8 포인터 II

8-1 배열과 포인터

8-1-1 포인터를 이용하여 배열 다루기

8-1-2 이차원 배열로의 확장

8-1-3 배열 포인터

8-1-4 포인터 배열

8-2 문자열과 포인터

8-2-1 문자열의 종류

8-2-2 문자열의 종류에 따른 포인터 이용하기

8-2-3 const 키워드에 대한 정성적 이해

8-3 구조체와 포인터

8-3-1 포인터 배열의 확장: 구조체 이용하기

8-3-2 구조체 포인터 배열의 동적 할당

8-3-3 구조체끼리 연결하기 (Linked list)

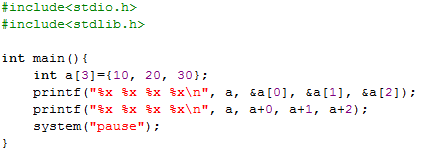
Cf. 함수와 포인터 단원은 11장 함수의 활용에 수록

8-1 배열과 포인터

포인터를 이용하면 배열과 같은 데이터를 쉽고 빠르게 처리할 수 있다. 이번에는 포인터와 배열이 어떻게 상호작용 하는지 알아볼 것이다.

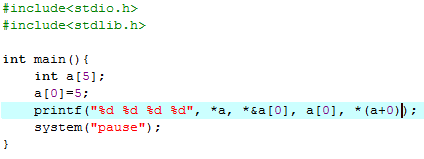
8-1-1 포인터를 이용하여 배열 다루기

앞서 7장에서 주소와 포인터의 개념을 잘 학습하였다면, 배열 단원에서 언급되었던 배열의 이름이 주소의 역할을 한다는 것이 조금 더 쉽게 다가올 것이다. 배열의 주소를 나타내는 방법은 배열의 이름을 통한 방법뿐만 아니라 다른 방법들도 있다. 다음 코드를 보면서 다양한 주소 표현을 이용해 배열 안의 값에 접근해보자.



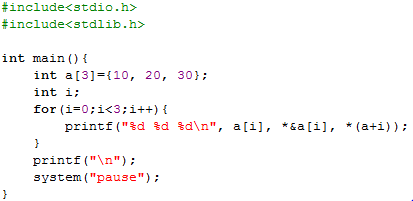
첫 번째로 출력하는 줄은 값에 &를 붙여서 주소를 출력한 것이라는 것은 쉽게 알 수 있을 것이다. 두 번째 줄을 보자. a가 배열의 주소라는 것은 당연한데, a+0은 무엇일까? 결과적으로는 배열 a의 첫 번째 값의 주소이다. a+1은 두 번째 값이고, 다음은 세 번째 값이다. 두 줄의 결과를 비교해보면, 앞의 두 주소 값은 같고, 그 다음부터는 4씩 증가하는 것을 알 수 있다. int형 배열이기 때문에 4바이트씩 증가하는 것이다. 만일 char형 배열을 선언했다면 1바이트씩 증가하는 것을 확인할 수 있을 것이다. 이를 통해 알 수 있는 점은 포인터가 가감 연산이 가능하다는 점이다. 포인터에서의 연산은 그 자료형의 크기에 따라 더해지고 빼지는 정도가 달라진다고 생각하면 된다.

이제 배열의 주소가 어떻게 되는지 확인해 봤으니, 주소를 이용해 값에 접근해보자.



하나씩 자세히 보면, a[0]는 a라는 배열의 첫 번째 요소 값이다. 두 번째로 a는 배열의 이름이자 주소이다. 주소라는 것은 일반적인 변수에서 &를 이용해 구한 것과 같고, 따라서 \*a는 배열의 시작 주소 위치에 저장된 값, 즉 a[0]를 말한다. 세 번째로 우린 앞 장에서 \*와 &연산자가 서로 상쇄된다는 것을 배웠다. 따라서 \*&a[0]은 a[0]를 말해준다. 마지막으로, a+0는 배열 a의 첫 번째 값의 주소를 뜻하므로 결국, 4가지 경우 모두 a[0]와 같은 값을 출력하게 된다.

위 코드를 통해 배열 안에 있는 값 하나에만 접근해 보았다. 그렇다면 배열의 모든 값을 출력하고 싶을 때는 어떻게 해야 할까? 다음 코드를 보면서 함께 알아가 보자.



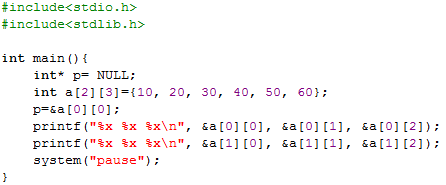
for문을 돌면서 출력이 되는데, a[i]는 값 자체를 접근하는 것이고, \*&는 서로 상쇄되기 때문에 \*&a[i]를 이해하는 것도 어렵진 않을 것이다. 그럼 마지막 \*(a+i)에 대해 한 번 알아보자. 위해서도 잠깐 봤듯이 a는 배열의 주소이고, 주소의 증가를 통해서 다음 주소로 넘어갈 수 있다. a+0은 a의 주소에서 아무것도 증가시키지 않은 것, a+1는 a 다음 다음의 주소를 가리키게 된다. 즉, \*(a+i)는 배열 a에서 (i+1)번째 값을 출력해준다.

