TI DSP,MCU및 Xilinux Zynq FPGA 프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/2/23
수업일수	3일차
담당강사	Innova Lee(이상훈)
이메일	gcccompil3r@gmail.com

목차

1. 재귀함수를 while문으로 풀어쓰기

예제1)

```
#include<stdio.h>
int fib(int num)
{
	int first = 1;
	int second = 1;
	int tmp = 0;
	if(num == 1 || num == 2)
		return 1;

	while(num-- > 2)
	{
		tmp = first + second;
		first = second;
		second = tmp;
	}
return tmp;
}

int main(void)
{
	int result,final_val;
	printf("피보나치 수열의 항의 개수를 입력하시오 : ");
	scanf("%d", &final_val);
	result = fib(final_val);
	printf("%d번째 항의 수는 = %d₩n", final_val, result);
	return 0;


}
```

~ 결과

6 입력시 8 출력

final_val의 변수에 입력 값을 입력받아 fib함수로 보낸다. fib함수에서는 num으로 final_val의 값을 받게 된다. 그리고 변수 first,second는 초기값을 1로 지정해주고 tmp는 0으로 지정하게 되는데 num이 1이거나 num2가 2일 때 값을 1로 리턴한다.

num-- > 2는 num에서 1을 뺀 값이 2보다 클 때에는 tmp에 앞의 수와 뒤의 수의 합을 넣고 뒤의 수는 앞에 수에 보관했다 tmp의 값은 뒤의 값에 넣어 각 변수에 들어있는 변수 값을 서로 바꿔준다.

예제2)

```
#include<stdio.h>

int fact(int n)
{

        if(n==0)
                return 1;

        else
                return n*fact(n-1);
}

int main(void)
{
        int result, fact_val;
        printf("계산할 팩토리얼을 입력하시오 : ");
        scanf("%d",&fact_val);
        result=fact(fact_val);
        printf("%d번째 항의 수는 = %d₩n",fact_val,result);

        return 0;

}
```

~ 결과

5 입력시 120 출력

scanf로 fact_val에 입력된 값을 int n의 변수에 입력값을 받아 fact함수 result에 fact의 리턴값을 저장한다. 만일 n이 =일 때에는 1로 리턴하고 그렇지 않을 때에는 이전의n-1의fact값과 n을 곱하여 리턴갑을 출력한다.

예제3)

```
#include<stdio.h>

int fact(int n)
{
        int first=1;
        while(n>0)
                first = first *n--;
        return first;
}

int main(void)
{
        int result,fact_val;
        printf("계산할 팩토리얼을 입력하시오:");
        scanf("%d",&fact_val);
        result=fact(fact_val);
        printf("%d번째항의 수는 = %d₩n",fact_val,result);
        return 0;
}
```

~ 결과
5 입력시 120 출력

2번과 동일하게 출력되지만 소스코드에서 if-else를 사용하지 않았다.
main함수에서 int형 result와 fact_val 변수를 선언하고 fact_val에 값을 받는다. result는 fact_val를 fact함수의 인자로 보내 리턴된 값을 저장한다.
fact는 fact_val를 받아 int n에 값을 저장하고 first라는 변수에 1이라는 초기값을 넣어 n이 0보다 큰 동안 n에서 1을 뺀 값과 first를 곱한 값을 first에 저장하게 된다. n이 0이 되는 순간 while문을 벗어나게 되고 저장된 first의 값이 리턴되어 main함수도 돌아오고 printf로 값을 불러낸다.

2.배열

-배열의 사용이유
많은 개수의 변수를 만들 때 사용하는데 for문과 결합하여 사용할 수 있다.
배열 선언하는 법 : int 배열이름 [길이] = {0};
[길이]는 변수를 사용할 개수이므로 배열의 시작은 0이다.
배열은 메모리 상에 순차적으로 배열되어있다. 그러므로 2차원, 3차원 배열이라는 것은 이중배열 삼중배열이라고 생각해야한다.

