일일 업무 사항 정리

작성자	제품팀 이민성 인턴
업무 일시	20221010~20221014

세부 사항

1. 업무 내역 요약 정리

목표 내역	Done & Plan
1 주차) C 언어 개요 2 주차) 연산자/제어문 3 주차) 함수 4 주차) 배열/포인터 5 주차) 구조체/공용체 6 주차) 전처리기	**1. 1 차원 배열** 1-1. 배열의 이해와 배열의 선언 및 초기화 방법 1-2. 배열을 이용한 문자열 변수의 표현 **2. 포인터의 이해** 2-1. 포인터란 무엇인가? 2-2. 포인터와 관련 있는 연산자: &연산자와 *연산자 **3. 포인터와 배열** 3-1. 포인터와 배열의 관계 3-2. 포인터 연산 3-3. 상수 형태의 문자열을 가리키는 포인터 3-4. 포인터 변수로 이뤄진 배열: 포인터 배열 **4. 포인터와 함수에 대한 이해** 4-1. 함수의 인자로 배열 전달하기 4-2. Call-by-value vs Call-by-reference 4-3. 포인터 대상의 const 선언

2. 내용 세부 (업무 세부 내역 정리 및 기타 사항 정리)

1. 1 차원 배열

1. 배열의 이해와 배열의 선언 및 초기화 방법

단순하게 배열은 '둘 이상의 변수를 모아 놓은 것'으로 설명할 수 있다. 하지만 선언방법부터 접근방법까지 일반적인 변수들과는 차이가 있다.

- 배열이란 무엇인가?

다수의 데이터를 저장하고 처리하는 경우에 유용하게 사용할 수 있는 것이 배열이다. 예를 들어서 '좋은 아파트'의 가구별 가족 수를 저장 및 갱신하는 프로그램을 만든다고 가정해보자. '좋은 아파트'는 10 층까지 있고 각 층에는 네 가구씩 살고 있다. 그렇다면 가구별 가족 수의 기록을 위해서는 총 40 개의 변수를 선언해야 하는데, 이를 위해서는 아마도 다음과 같이 코드를 작성해야 할 것이다.

```
Int main(void)
{
        Int floor101, floor102, floor103, floor104; // 1 층 101 호부터 104 호까지
        Int floor201, floor202, floor203, floor204; // 2 층 201 호부터 204 호까지
        Int floor301, floor302, floor303, floor304; // 3 층 301 호부터 304 호까지
        . . . .
}
```

이렇게 코드를 작성한다는 것은 프로그래머에게 있어서 아주 끔찍한 일이다. 바로 이러한 상황에서 배열은 우리가 끔찍한 일을 겪지 않아도 되도록 편의를 제공해준다. 배열을 사용하면 그 수에 상관없이 한번에 많은 수의 변수를 선언할 수 있기 때문이다.

배열은 선언방식에 따라서 1 차원 구조가 될 수도 있고, 2 차원 구조가 될 수도 있다.

- 1 차원 배열의 선언에 필요한 것 3 가지: 배열이름, 자료형, 길이정보

배열은 일반적인 변수와 달리 여러 개의 값을 지정할 수 있다. 여러 개의 변수가 모여서 배열을 이루기 때문이다. 이러한 배열 중에서 1 차원 배열은 다음과 같이 선언한다.

```
int oneDimArr[4];
```

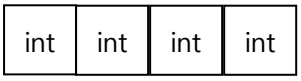
위의 문장을 이루는 요소 셋은 다음과 같다.

- int : 배열을 이루는 요소(변수)의 자료형
- oneDimArr : 배열의 이름
- [4] : 배열의 길이

즉, 위의 배열 선언문이 의미하는 바는 다음과 같다.

"int형 변수 4 개로 이뤄진 배열을 선언하되, 그 배열의 이름은 oneDimArr로 해라"

그리고 위의 문장으로 인해서 선언되는 배열의 형태는 다음과 같다.



위 그림에서 중요한 사실은 int형 변수 4 개가 나란히 선언되어 있다는 점이다. 실제로 배열을 이루는 요소들의 주소 값을 확인해보면 이러한 사실을 확인할 수 있다. 참고로 다양한 자료형과 다양한 길이의 1 차원 배열은 다음과 같이 선언하면 된다.

```
int arr1[7]; // 길이가 7 인 int형 1 차원 배열 arr1
```

float arr2[10]; // 길이가 10 인 float형 1 차원 배열 arr2
double arr3[12]; // 길이가 12 인 double형 1 차원 배열 arr3
이렇듯 배열의 선언방식에는 일정한 규칙이 존재하기 때문에 배열의 선언은 어렵지 않다,

- 선언된 1 차원 배열의 접근
배열의 선언방법을 알았으니, 이제 접근방법을 살펴볼 차례이다. 그런데 배열의 접근방법도 어렵지 않다. 그럼 아래의 배열을 대상으로 1 차원 배열의 접근방법을 살펴보겠다.
int arr[3]; // 길이가 3 인 int형 1 차원 배열
위의 배열을 대상으로 값을 저장할 때에는 다음의 형태로 접근을 한다.
arr[0]=10; // 배열 arr의 첫 번째 요소에 10 을 저장해라
arr[1]=12; // 배열 arr의 두 번째 요소에 12 를 저장해라
arr[2]=25; // 배열 arr의 세 번째 요소에 25 를 저장해라
즉, 배열요소의 접근에는 다음의 식이 존재한다.
arr[idx]=20; → 배열 arr의 idx+1 번째 요소에 20 을 저장해라
이렇듯 []연산자 사이에 배열의 위치정보를 명시하게 되는데, 여기서 중요한 사실은 첫 번째 요소를 지칭할 때 사용되는 숫자가 1 이 아닌 0 이라는 점이다. 즉, 배열의 위치 정보를 명시하는 인덱스 값은 1 이 아닌 0 에서부터 시작한다. 그럼 다음 예제를 통해서 간단히 배열을 선언하고 배열요소에 접근해보겠다.

```c
#include <stdio.h>

int main(void)
{
        int arr[5];
        int sum=0, i;

        arr[0]=10, arr[1]=20, arr[2]=30, arr[3]=40, arr[4]=50;

        for(i=0; i<5; i++);
                sum += arr[i];

        printf("배열요소에 저장된 값의 합 : %d \n", sum);
        return 0;
}
```

```
배열요소에 저장된 값의 합 : 32767
```

위 예제를 통해서 알 수 있는 사실은 배열의 접근방법이 전부가 아니다. 위 예제에 있는 for문을 다시 한번 보자. 이 for문을 통해서 추가로 "배열의 모든 요소는 반복문을 이용해서 순차적으로 접근하는 것이 가능하다."라는 것이다.
다수의 변수를 선언해서는 반복문 기반의 순차적인 접근이 불가능하다. 때문에 반복문 기반의 순차적 접근은 배열의 또 다른 매력으로 작용한다.

- 배열, 선언과 동시에 초기화하기
기본 자료형 변수들은 선언과 동시에 초기화가 가능하다. 마찬가지로 배열도 선언과 동시에 원하는 값으로 초기화할 수 있다. 초기화의 방법은 총 3 가지로 구분이 가능한데, 그 중 하나는 다음과 같다.
int arr1[5]={1,2,3,4,5}; // 순차적으로 1,2,3,4,5 로 초기화함
참고로 배열의 초기화를 목적으로 선언된, 중괄호로 묶인 부분을 가리켜 초기화 리스트라 한다.
자, 그럼 2 번째 초기화의 예를 보이겠다.
int arr2[ ]={1,2,3,4,5,6,7}; // 컴파일러에 의해서 자동으로 7 이 삽입됨
위의 문장에서 보이듯이 초기화를 목적으로 초기화 리스트가 선언되면, 배열의 길이정보를 생략할 수 있다.