이렇게 위 코드들을 통해 \*(a+i)==\*&a[i]==a[i]인 것을 확인해볼 수 있었다. 여기에서 한 가지 더 재미있는 사실을 알려주려고 한다. 위의 코드에서 a[i] 대신에 i[a]를 입력해보라! 이게 무슨 소리냐 하겠지만, 컴파일을 해 보면 너무도 당당하게 정상적인 출력 화면이 떠서 당황할 수도 있겠다. 사실 C언어에서 a[i], i[a] 모두 \*(a+i), \*(i+a)로 인식을 하기 때문이다! 컴파일러는 [ ] 이런게 오면 모두 저렇게 괄호로 묶고 포인터로 생각하기 때문에 이런 재밌는 일이 생긴다. 이러한 사실을 잘 활용하여, 배열에 인덱스로 접근을 하거나 포인터로 직접 접근을 하는 것은 독자 여러분들에 취향에 맡기면 될 것 같다.

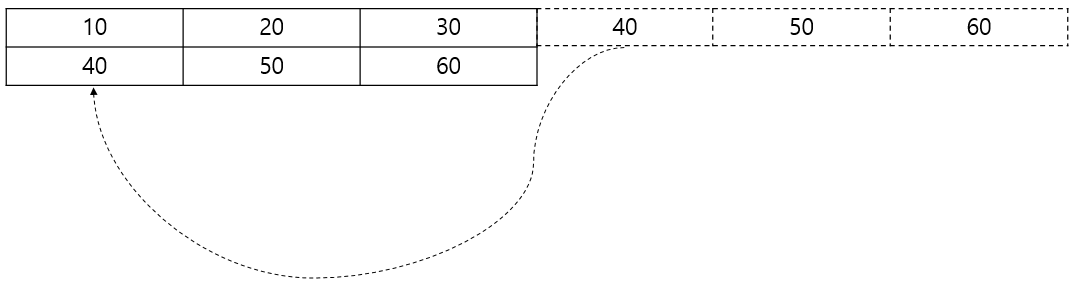
이렇게 위의 내용을 배운 독자들은 아마도 배열과 포인터를 구분하지 않게 될 수도 있다. 그렇지만 여기에서 유의해야 할 점이 한 가지 있다. 배열은 그 시작 주소를 변경 할 수 없으나(배열 이름이 시작 주소가 되므로) 순수한 포인터는 주소를 아주 자유롭게 변경 할 수 있다는 점이다. 물론 이러한 성질에 대한 차이점도 뒤의 const 단원을 학습하면 더 잘 이해할 수 있을 것이다.

8-1-2 이차원 배열로의 확장

이번엔 일차원 배열에서 나아가 이차원 배열에서의 포인터의 사용에 대해 알아보자. 먼저, 이차원 배열로 어떻게 변수들의 주소가 결정되는지 다음 코드를 통해 학습해보자.

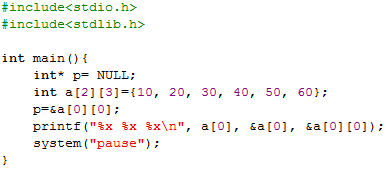


|  |  |  |
| --- | --- | --- |
| 10 | 20 | 30 |
| 40 | 50 | 60 |



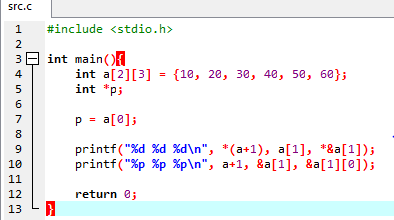
그림으로는 2x3 행렬로 표시할 수 있다. 한 행에서 다른 행으로 넘어갈 때 주소 변화는 어떨까? &a[0][2]와 &a[1][0]를 비교해보니 4바이트 차이가 난다. 즉, 한 행에서 다음 행으로 넘어갈 때도 주소가 그대로 이어진다는 것이다. 이 말을 쉽게 풀어내자면, 이차원 배열이란 사실 길쭉한 일차원 배열을 몇 개씩 단위로 끊어 놓은 것과 같다. 그래서 위의 소스 코드에서 a[0][3] 도 접근이 되는데, 주소 값을 직접 출력해 보면 a[1][0] 과 같다는 사실을 알 수 있을 것이다.

이차원 배열에서도 역시 \*(arr+i)==arr[i]==\*&arr[i] 관계가 성립한다. 그런데 일차원 배열과는 의미하는 바가 좀 다르다. 일차원 배열에서는 저 식이 배열의 값을 가리켰다면, 이차원 배열에서는 주소 값을 가리킨다. 무슨 말인지 잘 이해가 안 될 것 같아서 밑의 코드를 준비해봤다.



차례대로 보면, a[0]는 1행의 주소를 의미한다. 이차원 배열에서 행의 개수만큼 배열이 있다는 것은 ‘문자와 문자열’ 단원에서 배웠을 것이다. 다시 말하면 각 행이 일차원 배열이라는 것이다. 따라서 a[0]는 1행 배열의 이름이자 주소이다. 이차원 배열을 초기화할 때는 행부터 먼저 채우므로 행의 시작을 알리는 주소로 &a[0]이 이용된다. &a[0][0]은 맨 처음 값의 주소를 나타낸다. 처음 값의 주소가 1행의 주소이므로 세 개가 모두 같은 값을 가진다.