예제1)

```
#include<stdio.h>

int main(void)
{
	int i;
	int num[7];

	for
		(i=0;i<7;i++)
		{
			num[i]=i;
			printf("num[%d]=%d₩n",i,num[i]);
		}
	return 0;
}
```

~결과

```
num[0]=0
num[1]=1
num[2]=2
num[3]=3
num[4]=4
num[5]=5
num[6]=6
```

위의 예제와 같이 변수 여러개에 값을 넣기 위해 배열을 사용하게 되는데 for문을 사용하여 i값이 0~6인 동안 num[]과 초기값에 각 i를 넣고 printf로 출력한다.

예제2)

```
#include<stdio.h>

int main(void)
{
	int i;
	int num1_arr[]={1,2,3,4,5};
	int num2_arr[3]={1,2,3};

	int len1=sizeof(num1_arr)/sizeof(int);
	int len2=sizeof(num2_arr)/sizeof(int);

	printf("num1_arr lenght=%d₩n",len1);
	printf("num2_arr lenght=%d₩n",len2);

	for(i=0;i<len1;i++)
	{
		printf("num1_arr[%d}=%d₩n",i,num1_arr[i]);
	}


	for(i=0;i<len2;i++)
	{
		printf("num2_arr[%d}=%d₩n",i,num2_arr[i]);
	}

	return 0;
}
```

~ 결과
num1_arr lenght=5
num2_arr lenght=3
num1_arr[0}=1
num1_arr[1}=2
num1_arr[2}=3
num1_arr[3}=4
num1_arr[4}=5
num2_arr[0}=1
num2_arr[1}=2
num2_arr[2}=3

위의 소스코드는 num1_arr[]과 num2_arr[3]의 초기값을 넣고 사이즈를 출력하는 함수인데 배열에[ ]의 빈공간에는 초기화 한 갯수의 값이 들어가게 된다. num2_arr[3]같은 경우는 길이를 미리 설정해 주어 초기화 할 때 3개까지의 값을 넣을 수 있다.
각 배열의 값을 보기 위해 변수 len1과 len2에 각 배열의 sizeof에서 sizeof(int)를 나누어 문자열의 길이를 출력했다. 밑에 2개의 for문은 i와 위에서 구했던 문자열의 길이-1동안 각 배열에 저장된 요소들을 출력하는 함수이다.

예제3)

```
#include<stdio.h>

int main(void)
{

	int i;
	int num1_arr[7]={1,2,3};
	for(i=0;i<7;i++)
	{
		printf(" num1_arr[%d]=%d₩n",i,num1_arr[i]);
	}

	return 0;
}
```

~ 결과

```
 num1_arr[0]=1
 num1_arr[1]=2
 num1_arr[2]=3
 num1_arr[3]=0
 num1_arr[4]=0
 num1_arr[5]=0
 num1_arr[6]=0
```

초기값을 순서대로 주고 남은 배열에 초기값을 주지 않으면 자동으로 0이 입력되는 것을 for문을 통해 출력했다.

-char형 배열

char str[32]="문장";
으로 사용 가능하다. 문장의 문자 하나씩 쪼개져서 배열에 저장하게 된다.
문자열은 Null character(널문자)가 필요한데 원래같으면 '₩0'를 문자열 뒤에 넣어주어야 하지만 요즘에는 컴파일러가 좋아져서 쓰지 않아도 된다.

예제4)

```
#include<stdio.h>

int main(void)

{
        char str1[5]="AAA";
        char str2[]="BBB";

        char str3[]={'A','B','C'};
        char str4[]={'A','B','C','₩0'};

        printf("str1=%s₩n",str1);
        printf("str2=%s₩n",str2);
        printf("str3=%s₩n",str3);
        printf("str4=%s₩n",str4);

        str1[1]='E';
        str2[1]='H';

        printf("str1=%s₩n",str1);
        printf("str2=%s₩n",str2);
        return 0;
}
```

~ 결과

```
str1=AAA
str2=BBB
str3=ABC
str4=ABC
str1=AEA
str2=BHB
```

이전에는 str3은 이상한 문자도 뒤에 출력될텐데 컴파일러가 발전하여 ABC로 출력된다.