이러한 경우에는 컴파일러가 초기화 리스트의 수를 참조하여 길이정보를 채워주기 때문이다.
이제 마지막으로 세 번째 초기화의 예를 보이겠다. 이는 배열의 길이를 다 채울 만큼의 초기 값이 선언되지 않은 경우이다.
Int arr3[5]={1,2}; // 3,4,5 번째 배열요소는 0 으로 채워짐
위의 문장에서 선언하는 배열의 길이는 5 이다. 그런데 초기화할 값은 2 개밖에 존재하지 않는다. 이러한 경우 첫번째 요소부터 순차적으로 값을 채워나가되 채울 값이 존재하지 않는 요소들은 0 으로 채워진다.
그럼 예제를 통해서 지금까지 설명한 내용을 확인해보겠다. 더불어 배열의 이름을 대상으로 하는 sizeof 연산의 결과도 함께 확인해보겠다.

```c
#include <stdio.h>

int main(void)
{
	int arr1[5]={1,2,3,4,5};
	int arr2[ ]={1,2,3,4,5,6,7};
	int arr3[5]={1,2};
	int ar1len, ar2len, ar3len, i;

	printf("배 열  arr1의  크 기 : %d \n", sizeof(arr1));
	printf("배 열  arr2의  크 기 : %d \n", sizeof(arr2));
	printf("배 열  arr3의  크 기 : %d \n", sizeof(arr3));

	ar1len = sizeof(arr1) / sizeof(int); // 배 열  arr1의  길 이  계 산
	ar2len = sizeof(arr2) / sizeof(int); // 배 열  arr2의  길 이  계 산
	ar3len = sizeof(arr3) / sizeof(int); // 배 열  arr3의  길 이  계 산

	for(i=0; i<ar1len; i++)
		printf("%d", arr1[i]);
	printf("\n");

	for(i=0; i<ar2len; i++)
		printf("%d", arr2[i]);
	printf("\n");

	for(i=0; i<ar3len; i++)
		printf("%d", arr3[i]);
	printf("\n");
	return 0;
}
```

```
배 열  arr1의  크 기 : 20
배 열  arr2의  크 기 : 28
배 열  arr3의  크 기 : 20
12345
1234567
12000
```

위의 실행결과에서 보이듯이 배열의 이름을 대상으로 하는 sizeof 연산의 결과로는 바이트 단위의 배열 크기가 반환된다. 따라서 배열의 크기가 아닌, 길이를 계산하고 싶다면 위 예제 14~16 행에 보이는 바와 같이 문장을 구성해야한다.

2. 배열을 이용한 문자열 변수의 표현

배열을 공부했으니 이제 서서히 문자열에 대해서 이야기를 꺼낼 때가 되었다. 그런데 도대체 문자열과 배열이 무슨 상관이 있길래 배열을 이야기하다 말고 문자열을 이야기하려는 것일까? Char형 배열을 이용하면 문자열의 저장뿐만 아니라 문자열의 변경도 가능해진다. 즉, 변수형태의 문자열 선언이 가능해진다. 때문에 배열, 아니 char형 배열과 문자열에는 큰 상관이 있다.

- char형 배열의 문자열 저장과 널 문자

이전에도 언급했듯이 c언어에서는 큰 따옴표를 이용해서 문자열을 표현한다. 따라서 다음과 같이 문장을 구성하면 배열에 문자열이 저장된다.

char str[14]="Good morning!";

위의 선언을 통해서 메모리 공간에는 char형 배열이 할당되고, 이 배열에는 다음의 형태로 문자열이 저장된다.

G	O	O	D		m	o	r	n	i	n	g	!	₩0

물론 다음과 같이 배열의 길이를 생략하는 것도 가능하며, 이 경우에는 컴파일러가 문자열의 길이를 계산해서 배열의 길이를 14 로 결정한다.

Char str[ ]="Good morning!"; // 배열의 길이는 14 로 결정

그런데 배열의 길이가 14 라니 좀 이상하지 않은가? 문자열을 구성하는 문자의 수는 중간에 삽입된 공백 문자를 포함하여 분명 13 이다. 그러나 위의 그림에서 보이듯이 문자열의 끝에는 ₩0 이라는 특수문자가 자동으로 삽입이 되어 실제 문자열의 길이는 14 가 된다. 따라서 문자열의 저장을 목적으로 char형 배열을 선언할 경우에는 특수문자 ₩0 이 저장될 공간까지 고려해서 배열의 길이를 결정해야한다. 이렇듯 문자열의 끝에 자동으로 삽입되는 문자 ₩0 을 가리켜 널 문자라 한다. 다음 예제를 통해 실제로 널 문자가 삽입이 되는지 확인해보기로 하자.

```c
#include <stdio.h>

int main(void)
{
        char str[]="Good morning!";
        printf("배열 str의 크기 : %d \n", sizeof(str));
        printf("널 문자 문자형 출력 : %c \n", str[13]);
        printf("널 문자 정수형 출력 : %d \n", str[13]);

        str[12]='?'; // 배열 str에 저장된 문자열 데이터는 변경 가능
        printf("문자열 출력 : %s \n", str);
        return 0;
}
```

```
배열 str의 크기 : 14
널 문자 문자형 출력 :
널 문자 정수형 출력 : 0
문자열 출력 : Good morning?
```

위 예제에서는 널 문자의 서식문자 %c와 %d를 이용해서 각각 문자와 정수의 형태로 출력해 보이고 있다. 그리고 그 출력결과를 통해서 다음의 사실을 알 수 있다.

"널 문자의 아스키 코드 값은 0 이다. 그리고 이를 문자의 형태로 출력할 경우, 아무런 출력이 발생하지 않는다."

그런데 위의 실행결과만 놓고 보면 널 문자와 공백 문자를 혼동할 수 있기 때문에 주의를 해야 한다. 공백문자의 아스키 코드 값은 32 로, 이는 아스키 코드 값이 0 인, 그리고 ₩0 으로 표시되는 널 문자와는 엄연히 다른 것이다.

- scanf 함수를 이용한 문자열의 입력

이번에는 scanf 함수를 이용해서 배열에 문자열을 입력 받는 방법을 소개하고자 한다. Printf 함수를 이용해서 문자열을 출력할 때 서식문자 %s를 사용하듯이 scanf 함수를 이용해서 문자열을 입력받을때에도 서식문자 %s를 사용한다. 그럼 이와 관련해서 다음 예제를 보자

```c
#include <stdio.h>

int main(void)
{
        char str[50];
        int idx=0;

        printf("문자열 입력 : ");
        scanf("%s", str); // 문자열을 입력 받아서 배열 str에 저장
        printf("입력 받은 문자열 : %s \n", str);

        printf("문자 단위 출력 : ");
        while(str[idx] != '\0');
        {
                printf("%c", str[idx]);
                idx++;
        }
        printf("\n");
        return 0;
}
```

```
문자열 입력 : choco
입력 받은 문자열 : choco
문자 단위 출력 : choco
```

위 예제의 9 행에는 다음 문장이 삽입되어 있다.

Scanf("%s", str); // str 앞에 &연산자를 삽입하지 않는다.

위의 문장에서 서식문자 %s는 문자열의 입력을 뜻한다. 그리고 str은 문자열이 저장될 배열을 명시한 것이다. 그런데 여기서 이상한 점이 하나 있다. 지금까지는 입력 받을 대상 앞에는 다음과 같이 &연산자를 붙여왔다.