따라서 1차원 배열의 형태를 가진 \*(a+i)==a[i]==\*&a[i] 의 관계는 주소 값을 출력하게 된다. 심심하면 한 번 출력해보자.



바로 앞에서 포인터는 가감 연산이 가능하다고 했던 것을 기억하는 독자들은 이 소스코드를 보고 조금 헷갈리는 부분이 있을 수도 있다. a == &a[0][0] 이니까 a+1과 &a[0][1]이 같아야 하지 않겠냐 라고 생각하며 위의 소스코드에 문제가 있는 것 같다고 느끼는 사람들도 있을 테지만, 포인터의 가감 연산의 기본은 “단위”별로 더하고, 빼진다는 것을 염두에 두어야만 한다. 일반적인 배열에서 char문자 하나를 가감할 경우 1바이트, int의 경우 4바이트씩 더하고 빼지겠지만 배열 포인터의 경우 일차원 배열 하나를 한 덩어리로 보기 때문에, 더하고 빼는 것을 배열 한 덩어리로 생각해야만 한다는 특징이 있다. 그래서 위의 가감연산의 경우 \*(a+1) == a[1]이 성립하게 된다. 잘 안다고 생각하더라도 포인터를 다룰 때에는 어떤 작은 실수를 할 지 모르니 항상 주의하도록 하자.

만약 값을 출력 하고 싶으면 위의 식 대신 다음 식을 사용해주면 된다.

#include<stdio.h>

#include<stdlib.h>

int main()

{

int\* p = NULL;

int i, j;

int a[2][2] = {10,20,30,40};

p=a[0];

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

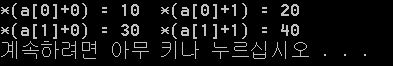
printf("\*(a[%d]+%d) = %d ",i,j, \*(a[i]+j));

}

printf("\n");

}

}



#include<stdio.h>

#include<stdlib.h>

int main()

{

int\* p = NULL;

int i, j;

int a[2][2] = {10,20,30,40};

p=a[0];

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

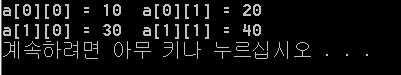
printf("a[%d][%d] = %d ",i,j, a[i][j]);

}

printf("\n");

}

}



#include<stdio.h>

#include<stdlib.h>

int main()

{

int\* p = NULL;

int i, j;

int a[2][2] = {10,20,30,40};

p=a[0];

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

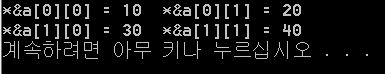
printf("\*&a[%d][%d] = %d ",i,j, \*&a[i][j]);

}

printf("\n");

}

}



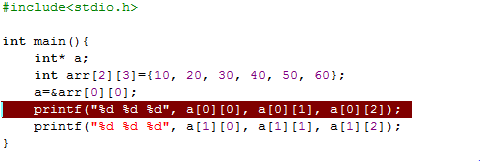
8-1-3 배열 포인터

편지 배달부 아저씨는 한 집 말고도 여러 집에 편지를 배달해 준다. 배달하는 집들의 주소를 잘 보면 번지수가 연속적인 것을 알 수 있다. 예를 들어, 광성로 라고 시작하는 도로명 주소가 있다고 하자. 그 아래에는 ()-() 번지로 표기되는 많은 집들이 있다. 즉, 광성로 라는 배열 안에 있는 여러 **집 주소들**이 있는 것이다.

지금부터 배울 배열 포인터 변수란 말 그대로 배열을 가리키는 포인터 변수이다. 그런데 배열 포인터 변수가 왜 필요할까? 방금 전에 우린 이차원 배열의 포인터 사용을 알아봤는데 말이다.

방금 전 코드를 보면 일차원 배열에서 사용하는 것처럼 2차원 배열의 값들에 접근하는 것을 알 수 있다. 그렇다면 2차원 배열의 형태를 그대로 유지하면서 쓰는 방법은 없을까? 한번 다음 코드를 실행시켜보자.

(코드. 이차원 배열의 주소를 한 포인터 변수에 넣고, 2차원 배열의 접근 형태로 접근하는 코드)



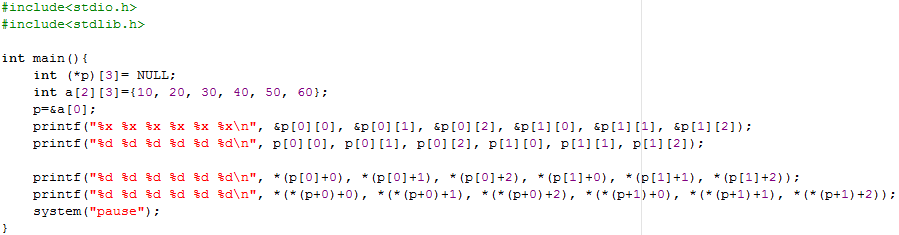
먼저, 이차원 배열의 주소를 포인터 변수 a 안에 넣고, 그대로 2차원 배열을 접근해 보았다. 결과적으로는 출력하는 줄에서 에러가 난다. 에러가 뜨는 이유는 1차원 포인터 변수로 선언한 a를 2차원 배열을 사용하는 것 같이 사용했기 때문이다. 결론부터 말하자면 배열 포인터 변수는 이차원 이상의 배열을 포인터로 쉽게 다루기 위해 사용되는 것이다. 그래서 이렇게 2차원 배열의 경우 일반 포인터 변수가 아닌, 배열 포인터 변수라는 것을 이용해야 한다.

