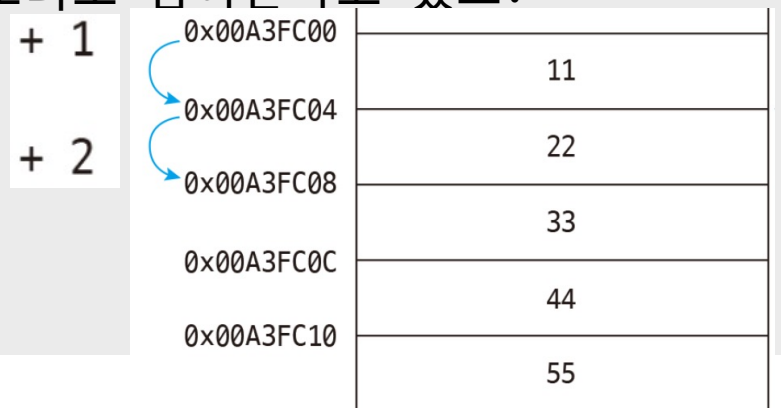
컴퓨터의 메모리는 일자로 이루어져 있다고 했었죠.
배열을 사용하는 이유가 바로 여기에 있습니다.
배열은 메모리 상에서 순서대로 메모리를 차지합니다.
즉 int 5개짜리 배열은 int의 크기 4byte * 5개 , 총 20byte 만큼의 메모리를 차지하고 있는 것이죠.

포인터에 덧셈 연산을 하면, 자료형의 크기 만큼 다음 메모리로 넘어갑니다.
즉 int 포인터 + 1 을하면 4byte 다음 메모리를 가리키는 것이죠.



배열은 왼쪽과 같이 메모리를 차지하고 있습니다.

그리고 포인터 덧셈을 하면 다음 메모리로 넘어간다고 했죠.



그러므로, 오른쪽과 같은 방식으로 더해주면
다음 원소에 접근할 수 있습니다.

0x03 Array & Pointer

배열 포인터

다음 코드의 실행 결과는 아래와 같습니다.

```c
#include <stdio.h>

int main(void)
{

  int arr[] = {1, 2, 3};
  int *ptr = arr;

  printf("%p %p | %p %p == %p %p\n", arr, ptr, arr + 1, ptr + 1, &arr[1], &ptr[1]);
  printf("%d %d | %d %d == %d %d", *(arr), *(ptr), *(arr + 1), *(ptr + 1), arr[1], ptr[1]);

}
```

```
0x7ffee63e167c 0x7ffee63e167c | 0x7ffee63e1680 0x7ffee63e1680 == 0x7ffee63e1680 0x7ffee63e1680
1 1 | 2 2 == 2 2
```

즉, arr+i == &arr[i], *(arr+i) == arr[i] 이것만 기억하시면 편할 것 같습니다.
이것이 배열을 사용하는 가장 큰 이유들 중 하나입니다.
포인터 연산을 통해 원하는 인덱스에 바로 접근이 가능합니다.
그리고 전에 언급했던 배열 인덱스가 0부터 시작하는 가장 큰 이유이기도 하죠.

0x04 Practice

실습

```
예시 -
https://www.acmicpc.net/problem/1157
```

0x05 Assignment

과제

https://www.acmicpc.net/problem/1546 - 1546 : 평균 – 배열
https://www.acmicpc.net/problem/4344 - 4344 : 평균은 넘겠지 – 배열
https://www.acmicpc.net/problem/2577 - 2577 : 숫자의 개수 - 배열
https://www.acmicpc.net/problem/1152 - 1152 : 단어의 개수 - 문자열
3문제 이상
이왕이면 저한테 마음껏 질문하시면서 다 풀어보세요!

부족한 수업
들어 주시느라 수고 많으셨습니다!

들어주셔서 감사하고, 질문과 밥약은 언제든 환영이에요

다음 주에 봅시다~