Scanf("%d", &num); // & 연산자 반드시 삽입

그래서 문자열을 입력 받을 때에도 다음과 같이 & 연산자를 추가해야 한다고 생각할 수 있다. 하지만 문자열을 입력 받는 배열의 이름 앞에는 &연산자를 붙이면 안된다. "scanf함수 호출문 구성 시, 데이터를 저장할 변수의 이름 앞에는 &연산자를 붙여줘야한다. 그러나 문자열을 입력 받는 배열의 이름 앞에는 &연산자를 붙이지 않는다."

그리고 위 예제의 while문을 보면 알 수 있듯이, scanf 함수호출을 통해서 입력 받은 문자열의 끝에도 널 문자가 삽입되어 있다. c언어에서 표현하는 모든 문자열의 끝에는 널 문자가 삽입된다.

이렇듯 문자열에 있어서 널 문자의 존재는 매우 중요하다.

Char arr1[ ]={'H', 'I', '~'}; // 마지막에 널 문자가 없으므로 문자 배열

이는 다만 문자가 저장된 배열일 뿐이다. 하지만 다음은 문자열이 저장된 배열이라 할 수 있다. 마지막에 널 문자가 삽입되었기 때문이다.

Char arr2[ ]={'H', 'I', '~', '₩0'}; // 마지막에 널 문자가 있으므로 문자열

이렇듯 문자열의 판단여부에 있어서 선언방법은 중요하지 않다. 어떻게 선언이 되든 널 문자가 마지막에 존재하면 이는 C언어의 관점에서 문자열이 되는 것이다.

- 문자열의 끝에 널 문자가 필요한 이유

사실 문자열의 끝에 널 문자를 삽입한 이유는 앞서 보인 예제의 다음 반복문을 통해서 알 수 있다.

```
While(str[idx] != '₩0')
{
        Printf("&c", str[idx]);
        Idx++;
}
```

문자열의 출력에 사용되는 서식문자 %s가 존재하지 않는다고 가정해보자. 그래서 배열요소에 일일이 접근을

해서 문자열의 전부를 출력해야 한다고 가정해보자. 이러한 상황에서 문자열의 끝을 나타내기 위한 어떠한 도구도 마련되어 있지 않다면, 문자열을 출력할 수 있겠는가? 다시 말해서 하나의 문자열을 구분해 낼 수 있겠는가?

사실 메모리상에서 문자열은 이진 데이터로 저장되기 때문에 문자열의 시작과 끝이 표시되어 있지 않다면 문자열을 구분하는 것은 불가능하다. 그래서 널 문자를 이용해서 문자열의 끝을 표시하는 것이다. 다행히도 우리는 다음 배열을 보면서,

Char str[100] = " . . . . "; // 문자열의 길이는 100 보다 작다

문자열의 시작과 끝을 str[0]이 문자열의 시작위치이고, 문자열의 끝에는 널 문자가 삽입되어 있다는 것을 판단할 수 있다. Printf 함수도 위의 기준으로 문자열을 구분한다. %s를 기반으로 문자열의 출력을 명령하면, 위의 기준을 가지고 문자열을 출력한다. 그럼 이와 관련해서 다음 예제를 관찰하자.

```c
#include <stdio.h>

int main(void)
{
        char str[50]="I like C programming";
        printf("string: %s \n", str);

        str[8]='\0'; // 9번째 요소에 널 문자 저장
        printf("string: %s \n", str);

        str[6]='\0'; // 7번째 요소에 널 문자 저장
        printf("string: %s \n", str);

        str[1]='\0'; // 2번째 요소에 널 문자 저장
        printf("string: %s \n", str);
        return 0;
}
```

```
string: I like C programming
string: I like C
string: I like
string: I
```

위의 예제에서는 문자열의 중간에 널 문자를 삽입해서 문자열의 끝을 변경하고 있다. 그리고 이렇게 문자열이 변경되었을 때, 변경된 끝을 기준으로 문자열이 출력됨을 보이고 있다.

- scanf 함수의 문자열 입력특성

앞서 제시한 (scanf를 통해 문자열을 입력하는)예제에 "He is my friend"라는 문자열을 입력하면 실행의 결과는

문자열 입력: He is mt friend

입력 받은 문자열: He

문자단위출력: He

Scanf 함수는 데이터를 구분 짓는 기준이 공백이기 때문이다. 따라서 이러한 출력결과를 보여준다. 즉, scanf 함수는 문장을 입력 받기에는 적절치 않다.

2. 포인터의 이해

1. 포인터란 무엇인가?

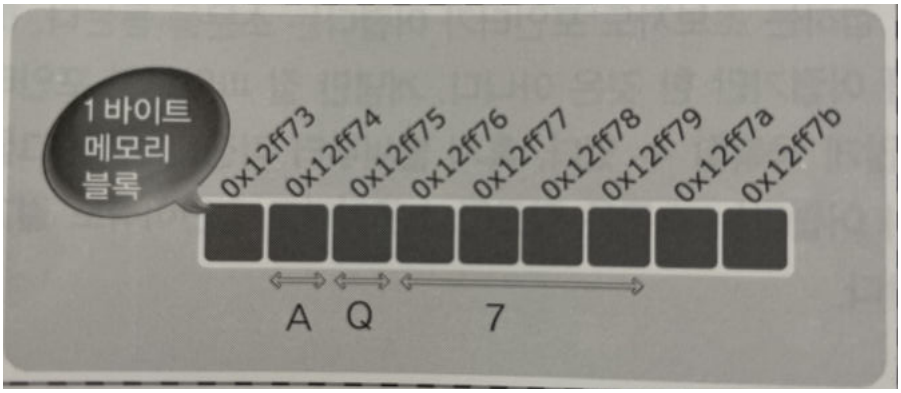
포인터는 C언어가 Low레벨 언어의 특성을 지닌다고 이야기하게 만든 장본인이다. 왜냐하면 포인터를 이용하면 메모리에 직접 접근이 가능하기 때문이다.

- 주소 값의 저장을 목적으로 선언되는 변수

포인터를 설명하기에 앞서 변수가 메모리에 어떻게 존재하게 되는지, 복습을 겸하여 그림을 통해서 설명해 보겠다. 다음과 같이 변수가 선언되었다고 가정해보자.

```
Int main(void)
{
        char ch1='A', ch2='Q';
        Int num = 7;
        . . . .
}
```

그럼 총 6 바이트(1+1+4)가 메모리 공간에 다음과 같이 할당된다.



위 그림에서 메모리 블록의 상단에 표시한 것은 메모리 주소 값이다. 그림에서 보이듯이 1 바이트 메모리 공간을 단위로 하나의 주소 값이 할당되며, 주소 값도 1 씩 증가한다. 그리고 이 그림에서는 문자 A가 저장되어 있는 변수 ch1 과 문자 Q가 저장되어 있는 변수 ch2 가 각각 0x12ff74 번지와 0x12ff75 번지에, 그리고 정수 7 이 저장되어 있는 변수 num은 0x12ff76 번지에 할당된 것으로 표현되고 있다.

Int형 변수 num은 어디에 할당되어 있을까? Int형 변수 num은 0x12ff76 번지에서부터 0x12ff79 까지 할당되어 있다. 하지만 C언어에서는 시작번지만을 가지고 위치를 표현한다. 왜냐하면 int형 변수는 4 바이트이므로 변수의 끝이 어딘지는 쉽게 계산이 가능하기 때문이다. 따라서 우리는 위와 같은 질문에 0x12ff76 번지에 할당되어 있다고 할 수 있다. 그런데 0x12ff76 역시 정수이다. 따라서 이것도 저장이 가능한 값이며, 이의 저장을 위해 마련된 변수가 바로 포인터 변수이다.

"포인터 변수란 메모리의 주소 값을 저장하기 위한 변수이다"

참고로 포인터는 변수 형태의 포인터와 상수 형태의 포인터를 어우르는 표현이다. 그런데 포인터와 관련된 이야기의 대부분이 포인터 변수와 관련이 있으므로, 포인터라 하면 우선 포인터 변수를 연상하면 된다.