배열 포인터 변수의 선언 방법은 아래와 같다.

자료형 (\*배열포인터 변수 이름) [배열 크기];

int (\*p) [3];

문법이니 외우도록 하자~

 선언을 해봤으니 코드를 통해 어떻게 사용되는지 알아보자.

p에 배열 포인터를 이용하여 이차원 배열 a의 주소를 넣었다. 첫 번째 줄이 주소를 두 번째 줄이 값을 출력한다는 것은 쉽게 알 수 있을 것이다. 세 번째에서 p[0]는 1행의 주소를 나타내고 +0은 첫 번째 주소를 뜻한다. 앞에 \*가 있으니 결과적으로 p[0][0]을 출력해준다. 마지막 줄에서 p는 이차원 배열의 주소이다. 거기서 +0을 했으니 \*(p+0)은 1행의 주소를 가리킨다. 즉, p[0]==\*(p+0)인 것이다. 결과적으로 printf 구문의 두 번째 줄부터 네 번째 줄까지는 같은 내용을 출력한다고 보면 된다!

8-1-4 포인터 배열

포인터 배열은 주소를 저장하는 포인터 변수 여러 개를 한꺼번에 쓰고 싶을 때 이용된다. 배열인데, 포인터 변수들이 나열되어있는 배열이다. 우리가 처음 변수와 배열을 배웠을 때와 마찬가지로 많은 포인터 변수들을 선언해야 할 때 포인터 배열이 사용된다. 포인터 배열의 선언 방법은 다음과 같다.

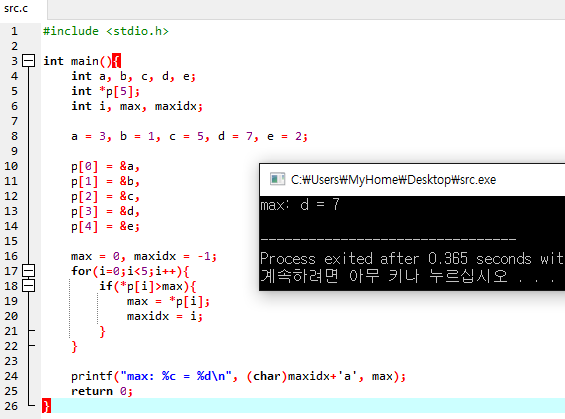
데이터 형 \*변수 이름 [배열 크기]

int \*pointer[3];

배열 포인터와 포인터 배열은 개념 구분을 명확히 하지 않으면 헷갈리기 쉽다. 지금 배우는 포인터 배열은 주소를 저장하는 변수를 많이 만들 때 쓰는 것으로 말 그대로 **배열**이다. 바로 앞에서 배운 배열 포인터 변수는 배열을 가리키는 **포인터 변수**이다. 헷갈리지 않도록 정의와 차이점을 잘 기억해 놓도록 하자.

포인터 배열 : int \*p[5];

배열 포인터 변수 : int (\*p)[5];



포인터 배열의 경우 도대체 어디에서 쓸까 싶다. 첫 번째로는 위의 소스 코드처럼 여러 가지 다른 변수들의 주소를 한 번에 모아 저장하고 싶을 때 쓴다. 쉽게 풀어서 얘기하자면

int a, b, c, d, e;

이렇게 개별적인 변수들을 반복문과 같은 곳에서 연속적으로 사용하고 싶을 때, 어떻게 사용할 지 막막한 경우다 많다. 이런 경우에 위의 소스 코드처럼 포인터 배열에다 a~e 변수의 주소를 각각 담아주고, 이 배열을 이용해 각각의 변수에 접근하면 개별적인 변수를 연속적으로 이용하는 것이 가능하다.

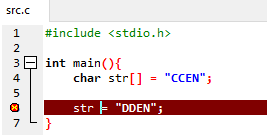
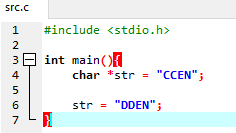
포인터 배열은 이외에도 유용하게 사용되는 경우가 많은데, 이는 뒤의 8-3-1의 구조체 포인터 배열 단원에서 다루기로 한다.