-다중 배열

C언어에서 배열에 차원개념은 존재하지 않는다. 메모리에 일직선으로 저장되기 때문이다.
 이중배열은 주로 행렬을 사용하기 위해 이용한다. [x],[y]로 x명의 y개의 과목을 관리할 수 있고 삼중배열은 위 경우에서 반이 z개 있을 때 사용 가능하다.

다중배열의 초기화는 arr[2][3]= {
 {{1,2,3},{1,2,3} } ,
 {{1,2,3},{1,2,3} }
 };
 로 { }안에 { }를 넣는 형식이다.

예제5)

```
#include<stdio.h>

int main(void)
{
	int arr[4][4];
	int i,j;

	for(i=0;i<4;i++)
	{

		for(j=0;j<4;j++)
		{
			if(i==j)
				arr[i][j]=1;
			else
				arr[i][j]=0;
		}
	}
	for(i=0;i<4;i++)
	{
		for(j=0;j<4;j++)
		{
			printf("%d",arr[i][j]);
		}
		printf("₩n");
	}
	return 0;
}
```

~결과
1000
0100
0010
0001

결과를 살펴보면 위의 행렬은 단위행렬을 만드는 함수 인 것을 알 수 있다. 위의 for문은 행과 열이 같을 때 1을 리턴하고 이외의 것들은 0을 리턴하여 단위행렬을 만들어내고 밑의 for문은 위의 for문에서 만든 값들을 printf를 사용하여 출력하는 함수이다.

예제6)

```
#include<stdio.h>

int main(void)
{
	int arr[2][2]={{10,20},{30,40}};
	int i,j;

	for(i=0;i<2;i++)
	{
		for(j=0;j<2;j++)
		{
			printf("arr[%d][%d]=%d₩n",i,j,arr[i][j]);
		}
	}
	return 0;
}
```

~ 결과
arr[0][0]=10
arr[0][1]=20
arr[1][0]=30
arr[1][1]=40

위의 함수는 다중배열의 값을 보여주는 것이다. 배열arr에 각 초기값을 설정하고 중첩 for문을 이용하여 값을 출력해냈다. 이중배열을 초기화 할 때에는 { }안에 { }가2개가 들어가게 된다.

-배열과 포인터

예제7)
배열은 주소이다. 배열도 주소를 가지고 접근 할 것인데 그 주소는 바로 배열의 이름이다.

```
#include<stdio.h>

int main(void)
{
	int arr[4]={10,20,30,40};
	int i;

	printf("adress=%p₩n",arr);

	for(i=0;i<4;i++)
	{
		printf("adress=%p, arr[%d]=%d₩n",&arr[i],i,arr[i]);
	}
	return 0;
}
```

~결과

```
adress=0x7fff4275a400, arr[0]=10
adress=0x7fff4275a404, arr[1]=20
adress=0x7fff4275a408, arr[2]=30
adress=0x7fff4275a40c, arr[3]=40
```

위의 결과를 살펴보게 되면 arr[]이 1씩 증가할 때 마다 0, 4, 8, c로 4byte씩 증가하는 것을 알 수 있는데 이는 배열을 선언할 때 int형의 배열을 선언하였기 때문에 각 배열의 한 공간은 4byte이다..

예제8)

```
#include<stdio.h>

int main(void)
{
int arr[3]={1,2,3};
int *p=arr;
int i;

for(i=0;i<3;i++)
        printf("p[%d]=%d₩n",i,p[i]);
return 0;
}
```

~결과

```
p[0]=1
p[1]=2
p[2]=3
```

위의 예제는 배열 arr을 포인터 *p에 저장하는 것 인데 이를 printf로 출력해 보게 되면 *p가 p[i] 의 형식을 따라 배열 arr에 저장된 값들을 출력해낸다.