- 포인터 변수와 & 연산자에 대해서 간단히 맛보기

정수 7 이 저장된 int형 변수 num을 선언하고 이 변수의 주소 값 저장을 위한 포인터 변수 pnum을 선언하자. 그리고 나서 pnum에 변수 num의 주소 값을 저장하자. 코드로는 다음과 같다.

```
int main(void)
{
        int num=7;
        int * pnum; // 포인터 변수 pnum의 선언
        pnum = &num; // num의 주소 값을 포인터 변수 pnum에 저장
        . . . .
}
```

위의 코드에서 다음의 문장이 바로 포인터 변수의 선언이다.

Int * pnum;

위의 문장은 다음과 같이 해석하면 된다.

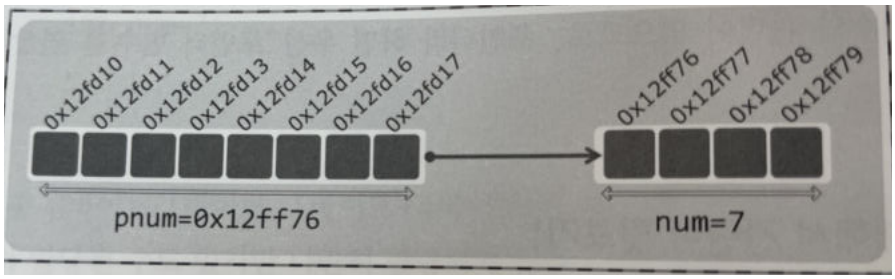
Pnum: 포인터변수의 이름

Int * : int형 변수의 주소 값을 저장하는 포인터 변수의 선언

때문에 pnum은 int형 변수의 주소 값을 저장할 수 있는 포인터 변수가 된다. 이어서 위 코드의 다음 문장을 보자.

Pnum = #

위의 문장에서 &연산자는 오른쪽에 등장하는 피연산자의 주소 값을 반환하는 연산자이다. 따라서 위의 문장에서는 & 연산의 결과로 변수 num의 주소 값이 반환되며, 이를 포인터 변수 pnum에 저장하게 된다.



위의 그림에서 보이듯이 포인터 변수 pnum에는 변수 num의 시작번지 주소 값이 저장된다. 그리고 이 상황을 다음과 같이 표현한다. "포인트 변수 pnum이 int형 변수 num을 가리킨다."

그런데 위 그림에서 변수 num의 크기는 4 바이트(32 비트)인데 비해 포인터 변수 pnum의 크기는 8 바이트(64 비트)로 묘사되어 있다. 그렇다면 포인터 변수의 크기가 8 바이트란 뜻인가?

포인터 변수의 크기는 4 바이트가 될 수도 8 바이트가 될 수도 있다. 32 비트 시스템에서는 주소 값을 32 비트로 표현하기 때문에 포인터 변수의 크기가 4 바이트인 반면, 64 비트 시스템에서는 주소 값을 64 비트로 표현하기 때문에 포인터 변수의 크기가 8 바이트이다. 이렇듯 포인터 변수의 크기가 주소 값의 크기와 일치해야 주소 값을 저장할 수 있을 것 아닌다, 따라서 주소 값의 크기와 포인터 변수의 크기가 동일한 것은 당연하다.

- 포인터 변수 선언하기

포인터 변수는 가리키고자 하는 변수의 자료형에 따라서 선언하는 방법이 달라진다. 사실 주소 값은 동일한 시스템에서 그 크기가 동일하며 모두 정수의 형태를 띤다. 그래도 가리키고자 하는 변수의 자료형에 따라서 선언하는 방법이 다음과 같이 달라진다.

- Int * pnum1;
  Int * 는 int형 변수를 가리키는 pnum1 의 선언을 의미함
- Double * pnum2;
  Double * 는 Double형 변수를 가리키는 pnum2 의 선언을 의미함

- Unsigned int * pnum3;
  Unsigned int * 는 Unsigned int형 변수를 가리키는 pnum3 의 선언을 의미함

위의 예에서 보이듯이 포인터 변수 선언의 기본 공식은 다음과 같다.

- Type * ptr;
  Type형 변수의 주소 값을 저장하는 포인터 변수 ptr의 선언

그렇다면 가리킬 변수의 자료형에 따라서 포인터 변수의 선언방식을 나누어 놓은 이유는 무엇일까? 이에 대해서는 잠시 후에 설명할 것이다. 하지만 한가지는 여기서 확실히 하고 넘어갈 수 있다. 이와 관련해서 다음 코드를 보자.

```
Int main(void)
{
        Int * pnum;
        . . . .
        Pnum = & . . . .
        . . . .
}
```

위의 코드에서 pnum이 가리키는 변수의 자료형은 무엇이겠는가? 코드의 상당수가 가려져있지만 보나마나 pnum은 int형 변수를 가리킬 것이다. 이렇듯 우리는 포인터 변수의 선언형태만 보고도 이 포인터가 현재 가리키는 변수의 자료형을 짐작할 수 있다.

- 포인터의 형

Int, char, double과 같이 변수의 선언 및 구분에 사용되는 키워드를 자료형이라 하듯이 포인터 변수의 선언 및 구분에 사용되는 int *, char *, double *등을 가리켜 포인터 형이라한다. 하지만 포인터 변수도 값을 저장하는 변수이기 때문에 포인터 형 역시 자료형의 범주에 포함시키기도 한다. 이들 포인터 형은 다음과 같이 표현한다

Type * : type형 포인터

Type * ptr : typr형 포인터 변수 ptr

2. 포인터와 관련 있는 연산자: &연산자와 *연산자

일반적으로 &연산자와 *연산자를 가리켜 포인터 연산자라 하는데, 여기서 말하는 * 연산자는 곱셈 연산자를 뜻하는 것이 아니다. 곱셈 연산자는 피연산자가 두 개인 이항 연산자인 반면, 이어서 언급할 * 연산자는 피연산자가 한 개임 단항 연산자이다. 이렇듯 * 연산자는 사용되는 위치에 따라서 그 의미가 달라진다.

- 변수의 주소 값을 반환하는 & 연산자

앞서 한 차례 설명한 & 연산자는 피연산자의 주소 값을 반환하는 연산자이다. 따라서 다음의 형태로 연산문을 구성해야 한다.

```
int main(void)
{
        int num = 5;
        int * pnum = &num; // num의 주소 값을 반환해서 포인터 변수 pnum을 초기화
        . . . .
}
```

이렇듯 &연산자의 피연산자는 변수이어야 하며, 상수는 피연산자가 될 수 없다. 그리고 다음과 같이 변수형에

맞지 않는 포인터 변수의 선언은 문제가 될 수 있다.

```
int main(void)
{
        int num1 =. ;
        double * pnum1 = &num1; // 일치하지 않음

        double num2 = 5;
        int * pnum2 = &num2; // 일치하지 않음
        . . . .
}
```

위와 같이 int형 변수의 주소 값을 double형 포인터 변수에 저장하거나 double형 변수의 주소 값을 int형 포인터 변수에 저장하는 것을 잘못된 것이다. 비록 컴파일 에러는 발생하지 않지만 이어서 설명하는 포인터 관련 * 연산 시 문제가 발생한다.

- 포인터가 가리키는 메모리를 참조하는 * 연산자

* 연산자는 포인터가 가리키는 메모리 공간에 접근할 때 사용하는 연산자이다. 이와 관련해서 다음 코드를 보자.

```
int main(void)
{
        int num = 10;
        int * pnum = &num; // 포인터 변수 Pnum이 변수 num을 가리키게 하는 문장
        *pnum=20; // pnum이 가리키는 변수에 20 을 저장하라
        printf("%d", *pnum); // pnum이 가리키는 변수를 부호 있는 정수로 출력하라!
        . . . .
}
```

위의 포인터 변수 pnum은 변수 Num을 가리키고 있다. 따라서 *pnum이 의미하는 바는 다음과 같다.