8-2 문자열과 포인터

8-2-1 문자열의 종류

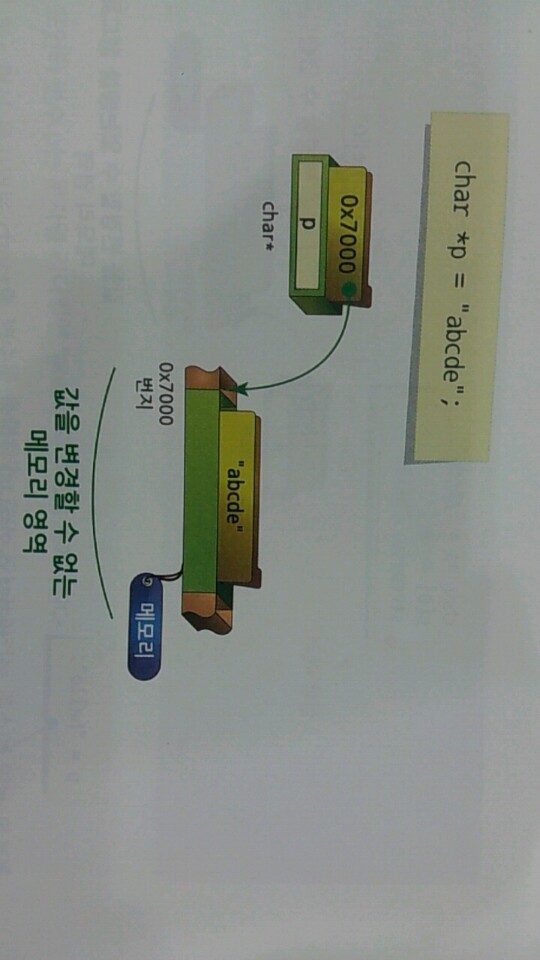
앞의 5장에서 우리는 배열에 각각의 문자들이 들어가고 마지막에 NULL 문자가 들어가는 기초적인 배열과 같은 문자열에 대해 알아보았다. 하지만 C언어에서 다루는 모든 문자열이 그런 형태를 띠는 것은 아니다. 그럼 과연 어떤 문자열이 또 있다는 것일까?

문자열 상수 (혹은 문자열 리터럴) 이라는 것이 있다. 문자열을 하나의 상수처럼 이용하는 것이다.

배열로 구성된 문자열의 경우 처음 char 배열 변수를 선언하면서 초기화 하는 경우는 문제가 없지만, 이후에 문자열을 대입하는 경우 컴파일 에러가 발생한다. 그렇지만 char\* 형으로 선언된 문자열 상수의 경우, 이후에 다시 문자열을 대입하더라도 문제 없이 잘 컴파일이 된다. 앞에서 배열도 포인터처럼 사용할 수 있다고 했고, 그래서 배열과 포인터의 구분이 없다고 생각하게 된 독자들도 있을 것이다. 물론 그 말은 맞다. 그렇지만 5장에서도 말했듯 C에서 문자열이란 매우 특별한 존재이다.

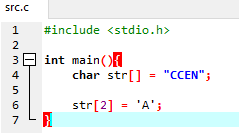
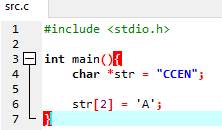
다시 위에서 언급했던 말을 되돌아보면, 문자열 자체를 하나의 상수로 보는 것을 문자열 상수라고 하는데 ‘문자열 상수는 말 그대로 상수이다’라고 생각해 주면 될 것 같다. 일반적인 상수 ‘a’, 3.14, 5 와 같은 상수들과 유사하게 이용되는 것이다. 다만 차이점이 있다면, 이러한 일반적인 상수들의 경우 cpu의 레지스터에 사용할 때마다 일시적으로 값을 저장해두고 쓰지만 문자열 상수의 경우 길이가(크기가) 고정되어있지 않으므로 다른 메모리 공간에 넣어두고 사용한다고 보면 된다. ‘다른 메모리 공간에 넣어둔다’라는 말은, 값을 변경할 수 없는 메모리 영역에 문자열 상수를 저장한다는 말이 되는데, 이는 뒤의 const 키워드를 알아보는 단원에서 더 자세히 다루도록 한다.



문자열 상수는 이렇게 고유의 주소 값을 가지게 되고, char\* 형 변수에는 이 문자열 상수의 주소 값만 저장하는 것이므로 위에서 다루었던 소스 코드처럼 다른 ‘문자열 상수’를 대입하는 것이 가능하다. 겉으로는 그냥 문자열을 대입하는 것으로 보이지만, 다른 ‘문자열 상수’의 주소를 대입하는 것이기에 배열로 다루는 문자열과는 큰 차이점이 있다.

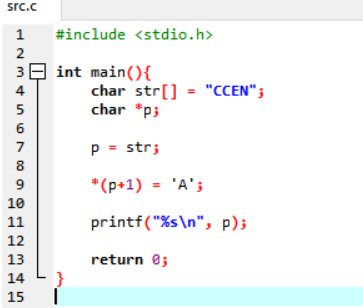
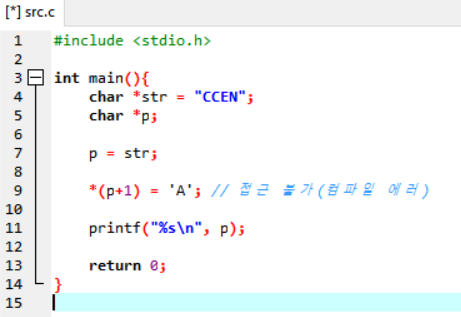
8-2-2 문자열의 종류에 따른 포인터 이용하기

문자열 상수에 대해 간략하게 알아보았는데, 이번에는 이 문자열 상수에 대해 조금 더 자세히 알아 보도록 한다.