예제9)

```
#include<stdio.h>

int main(void)
{
int arr[3][4];

printf("arr address=%p₩n",arr);
printf("arr[0] address=%p₩n",arr[0]);
printf("arr[1] address=%p₩n",arr[1]);
printf("arr[2] address=%p₩n",arr[2]);

return 0;
}
```

~결과
arr address=0x7ffe3f6a6ea0
arr[0] address=0x7ffe3f6a6ea0
arr[1] address=0x7ffe3f6a6eb0
arr[2] address=0x7ffe3f6a6ec0

각 배열의 요소를 %p로 뽑게되면 주소값들을 볼 수 있다. 위의 결과를 살펴보면 >16byte씩 증가함을 관찰할 수 있다. arr배열의 주소를 뽑아보면 맨 처음요소인 arr[0]과 같고 처음요소가 대표주소이다.

궁금점 16byte씩 증가하는 이유 int형이니 4byte씩 증가해야하는 것 아닌가?

예제10)

```
#include<stdio.h>

int main(void)
{
int arr[3][4];

printf("arr size=%lu₩n",sizeof(arr));
printf("arr[0] size=%lu₩n",sizeof(arr[0]));
printf("arr[1] size=%lu₩n",sizeof(arr[1]));
printf("arr[2] size=%lu₩n",sizeof(arr[2]));

return 0;
}
```

~결과

```
arr size=48
arr[0] size=16
arr[1] size=16
arr[2] size=16
```

위의 9번 예제와 동일한 배열로 printf하는 것이 %p에서 %lu로 바뀌었을 뿐이다. 사이즈의 값을 알아내기 위한 함수인데 위에서 증가한 16byte와 같이 각 배열의 요소 사이즈가 16이 나온 것을 확인했고 총 배열의 사이즈는 '요소 하나의 배열 사이즈 * 배열의 길이' 한 값이다.

-함수의 인자로 배열을 전달하려면

배열을 통째로 전달 할 수 없으므로 함수 호출시 배열의 주소값을 전달한다. 함수자체에서 index를 생략하고 []만 전달해도 된다.
되도록이면 index를 모두 정확하게 적는것이 좋다. 2차원, 3차원 배열도 모두 동일하다.

예제11)

```
#include<stdio.h>

void arr_print(int arr[])
{
        int i;
        for(i=0;i<3;i++)
                printf("%4d",arr[i]);
        printf("₩n");
}

int main(void)
{
        int arr[3]={3,33,333};
        arr_print(arr);
        return 0;
}
```

~결과

```
   3  33 333
```

함수의 인자로 배열을 전달하기 위해 인자로 사용하는 함수(배열[])의 형식에서 그 값을 출력할 때 각 요소값을 가져오는 형식으로 프로그래밍 되어있다.

예제12)

```
#include<stdio.h>
void add_arr(int *arr)
{
	int i;
	for(i=0;i<3;i++)
	{
		arr[i]+=7;
	}
}

void print_arr(int *arr)
{
	int i;
	for(i=0; i<3;i++)
	{
		printf("arr[%d]=%d₩n",i,arr[i]);
	}

}
int main(void)
{
	int i;
	int arr[3]={1,2,3};
	add_arr(arr);
	print_arr(arr);
	for(i=0; i<3;i++)
	{
		printf("arr[%d]=%d₩n",i,arr[i]);
	}

	return 0;
}
```

~결과
arr[0]=8
arr[1]=9
arr[2]=10
arr[0]=8
arr[1]=9
arr[2]=10

위의 프로그래밍을 살펴보면 main함수에서 변수i와 배열arr을 선언하고 add_arr이라는 함수에서 *arr을 호출한다. 함수add_arr을 살펴보면 for문을 이용하여 각 배열의 요소 값에 +7을 하게된다.
그리고 main함수에서 print_arr이라는 함수에서 또 arr의 배열을 호출하고 print_arr이라는 함수에서는 *arr를 인자로 받게된다. for문을 사용하여 arr의 각 요소를 출력하게 되는데 이를 main함수에서 for문으로 arr의 각 요소값을 출력했을 때와 같은 값이 출력된다. 한마디로 arr의 값은 정확하다.