"포인터 변수 pnum이 가리키는 메모리 공간인 변수 num에 접근을 해서..."

때문에 다음 두 문장은

```
*pnum=20;
printf("%d", *pnum);
```

각각 다음과 같이 해석이 된다.

"포인터 변수 pnum이 가리키는 메모리 공간인 변수 num에 20 을 저장해라"

"포인터 변수 pnum이 가리키는 메모리 공간인 변수 num에 저장된 값을 출력해라"

이렇듯 사실상 *pnum은 포인터 변수 pnum이 가리키는 변수 num을 의미하는 것이다. 그럼 지금까지 설명한 내용의 확인을 위해서 예제를 보자

```
#include <stdio.h>

int main(void)
{
        int num1=100, num2=200;
        int * pnum;

        pnum=&num1; // 포인터 pnum이 num1을 가리킴
        (*pnum)+=30; // num1+=30; 과 동일

        pnum=&num2; // 포인터 pnum이 num2를 가리킴
        (*pnum)-=30; // num2-=30 과 동일

        printf("num1:%d, num2:%d \n", num1, num2);
        return 0;
}
```

```
num1:130, num2:170
```

여기서 중요한 것은 포인터 변수 pnum이 가리키는 대상이 num1 에서 num2 로 한차례 변경되었다는 것이다.

- 다양한 포인터 형이 존재하는 이유

아래의 return문에서 pnum은 포인터 변수이다. 그럼 이 문장을 보고 잠시 후에 이어지는 질문에 답을 해보자.

return * pnum;

위의 문장을 통해서 값을 반환하려면 pnum이 가리키는 메모리 공간에 접근을 해서 값을 읽어 들여야 한다. 그렇다면 어떻게 값을 읽어 들여야 하겠는가? 이와 관련해서 아래의 질문에 답을 해보자

"pnum에 저장된 주소를 시작으로 몇 바이트를 읽어 들여야 하는가? 그리고 읽어 들인 데이터는 정수로 해석해야 하는가? 실수로 해석해야 하는가?"

사실 위의 return문만 가지고는 위의 질문에 답을 하지 못한다. 위의 질문에 답을 하기 위해서는 pnum의 포인터 형 정보가 필요하다. 자 그럼 Pnum을 int형 포인터 변수라고 가정하고 답을 해보자.

"pnum이 int형 포인터 변수이므로 pnum에 저장된 주소를 시작으로 4 바이트를 읽어 들여서 이를 정수로 해석해야한다."

그럼 pnum을 double형 포인터 변수라고 가정하고 답을 해보자.

"pnum이 double형 포인터 변수이므로 pnum에 저장된 주소를 시작으로 8 바이트를 읽어 들여서 이를 실수로 해석해야한다."

이렇듯 포인터의 형은 메모리 공간을 참조하는 기준이 된다. 즉, 포인터 형을 정의한 이유는 * 연산자를 통한 메모리 공간의 접근 기준을 마련하기 위함이다. 그럼 아래의 코드를 보면서 무엇이 문제인지 설명해보자.

```
int main(void)
{
        double num=3.14;
        int * pnum=&num; // 형의 불일치
        printf("%d", *pnum); // 예측 불가능한 의미 없는 출력
        . . . .
}
```

위의 코드를 컴파일 하면 경고 메시지는 출력되지만, 컴파일 에러가 발생하지는 않는다. 이는 c언어가 메모리 접근에 재한 유연성을 최대한 보장하기 때문인데, 그래서 프로그래머는 메모리 접근에 더 신중을 기해야 한다. 위의 코드를 실행하게 되면 int형 포인터 변수 pnum은 double형 변수 num을 가리키게 된다. 따라서 pnum이 가리키는 메모리 공간에 저장된 값을 얻기 위해서 * 연산을 하는 경우 다음의 형태로 데이터를 읽어서 해석하게 된다.

"4 바이트를 읽어 들여서 이를 정수로 해석한다."

즉, 실제 저장할 때는 8 바이트 크기의 실수형 데이터로 저장을 했는데, 해석을 전혀 엉뚱맞게 해버린 셈이다. 따라서 얼마가 출력될지 예측이 불가능할 뿐만 아니라, 이렇게 해석해서 얻게 되는 정수 값은 아무 의미도 없는 값일 뿐이다.

마지막으로 다시 한번 정리하겠다. 포인터의 형이 존재하는 이유는 메모리 기반의 메모리 접근기준을 마련하기 위함이다. 포인터에 형이 존재하지 않는다면 * 연산을 통한 메모리의 접근은 불가능하다.

- 잘못된 포인터의 사용과 널 포인터

포인터 변수에는 메모리의 주소 값이 저장되고, 이를 이용해서 해당 메모리 공간에 접근도 가능하기 때문에 포인터와 관련해서는 상당히 주의를 해야 한다. 따라서 이번에는 포인터가 잘못 사용된 예를 보이겠다. 다음은 그 첫번째 사례이다.

```
int main(void)
{
        int * ptr; // 포인터 변수 ptr은 쓰레기 값으로 초기화 됨
        *ptr=200;
        . . . .
}
```

위와 같이 포인터 변수를 선언만하고 초기화하지 않으면, 포인터 변수는 쓰레기 값으로 초기화 된다. 즉, 어디를 가리킬지 모르게 된다. 때문에 이러한 상태에서 * 연산을 통해 200 을 저장하는 것은 치명적인 결과로 이어질 수 있다. ptr이 가리키는 위치가 어디인 줄 알고 값을 저장하는 건가? 만약에 ptr이 가리키는 메모리 공간이 매우 중요한 위치였다면, 이는 시스템 전체에 심각한 문제를 일으킬 수도 있는 상황이다. 다행히 요즘의 운영체제는 이렇게 잘못된 메모리 접근의 시도가 있을 때, 이를 감지하고 해당 프로그램을 중지시켜서 잘못된 메모리의 접근은 미연에 방지한다. 그런데 프로그램을 중지시킨 이후에 전달하는 메시지의 형태는 운영체제 별로 차이가 으니 직접 확인해보기 바한다. 그럼 이번에는 포인터가 잘못 사용된 두 번째 사례를 보이겠다. 사실 이는 첫 번째 사례와 유사하기 때문에 여러분이 문제점을 찾아서 설명할 수도 있을 것이다.

```
int main(void)
{
        int * ptr = 125; // 125 번지가 어딘 줄 알고?
        *ptr=10;
        . . . .
}
```

위의 경우는 포인터 변수 ptr을 초기화 한답시고 125 를 저장하였다. 그러나 더 우스운 꼴이 되어버리고 말았다. 125 번지가 어딘 줄 알고 포인터 변수를 125 로 초기화 하는가? 결국 이는 쓰레기 값으로 포인터 변수를 초기화한 것과 다르지 않다.

"그렇다면 포인터 변수는 어떤 값으로 초기화를 시켜야 합니까?"

포인터 변수를 우선 선언만 해 놓고, 이후에 유효한 주소 값을 채워 넣을 생각이라면 다음과 같이 초기화를 하는 것이 좋다.

```
int main(void)
{
        int * ptr1=0;
        int * ptr2=NULL; // NULL은 사실상 0 을 의미함
        . . . .
}
```

위에서 ptr1 을 초기화하는 값 0 을 가리켜 널 포인터라 한다. 이는 0 번지를 의미하는 것이 아니다. 이것이 의미하는 바는 다음과 같다.