위의 소스코드를 보면 문자열 중간에 어떤 값을 바꾸고 싶을 때, 배열로 사용되는 문자열의 경우 문제 없이 잘 변경이 되고 실행이 되지만, 문자열 상수의 경우 컴파일 에러는 나지 않지만 실행 에러(검은 콘솔 화면에서 오류가 발생하는 에러)가 발생하는 것을 알 수 있다. 이는 위간에 부분적으로 수정하는 것이 불가능하다.

지금까지의 내용을 잘 따라왔다면 문자열 상수에 대한 전반적인 이해를 한 것이다. 이번에는 문자열 포인터에 대해 알아볼 것인데, 문자열 포인터의 경우 두 가지 방식으로 사용될 수가 있다.

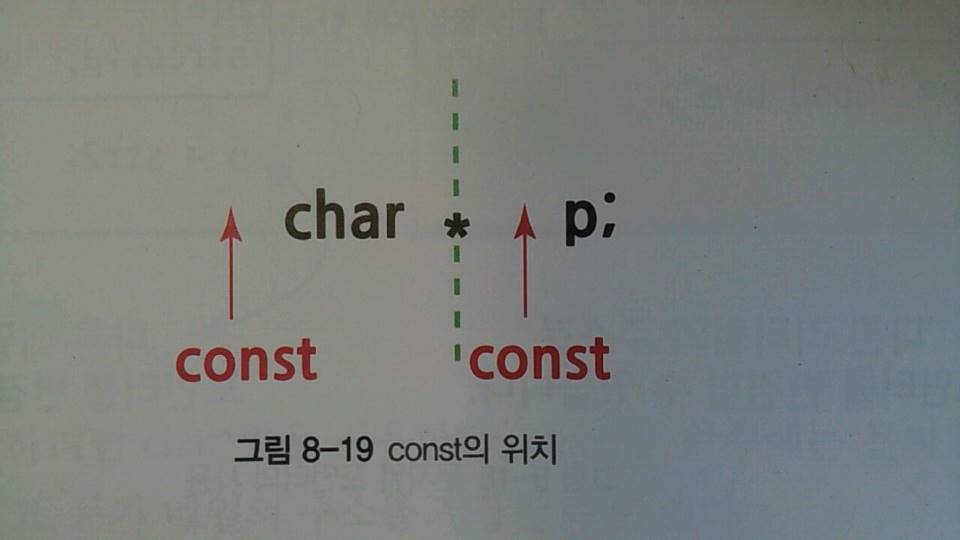
 

첫 번째 소스 코드처럼 배열로 된 문자열에 대해 그 문자 배열의 이름을 char \* 형 포인터 변수에 대입하는 경우(배열의 이름은 그 배열의 시작 주소와 같다는 사실을 이용한다) 배열에서 포인터를 이용한 접근과 마찬가지로 작동한다. 그리고 두 번째 소스 코드처럼 문자열 상수를 저장하는 포인터의 경우는 이번 장에서 계속 언급하고 있는 문자열 상수와 같다.

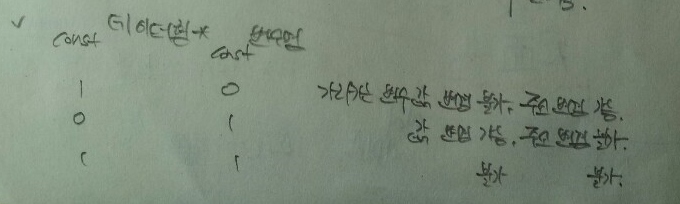
char\* 형 변수 즉 문자 포인터를 다루는 경우에, 저장하는 값이 문자 배열이냐 문자열 상수이냐에 따라 사용 양태가 매우 달라지게 된다는 사실을 독자 여러분들이 잘 숙지하고 기억해주었으면 좋겠다. 많은 사람들이 혼동하는 내용이므로 더욱 꼼꼼하게 잘 익히도록 하자.

8-2-3 const 키워드에 대한 정성적 이해

위의 문자열 상수는 const char\* 형과 같다고 볼 수 있다. 문자열 상수는 문자열 중간의 값을 변경할 수 없다는 특징을 가지고 있었다. 이렇게 const를 앞에 붙이면 그 포인터가 가리키는 변수의 값을 변경할 수 없게 된다. 또한 앞에서 잠깐 언급했었지만 배열의 경우 포인터처럼 접근을 할 수 있지만 주소를 변경 할 수 없다는 특징을 가지고 있었다. 이는 굳이 따지자면 char \* const 형(char 부분에는 배열의 자료형에 따라 달라질 수 있음)이라고 볼 수 있다. 이게 대체 무슨 황당무개한 소리인가? 하며 답답해하는 독자들도 있을 것이다. 다음 그림을 통해 하나씩 살펴보자.



const의 위치는 위의 그림과 같이 두 군데 각각 들어갈 수 있고, 모두 들어갈 수도 있다. 각각의 경우에 따라 포인터의 속성이 바뀌는데, 각 속성은 아래의 그림과 같다.