의문점 void add_arr(int *arr)인데 void는 return값이 없는 것이다. 포인터를 인자로 불러올 때에는 값을 리턴하지 않아도 되는 것?

3.포인터

주소, 주소를 저장할 수 있는 변수이다.
포인터의 크기는 HW가 몇 비트를 지원하느냐에 따르다.

-포인터 선언방법
가르키고 싶은 것의 자료형을 적고 주소값을 저장할 변수명을 적고 변수명 앞에 '*'을 적는다.
int *p;
주황색 부분은 데이터 타입이고 int형의 주소를 저장 가능하다.

예제1)

```
#include<stdio.h>

int main(void)
{
        int *ptr;
        printf("ptr=%p₩n",ptr);
        printf("ptr value=%d₩n",*ptr);

        *ptr=27;

        printf("ptr value=%d₩n",*ptr);
        return 0;
}
```

~결과
ptr=(nil)
Segmentation fault (core dumped)

의문점>
소스코드를 잘 못 입력한 것인지 잘 모르겠는데 위의 Segmentation fault가 나는 이유는 *ptr에 저장된 주소 값이 없기 때문?

-segmentation falt가 나는 이유

우리가 기예러르 보면서 살펴왔던 주소값들은 사실 가짜주소인데 이 주소값은 가상 메모리 주소에 해당하고 운영체제의 Paging 메커니즘을 통해서 실체 물리 메모리의 주소로 변환된다.
가상 메모리는 리눅스의 경우 32비트 버전과 64비트 버전으로 나뉘는데, 32비트 시스템은 2^32=4G의 가상 메모리 공간을 가지고 여기서 1:3으로 1은 커널이 3은 유저가 사용한다.
1은 시스템(HW,CPU,SW등 각 종 주요 지원들)에 관련된 중요한 함수 루틴과 정보들을 관리하게 된다.
3은 사용자들이 사용하는 정보들로 문제가 생겨도 그다지 치명적이지 않은 많은 정보들로 구성된다.
64비트 시스템은 1:1로 2^64에 해당하는 가상메모리를 각각 가진다. 문제는 변수를 초기화하지 않았을 때 쓰레기 값이 입력되는데 그 쓰레기 값은 0XCCCCC...로 구성된다.
32비트 경우에도 1:3의 경계인 0XC0000000을 넘어가게 되고 64비트의 경우에도 1:1경계를 한참 넘어가게 되어 접근하면 안되는 메모리 영역에 접근하였기에 Page Fault가 발생하게 되고 원래는 Interrupt가 발생해서 Kernel이 Page Handler(페이지 제어)가 동작해서 가상 메모리에 대한 Paging 처리를 해주고 실제 물리 메모리를 할당해 주는데 문제는 이것이 User 쪽에서 들어온 요청이므로 Kernel이 강제로 기각해버리면서 Segmentation fault가 발생하는 것이다.
Kernel쪽에서 들어온 요청은 위의 메커니즘에 따라 물리 메모리를 할당해주게 된다.

예제2)

```
#include<stdio.h>

int main(void)
{
        int num=3;
        *(&num)+=30;
        printf("num=%d₩n",num);
        return 0;
}
```

~결과
num=33

‘*(&num)’ 일때 *과 &는 상쇄되게 된다.(접근하고 받아오기)
*와 &은 상쇄되므로 num+=30으로 해석 가능하고 printf로 출력해보게 되면 3+30이어서 33이 출력된다.