"아무데도 가리키지 않는다"

따라서 이를 이용한 * 연산은 메모리 공간에 어떠한 영향도 미치지 않는다. 물론 프로그램이 멈추는 현상은 동일하게 일어나지만, 이는 잘못된 메모리의 접근에 대해 보호장치가 없는 운영체제에서도 시스템에 치명적인 영향을 주지 않는다. 그리고 키워드 NULL은 널 포인터를 의미하며, 실제로 이는 상수 0 으로 정의되어 있다.

끝으로 포인터는 그 특성상 다양한 형태의 버그 발생확률을 높이기 때문에, 처음에는 포인터의 사용이 부담스러울 것이다. 실제로 잘못된 포인터의 사용으로 인해서 밤을 꼬박 세우는 일도 종종 있을 것이다.

3. 포인터와 배열

1. 포인터와 배열의 관계

- 배열의 이름은 무엇을 의미하는가?

배열의 이름은 포인터이다. 단, 그 값을 바꿀 수 없는 상수 형태의 포인터이다. 다음 예제에서는 이러한 사실을 증명하고 있다.

```
#include <stdio.h>

int main(void)
{
        int arr[3]={0, 1, 2};
        printf("배 열 의  이 름 : %p \n", arr);
        printf("첫  번 째  요 소 : %p \n", &arr[0]);
        printf("두  번 째  요 소 : %p \n", &arr[1]);
        printf("세  번 째  요 소 : %p \n", &arr[2]);
        // arr = &arr[i]; // 이  문 장 을  컴 파 일  에 러 를  일 으 킨 다 .
        return 0;
}
```

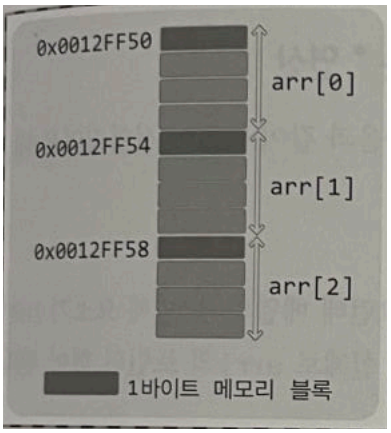
```
배 열 의  이 름 : 0x7ff7aff83a3c
첫  번 째  요 소 : 0x7ff7aff83a3c
두  번 째  요 소 : 0x7ff7aff83a40
세  번 째  요 소 : 0x7ff7aff83a44
```

%p는 주소 값의 출력에 사용되는 서식문자이다.

위의 실행결과가 우리에게 시사하는 바는 크다. 먼저 위의 실행결과를 통해서 우리는 다음 사실을 알 수 있다.

"int형 배열요소간 주소 값의 차는 4 바이트이다."

우리가 선언한 배열이 int형 배열이므로 각 요소 별로 할당하는 메모리 공간의 크기는 4 바이트이다. 따라서 배열요소간 주소 값을 차가 4 바이트임을 보이는 위의 실행결과는 모든 배열요소가 메모리 공간에 나란히 할당된다라는 사실을 증명하는 결과가 된다. 위의 실행결과를 근거로 하는 배열 arr의 메모리 공간 할당 형태는 다음과 같다.



그리고 위 예제의 실행결과를 통해서 다음 사실도 확인할 수 있다.

"베열의 첫 번째 바이트의 주소 값과 배열의 이름을 출력한 결과가 같다."

게다가 위 예제 10 행에서 보이듯이 배열의 이름은 대입연산자의 피연산자가 될 수 없으므로 다음의 결론을 내릴 수 있다.

"배열의 이름은 배열의 시작 주소 값을 의미하며, 그 형태는 값의 저장이 불가능한 상수이다."

비교조건 / 비교대상	포인터 변수	배열의 이름
이름이 존재하는가 ?	존재	존재
무엇을 나타내거나 저장하는가 ?	메모리의 주소 값	메모리의 주소 값
주소 값의 변경이 가능한가 ?	가능	불가능

위의 표에서 보이듯이 배열의 이름이나 포인터 변수나 둘 다 이름이 존재하며, 특정 메모리 공간의 주소 값을 지닌다. 다만 포인터 변수는 그 이름을 의미하듯이 변수지만, 배열의 이름은 가리키는 대상의 변경이 불가능한 상수하는 점에서만 차이를 보인다. 즉 배열의 이름은 상수 형태의 포인터이다. 그래서 배열의 이름을 가리켜 포인터 상수라 부르기도 한다.
"배열의 이름도 포인터라면 포인터 변수를 대상으로 하는 * 연산도 가능한가?"
가능하다. 배열의 이름도 포인터이기 때문에 배열의 이름을 피연산자로 하는 * 연산이 가능하다.

- 1 차원 배열이름의 포인터 형과 배열이름을 대상으로 하는 * 연산
사실 1 차원 배열이름의 포인터 형을 결정짓는 것은 어렵지 않다. 다음과 같이 배열이 선언되었을 때,
int arr1[5]; // arr1 은 int형 포인터 상수
배열의 이름 arr1 이 가리키는 것은 배열의 첫 번째 요소 아닌가? 그런데 배열의 첫 번째 요소가 int형 변수이니, arr1 은 int형 포인터(int *)라는 결론이 나오며, 이것이 실제로 arr1 의 포인터 형이 된다. 럼 다음 배열이름의 포인터 형을 결정지어 보자.
double arr2[7]; // arr2 는 double형 포인터 상수
배열이름 arr2 가 가리키는 것은 첫 번째 배열요소인 double형 변수이므로 arr2 는 double형 포인터(double *)가 된다. 이렇듯 1 차원 배열이름의 포인터 형은 배열의 이름이 가리키는 대상을 기준으로 결정하면 된다. 이제 1 차원 배열이름의 포인터 형을 결정할 수 있게 되었으니, 이를 대상으로 * 연산을 해 볼 차례이다.

```
#include <stdio.h>

int main(void)
{
        int arr1[3]={1,2,3};
        double arr2[3]={1.1, 2.2, 3.3};

        printf("%d %g \n", *arr1, *arr2);
        *arr1 += 100;
        *arr2 += 120.5;
        printf("%d %g \n", arr1[0], arr2[0]);
        return 0;
}
```

8 행: arr1 과 arr2 가 가리키는 대상을 출력하고 있다. arr1 은 int형 포인터이므로 4 바이크 크기의 메모리를 정수의 형태로 참조하게 되고, arr2 는 double형 포인터이므로 8 바이트 크기의 메모리를 실수의 형태로 참조하게 된다. 따라서 arr1 과 arr2 가 가리키는 배열의 첫 번째 요소가 출력된다.
9 행~11 행: 이번에는 arr1 이 가리키는 대상의 값을 100, arr2 가 가리키는 대상의 값을 120.5 증가시키고 있다. 그리고 그 결과의 확인을 위해서 arr1 과 arr2 의 첫 번째 요소에 저장된 값을 출력하고 있다.

```
1 1.1
101 121.6
```

위 예제를 통해서 배열의 이름도 포인터라는 사실을 충분히 이해하고 확인하였을 것이다. 그럼 여기서 아래의 코드를 보면서 한가지를 더 생각해보자.

```
int main(void)
{
        int arr[3]={1, 2, 3};
        arr[0] += 5;
        arr[1] += 7;
        arr[2] += 9; // 포인터를 대상으로 이 문장을 구성한 셈이다.
        . . . .
}
```

위의 코드에서 배열이름 arr은 int형 포인터이니, 포인터를 대상으로 배열의 모든 요소를 arr[0], arr[1], arr[2]로 접근한 셈이다. 그렇다면 포인터 변수를 대상으로도 이러한 형태의 접근이 가능해야 하지 않을까? 배열의 이름이나 포인터 변수나 둘 다 포인터이니 말이다. 실제로 포인터 변수는 배열의 이름처럼 사용할 수 있다. 즉, 포인터 변수 ptr을 대상으로 ptr[0], ptr[1], ptr[2]와 같이 배열의 형태로 메모리 공간에 접근이 가능하다.