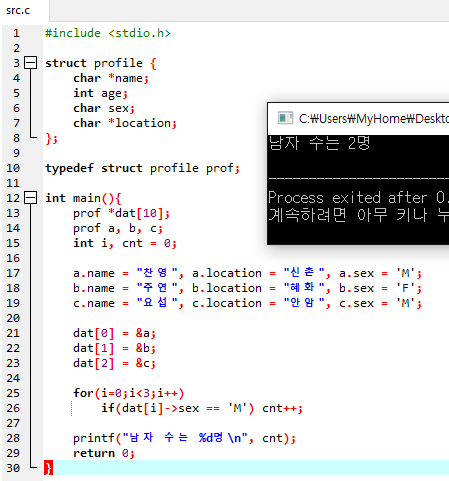
위의 그림처럼 구분을 하면 된다. 별로 쓰일 것 같지 않은 복잡한 기능이지만, 이렇게 const를 잘 지정해주면 불필요한 값의 변경이나 주소의 변경을 막을 수 있다. 이는 고급 프로그래밍에서 유용하게 사용되므로 잘 기억해두기를 바란다.

8-3 구조체와 포인터

이번 장에서는 6장에서 배웠던 구조체를 포인터를 이용하여 어떻게 다룰 수 있는지 알아볼 것이다. 구조체와 포인터가 만나면 엄청나게 많은 것들을 해낼 수가 있으므로, 독자 여러분들은 조금만 더 힘을 내서 이번 포인터 장의 마지막까지 완독하기를 바란다.

8-3-1 포인터 배열의 확장: 구조체 이용하기

앞의 8-1-4에서 다루었던 포인터 배열을 좀 더 활용하는 것을 이번 단원에서 다루게 될 것이다. 구조체가 포인터를 만났을 때 엄청난 시너지를 가져오게 되는데, 우선은 다음 소스 코드를 살펴보자.

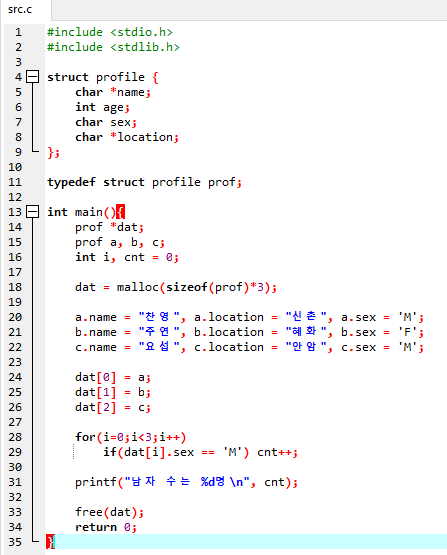


“어? 구조체 이름 옆에 “->” 이 화살표는 뭐지?” 하는 독자들이 있을 것이다. 구조체 포인터에서는 구조체에서 “. “ 로 멤버 접근을 했던 것과는 달리 “->” 연산자로 접근을 한다.

참고로, dat[i]->sex는 (\*dat[i]).sex 로도 사용될 수 있다.

8-3-2 구조체 포인터 배열의 동적 할당

바로 앞 8-3-1에서 다룬 내용만 보고는 어떤 독자들은 어? 구조체 포인터 배열 이게 왜 그렇게 유용하다는 것이지? 라고 생각할 수도 있다. 아직 끝나지 않았다. 이번 장에서 앞에서 다루었던 동적 할당을 구조체 포인터 배열에 잘 활용하면 C언어에서 가장 유용하게 쓰이는 최고의 사용자 정의 자료형이 탄생하게 된다.

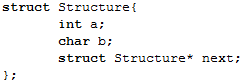


조금 전에 보았던 예제와 유사하지만, dat에 동적 할당을 하여 일반 구조체 배열로 만들었으므로 dat 배열에는 a, b, c를 대입만하고, 또 접근도 ->이 아닌 .으로 한 것을 알 수 있다. 처음에는 조금 어렵겠지만 이를 독자 여러분이 잘 숙지한다면, 앞으로의 코딩에서 정말 유연하게 자료들을 다룰 수 있는 능력을 얻게 될 것이다.

8-3-3 구조체끼리 연결하기 (Linked list)

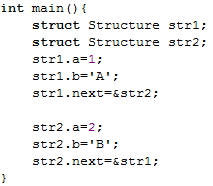
이번 장은 길고 길었던 포인터 여정의 끝을 (사실은 끝이 아니다! 11장을 기대하시라) 보는 장이라고 보면 된다. 사실 C언어 수준에서 몰라도 되는 내용인데, 좀 심화적인 내용이라고 생각하고 이 과목에서는 편하게 보면 될 것 같다.

구조체를 연결할 수가 있다. 이게 과연 무슨 말일까? 구조체 안에는 다양한 타입의 변수들이 들어갈 수 있고, 심지어는 구조체가 구조체 안에 들어갈 수도 있다는 사실을 기억할 것이다. 동일하게 포인터 변수 역시 변수의 일종이므로 구조체 안에 들어갈 수 있다. 그러면, 구조체를 가리키는 포인터 변수가 구조체 안에 들어가 있다면 어떻게 될까? 한 구조체가 다른 구조체를 가리키는 꼴이 되니 결과적으로 구조체들끼리 연결고리가 생기게 된다. 말로만 하면 어려우니 다음 코드를 보자.