4. 포인터와 배열

```
#include<stdio.h>

int main(void)
{
        char str1[33]="pointer it imprtant!";
        char *str2="pointer is important";

        printf("str1=%s₩n",str1);
        printf("str2=%s₩n",str2);
        return 0;
}
```

~결과
str1=pointer it imprtant!
str2=pointer is important

문자열을 출력할 때에는 두가지 방법이 있는데 char형의 배열에 문장을 넣는 방법과 char형의 포인터에 문장을 넣는 방법이 있다. 배열에 문장을 넣는 방법은 1byte의 메모리에 문자 하나하나를 넣어 저장하지만 -포인터로 나타낸 문장은 각 1byte메모리가 가르키는 주소값이 문자열 하나하나를 가르켜 문장을 출력한다.-

의문점 : 배열의 길이는 33개이고, 초기화를 pointer it imprtant!로 하였는데 이 문장은 띄어쓰기를 포함해서 총 20글자이다. 나머지의 빈 공간은 초기화 할 때 0으로 채워지지 않음?

-이중포인터

주소에 주소를 저장한다. *하나당 주소를 의미.

예제3)

```
#include<stdio.h>

int main(void)
{
        int num=3;
        int *p=&num;
        int **pp=&p;

        printf("num=%d₩n",num);

        printf("*p=%d₩n",*p);
        printf("**pp%d₩n",**pp);

        return 0;

}
```

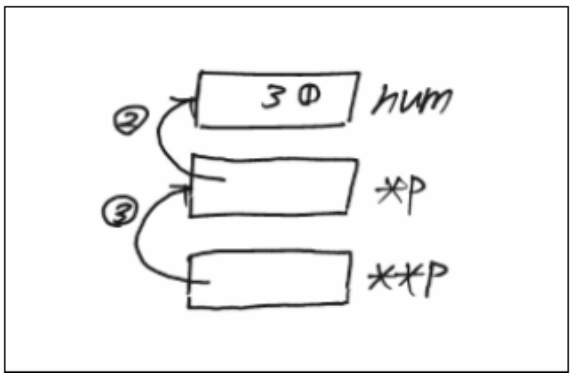
~결과

num=3

*p=3

**pp3

아래의 그림은 위의 소스코드를 그림으로 나타낸 것이다. num은 3의 값이 입력되어있고, *p는 num의 주소값이, **pp는 *p가 가르키는 주소값, 즉 num의 주소값을 가르키게 되어 결국 printf로 출력하게 되는것은 num에 들어있는 3을 출력하게 된다.



-null pointer

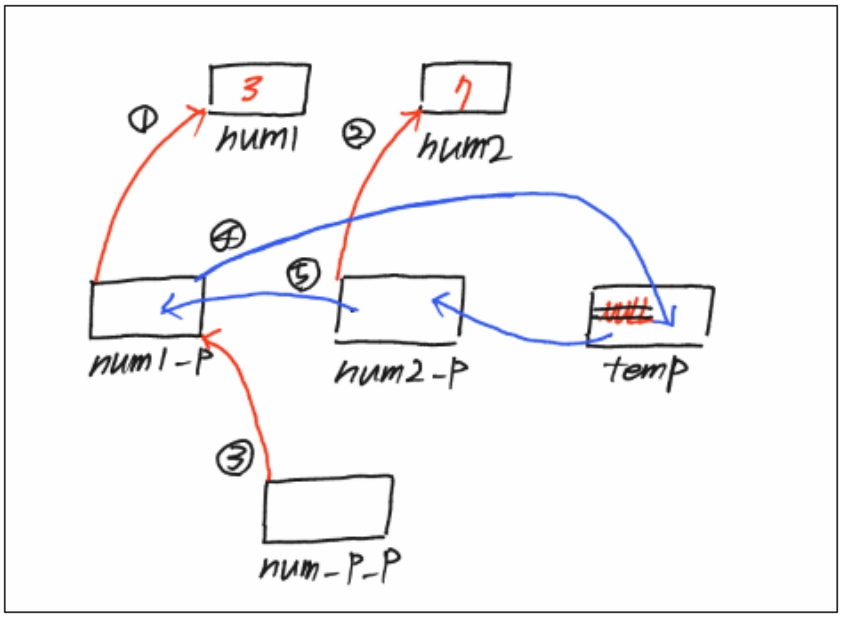
Null pointer란 아무것도 없는 것을 가르키는 포인터이다. 지금 당장 가르킬 초기값이 없을 경우 사용한다. 0XCCCCC...와 같은 쓰레기 값을 가지지 않도록 초기화 하기 위함이다.