- 포인터를 배열의 이름처럼 사용할 수도 있다.

사실 배열의 이름과 포인터 변수는 변수냐 상수냐의 특성적 차이가 있을 뿐, 둘 다 포인터이기 때문에 포인터 변수로 할 수 있는 연산의 배열의 이름으로도 할 수 있고, 배열의 이름으로 할 수 있는 연산은 포인터 변수로도 할 수 있다. 그럼 이러한 사실의 확인을 위해서 다음 예제를 보겠다.

```
#include <stdio.h>

int main(void)
{
        int arr[3]={15, 25, 35};
        int * ptr=&arr[0]; // int * ptr=arr;과 동일한 문장

        printf("%d %d \n", ptr[0], arr[0]);
        printf("%d %d \n", ptr[1], arr[1]);
        printf("%d %d \n", ptr[2], arr[2]);
        printf("%d %d \n", *ptr, *arr);
        return 0;
}
```

6 행: 포인터 변수 ptr이 5 행에 선언된 배열이름 arr과 동일한 주소 값을 갖도록 하기 위해서 첫 번째 요소의 주소 값을 저장하였다. 참고로 이 문장의 주석에서 보이듯이 배열이름을 이용해서 변수 ptr을 초기화해도 그 결과는 같다.

8 행~11 행: 연산의 형태만 놓고 보면, ptr과 arr중 어느 것이 배열의 이름이고 어느 것이 포인터 변수의 이름인지 분간이 안간다. 이렇듯 포인터 변수와 배열의 이름을 대상으로 수행 가능한 연산의 형태에는 차이가 없다.

```
15 15
25 25
35 35
15 15
```

참고로, 위 예제에서 보이듯이 포인터 변수를 배열의 이름처럼 사용하는 경우는 거의 없다. 마찬가지로 배열의 이름을 포인터 변수처럼 사용하는 경우도 거의 없다. 하지만 이러한 일이 가능하다는 사실은 알고 있어야 한다. 이는 포인터를 정확이 이해하는데 중요한 지식이기 때문이다.

2. 포인터 연산

포인터를 대상으로 메모리의 접근을 위한 * 연산 이외에 증가 및 감소연산도 가능하다. 그런데 중요한 것은 증가 및 감소연산이 가능하다는 사실이 아니다. 중요한 것은 연산의 결과이다.

- 포인터를 대상으로 하는 증가 및 감소연산

포인터 변수를 대상으로 다음과 같이 다양한 형태의 증가 및 감소연산을 진행할 수 있다.

```
int main(void)
{
        int * ptr1= . . . .; // 포인터 변수 ptr1 이 적정히 초기화되었다고 가정
        int * ptr2= . . . .; // 포인터 변수 ptr2 가 적절히 초기화되었다고 가정
        ptr1++;
        ptr1 += 3;
        ptr2 -= 5;
        ptr2=ptr1+2;
        . . . .
}
```

그렇다면 이들의 연산의 결과는 무엇일까? 우리가 알고 있는 단순한 산술연산의 결과와 별반 차이가 없을까? 이의 확인을 위해서 다음 예제를 실행해보자.

```
#include <stdio.h>

int main(void)
{
        int * ptr1=0x0010;
        double * ptr2=0x0010;

        printf("%p %p \n", ptr1+1, ptr1+2); // 4가 증가하고 8이 증가한다.
        printf("%p %p \n", ptr2+1, ptr2+2); // 8이 증가하고 16이 증가한다.

        printf("%p %p \n", ptr1, ptr2);
        ptr1++; // 4가 증가한다.
        ptr2++; // 8이 증가한다.
        printf("%p %p \n", ptr1, ptr2);
        return 0;
}
```

```
0x14 0x18
0x18 0x20
0x10 0x10
0x14 0x18
```

위 예제 5 행과 6 행에서 보이는 형태의 초기화는 적절한 초기화가 아니다. 그러나 이 예제에서는 증가연산의 결과확인을 위해서 부적절한 초기화를 진행하였다.

"int형 포인터를 대상으로 1 을 증가시키면 4 증가하고, double형 포인터를 대상으로 1 을 증가시키면 8 이 증가한다."

즉, 포인터를 대상으로 하는 증가연산의 결과는 다음과 같다.

- int형 포인터를 대상으로 n 증가 → n x sizeof(int)의 크기만큼 증가
- double형 포인터를 대상으로 n 증가 → n x sizeof(double)의 크기만큼 증가

그리고 예제를 통해서 보이지는 않았지만, 포인터를 대상으로 하는 감소연산의 결과도 다음과 같다.

- int형 포인터를 대상으로 n 감소 → n x sizeof(int)의 크기만큼 감소
- double형 포인터를 대상으로 n 감소 → n x sizeof(double)의 크기만큼 감소

"TYPE형 포인터를 대상으로 n의 크기만큼 값을 증가 및 감소 시, n x sizeof(TYPE)의 크기만큼 주소 값이 증가 및 감소한다."

이러한 포인터의 연산특성으로 인해서 다음 예제에서 보이는 형태의 배열접근이 가능하다.

```
#include <stdio.h>

int main(void)
{
        int arr[3]={11,22,33};
        int * ptr=arr; // int * ptr=&arr[0]; 과 같은 문장
        printf("%d %d %d \n", *ptr, *(ptr+1), *(ptr+2));

        printf("%d ", *ptr); ptr++; // printf 함수 호출 후, ptr++ 실행
        printf("%d ", *ptr); ptr++;
        printf("%d ", *ptr); ptr--; // printf 함수 호출 후, ptr-- 실행
        printf("%d ", *ptr); ptr--;
        printf("%d ", *ptr); printf("\n");
        return 0;
}
```

```
11 22 33
11 22 33 22 11
```

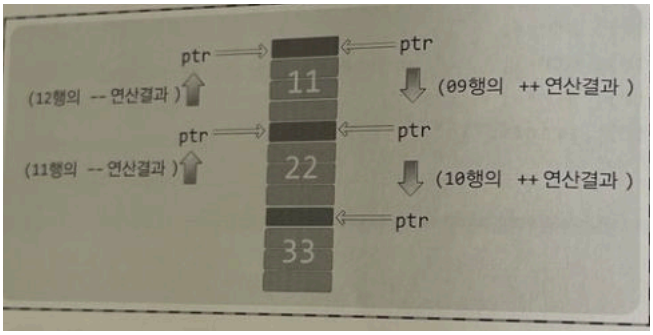
위 예제 6 행에 선언된 포인터 변수 ptr은 int형 포인터이므로 값을 1 증가시키는 연산을 할 때마다 실제로는 4 가 증가한다. 따라서 배열 arr이 할당된 위치의 주소 값을 0x001000 이라 가정할 때, ptr+1 과 ptr+2 의 연산결과로 반환되는 주소 값이 가리키는 위치는 다음과 같다.

ptr = 0x001000

ptr+1 = 0x001004

ptr+2 = 0x001008

때문에 *ptr, *(ptr+1), *(ptr+2)의 참조결과 출력 시 arr[0], arr[1], arr[2]에 저장된 요소가 출력된 것이다. 이어서 9~12 행에서는 포인터 변수 ptr에 저장된 값을 증가 및 감소시키는 연산을 진행하고 있다. 이 결과로 포인터 변수 ptr에 저장된 값은 4 씩 증가 및 감소가 이뤄지기 때문에 포인터 변수 ptr이 가리키는 위치는 다음과 같이 변경된다.