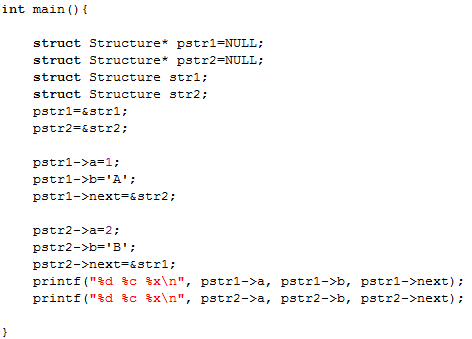
처음에 구조체를 선언한다. 여기서 int형 변수, char형 변수, 그리고 마지막으로 다음 구조체를 **가리킬** 구조체 포인터 변수 next를 선언해준다. 다음으로 main함수에서 구조체 변수들인 str1, str2를 선언해주자.

(이거를 이용할 수 있는 그림 넣어주기)



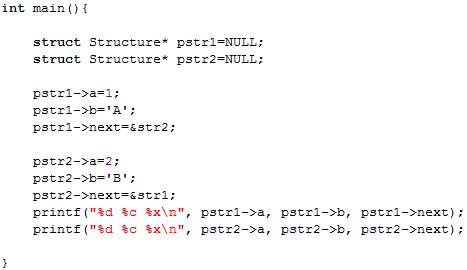
구조체를 접근할 때 ‘.’ 연산자를 사용한다는 것을 다 기억하고 있을 것이다. 그래서 구조체 str1에 a에는 1이, b에는 문자 A가, 그리고 가리킬 다음 구조체의 **주소**로 str2의 주소를 넣어주었다. str2도 마찬가지로 볼 수 있다. 다음으로 str2의 next를 이용해서 str1의 a값을 바꿔 보았다. 이를 통해 구조체들끼리 서로 연결할 수 있고, 연결되어있는 구조체끼리는 값에도 접근할 수 있다는 것을 알았다.

이렇게 직접적으로 값을 넣고, 바꿔주는 방법도 있지만, 포인터를 사용해서 구조체의 주소를 통해 접근할 수도 있다. 앞에서도 언급했듯 구조체 포인터에선 . 연산자 대신에 ‘->’ 연산자를 사용한다. 포인터가 가리키는 의미라는 것을 생각하며 기억하면 될 것이다. -> 연산자는 아래 코드와 같이 사용한다

. 

차근차근 보면, 먼저 구조체의 주소를 저장할 포인터 변수 두개를 선언하고, 이제 실제로 다룰 구조체 str1과 str2를 선언해준다. 다음으로 각 구조체의 주소를 구조체 포인터 변수에 넣어서 주소를 사용할 수 있게 만들어 놓는다. 그 다음부턴 연산자 . 이 ->로 바뀌었을 뿐, 앞의 코드와 다르지 않다.

혹시나 해서 보는 거지만, str1과 str2를 왜 선언하는지 이해가 안되는 독자들을 위해 다음 코드를 작성해 보았다.



결과적으로는 에러가 뜨는 것을 알 수 있다. 포인터 변수를 선언했는데 왜 에러가 뜨는 것일까? 다시 말하지만, 포인터 변수는 주소를 담는 변수이다. 지금 상황에선 주소를 담을 상자만 만들어 놓은 것이지, 실제로 재료를 넣을 상자는 만들지 않은 것이다. 더군다나 포인터 변수 pstr1과 pstr2에는 어떠한 주소도 넣지 않았다. 포인터 변수에 어떤 값의 주소를 담지 않고 바로 접근을 해 무언가를 하려고 했기 때문에 발생하는 오류이다.

지금까지 7, 8장 무려 두 장에 걸쳐 방대한 내용을 포함하는 포인터 단원을 열심히 공부하였다. 포인터는 C언어의 꽃이라고 불리울 만큼 가장 값진 보석과도 같은 존재이다. 포인터가 엄청 어렵고, 한번 학습한 것만으로는 완전한 이해가 어려웠을 수도 있다. 이해한다. 여기 있는 모두가 그렇고, 이 책의 집필진 모두가 그래왔기 때문이다. 가장 어려운 문이지만, 이 문턱을 잘 넘는다면 독자 여러분은 이미 한 차원 수준 높은 프로그래머가 된 것이므로 축하한다.

C언어의 2/3 정도를 학습하였고, 앞으로 우리에게 남은 과제는 바로 C언어, 혹은 프로그래밍 언어의 핵심이라고도 볼 수 있는 함수에 대해 학습할 것이다. 포인터는 두 장이었다. 함수는 무려 세 장에 걸친 더욱 방대한 내용을 다룰 것이다. 독자 여러분은 심호흡 한 번 크게 하고, 한 잔의 커피를 마시는 여유를 가진 후에 다음 장을 펼치며 또 다른 여정을 잘 해내기를 바란다.