예제4)

```
#include<stdio.h>

int main(void)
{

	int num1=3,num2=7;
	int *temp=NULL;
	int *num1_p=&num1;
	int *num2_p=&num2;
	int **num_p_p=&num1_p;

	printf("*num1_p=%d₩n",*num1_p);
	printf("*num2_p=%d₩n",*num2_p);

	temp=*num_p_p;

	*num_p_p=num2_p;
	num2_p=temp;

	printf("*num1_p=%d₩n",*num1_p);
	printf("*num2_p=%d₩n",*num2_p);
	return 0;
}
```

~결과
*num1_p=3
*num2_p=7
*num1_p=7
*num2_p=3

int형의 num1과num2에 3과 7의 값이 입력되고 *temp가 가르키는 주소의 값은 NULL로 아무것도 가르키고 있지 않다. *num1_p와 *num2_p가 가르키는 것은 각각 num1과num2의 주소이고 **num_p_p가 가르키는것은 num1_p의 주소이다.
이를 그림으로 그려 생각해보면 아래와 같은데



빨간색 선은 temp=*num_p_p; 이전의 과정이고 파란색 선은 이후의 과정이다.
printf("*num1_p=%d₩n",*num1_p); → 3
printf("*num2_p=%d₩n",*num2_p); → 7

printf("*num1_p=%d₩n",*num1_p); → 7
printf("*num2_p=%d₩n",*num2_p); → 3

*num1_p, *num2_p, temp는 값의 위치를 바꾸기 위한 과정이다. temp에는 null pointer로 할당된 주소 값이 없는 빈 공간이어서 값의 손실 없이 자리를 바꿀 수 있다.

- 포인터 배열, 배열 포인터

포인터배열 : 포인타(주소)를 저장할 수 있는 '배열'이다.

Ex) int *p[2]

배열포인터 : 배열을 가르키는 '포인터'이다. 배열의 요소값을 가르키고 밑의 예제는 int형 타입의 주소를 2개씩 가르킨다. 총 8 바이트의 크기를 가진다.

Ex) int (*p)[2] = int (*)[2] p

예제1)

```
#include<stdio.h>

int main(void)
{

	int i,j,n1,n2,n3;
	int a[2][2] ={{10,20},{30,40}};
	int *arr_ptr[3]={&n1,&n2,&n3};
	int (*p)[2]=a;

	for(i=0;i<3;i++)
		*arr_ptr[i]=i;

	for(i=0;i<3;i++)
		printf("n%d=%d₩n",i,*arr_ptr[i]);

	for(i=0;i<2;i++)
		printf("p[%d]=%d₩n",i,*p[i]);


	return 0;

}
```

~결과

```
n0=0
n1=1
n2=2
p[0]=10
p[1]=30
```

int형 변수를 i, j, n1, n2, n3를 선언한다. 아직 초기 값을 주지 않아 실제 스택이 쌓이지는 않는다. 그리고 int형 배열을 선언하여 각 스택에 10, 20, 30, 40을 쌓고 *arr_ptr인 포인터 배열은 각 배열의 요소가 n1, n2, n3의 주소를 가르킨다. 배열 포인터 (*p)[2]는 배열 a의 주소를 가르킨다.
첫번째 for문에서 i가 증가함에 따라 *arr_ptr[i]가 가르키는 주소에 각 i값을 넣는다.
두번째 for문에서는 printf로 i와 *arr_ptr[i]가 가르키는 주소에 존재하는 값을 출력하게 되고,
세번째 for문에서는 *p[i]가 가르키는 주소에 존재하는 값을 출력하는데 배열 a가 존재하므로 대표 배열인 10과 30이 *p[0], *p[1]에서 출력하게 된다.

*p[0] → 10
*p[1] → 30

만약 1행 2열과 2행 2열의 값도 표기하고 싶다면
*p[0][1] → 20
*p[1][1] → 40

으로 출력이 가능하다.