그렇다면 앞서 보인 예제를 근거로 하여 다음 두 연산의 차이점을 말해보겠는가?

- *(++ptr)=20; // ptr에 저장된 값 자체를 변경
- *(ptr+1)=20; // ptr에 저장된 값은 변경되지 않음

위의 두 문장 모두 현재 ptr이 가리키는 위치에서 4 바이트 떨어진 메모리 공간에 20 을 저장하는 문장이다. 하지만 연산 이후 포인터 변수 ptr의 상태에는 차이가 있다. 첫 번째 문장의 경우 ++ 연산의 결과로 인해서

포인터 변수 ptr에 저장된 값이 4 만큼 증가한다. 하지만 두 번째 문장의 + 연산으로 인해서는 ptr에 저장된 값이 증가하지 않는다. 다만 증가된 값을 연산의 결과로 얻어서 * 연산을 진행할 뿐이다.

- 중요한 결론! arr[i] == *(arr+i)

앞서 보인 예제의 코드 일부는 다음과 같다.

```
int main(void)
{
        int arr[3]={11,22,33};
        int * ptr=arr;
        printf("%d %d %d ₩n", *ptr, *(ptr+1), *(ptr+2));
        . . . .
}
```

그리고 *ptr, *(ptr+1), *(ptr+2)의 출력결과는 arr[0], arr[1], arr[2]의 출력결과와 동일함을 이미 확인하였다. 그런데 배열의 이름과 포인터 변수는 상수냐 변수냐의 차이만 있을 뿐, 사실상 동일하다. 따라서 ptr에 저장된 값이 arr의 주소 값이기 때문에 다음 네 문장은 사실상 같은 것이며 실제로 네 문장 모두 동일한 출력결과를 보인다.

printf("%d %d %d ₩n", *(ptr+0), *(ptr+1), *(ptr+2)); // *(ptr+0)는 *ptr과 같다.
printf("%d %d %d ₩n", ptr[0], ptr[1], ptr[2]);
printf("%d %d %d ₩n", *(arr+0), *(arr+1), *(arr+2)); // *(arr+0)는 *arr 같다.
printf("%d %d %d ₩n", arr[0], arr[1], arr[2]);

따라서 우리는 다음의 식을 하나 도출할 수 있다.

arr[i] == *(arr+i) // arr[i]는 *(arr+i)와 같다.

위의 식에서 arr은 배열의 이름이어도 성립하고 포인터 변수이어도 성립한다.

3. 상수 형태의 문자열을 가리키는 포인터

마지막에 널 문자가 삽입되는 문자열의 선언방식에는 두 가지가 있다. 하나는 앞서 설명한 배열을 이용하는 방식이다. 그리고 다른 하나는 char형 포인터 변수를 이용하는 방식이다.

- 두 가지 형태의 문자열 표현

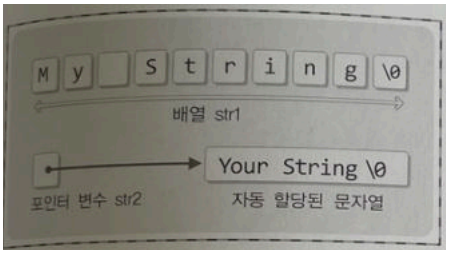
다음과 같이 배열을 기반으로 하는 문자열의 선언은 이미 우리에게 익숙하다.

char str1[ ] = "My String"; // 배열의 길이는 자동으로 계산된다

이는 배열을 기반으로 하는 변수 형태의 문자열 선언이다. 변수라 하는 이유는 문자열의 일부를 변경할 수 있기 때문이다. 반면 다음과 같이 포인터를 기반으로 문자열을 선언하는 것도 가능하다.

char * str2 = "Your String";

이렇게 선언을 하면 메모리 공간에 문자열 "Your String"이 저장되고, 문자열의 첫 번째 문자 Y의 주소값이 반환된다. 그리고 그 반환 값이 포인터 변수 str2 에 저장된다. 그래서 str2 를 char형 포인터로 선언한 것이다. char형 문자 Y의 주소 값이 저장되기 때문이다. 그렇다면 위의 두 문자열 선언의 차이점은 무엇일까? 일단 할당된 메모리의 형태를 비교하면 다음과 같다.



위의 그림에서 보이듯이 str1 은 그 자체로 문자열 전체를 저장하는 배열이고, str2 는 메모리상에 자동으로 저장된 문자열 your string의 첫 번째 문자를 단순히 가리키고만 있는 포인터 변수이다. 그러나 배열이름 str1 이 의미하는 것도 실제로는 문자 m의 주소 값이기 때문에 str1 도 str2 도 문자열의 시작 주소 값을 담고 있다는 측면에서는 동일하다. 다만 다음의 차이가 있을 뿐이다.

"배열이름 str1 은 계속해서 문자 M이 저장된 위치를 가리키는 상태이어야 하지만 포인터 변수 str2 는 다른 위치를 가리킬 수 있다."

str2 는 변수형태의 포인터이기 때문에 다음과 같은 문장을 구성 할 수 있다.

```
int main(void)
{
        char * str = "Your team";
        str = "Our team"; // str이 가리키는 대상을 문자열 our team으로 변경
        . . . .
}
```

하지만 배열이름인 str1 은 상수형태의 포인터이기 때문에 위와 같이 가리키는 대상을 변경할 수 없다. 그리고 또 한가지 중요한 차이점이 있다. 다음과 같이 선언이 되면 애초에 문자열은 배열에 저장된다. 그리고 배열을 대상으로는 값의 변경이 가능하기 때문에 다음과 같이 선언되는 문자열을 가리켜 변수 형태의 문자열이라 한다.

```
char str1[ ] = "My string";
```

반면, 다음과 같이 선언되는 문자열은 상수 형태의 문자열이라 한다.

```
char * str2 = "Your String";
```

그리고 실제로 포인터 변수 str2 가 가리키는 문자열은 그 내용의 변경이 불가능하다. 이와 관련해서 다음 예제를 실행해보자.

```
#include <stdio.h>

int main(void)
{
        char str1[]="My String"; // 변수 형태의 문자열
        char * str2="Your String"; // 상수 형태의 문자열
        printf("%s %s \n", str1, str2);

        str2="Our String"; // 가리키는 대상 변경
        printf("%s %s \n", str1, str2);

        str1[0]='X'; // 문자열 변경 성공
        str2[0]='X'; // 문자열 변경 실패
        printf("%s %s \n", str1, str2);
        return 0;
}
```

위의 예제는 컴파일은 된다. 하지만 실행 시 13 행에서 문제가 발생한다. 왜냐하면 6 행에서 선언한 문자열은 상수 형태의 문자열이기 때문이다. 참고로 문제가 발생하는 형태는 컴파일러에 따라서, 또는 컴파일 모드에 따라서 약간씩 차이가 있다. 예를 들어서 프로그램이 그냥 종료되는 경우도 있고, 실행은 되었으니 13 행의 연산이 무시되어서 문자열이 변경되지 않는 경우도 있다. 그리고 한가지 더 알아야 할 사실이 있다. 그것은 일부 컴파일러는 13 행의 연산을 허용한다는 사실이다. 하지만 그러한 컴파일러를 기반으로 프로그램을 작성할 때에도 13 행과 같은 형태의 문장은 작성하지 않는다. 이는 모든 컴파일러에서 동작하는 코드가 아니기 때문이다. 즉, 컴파일러의 특성에 관계없이 위 예제의 6 행에서 보이는 형태로 선언된 문자열은 상수로 간주하여 그 값을 변경시키지 않아야 한다.

- 어디서든 선언할 수 있는 상수 형태의 문자열

다음과 같이 선언이 되는 문자열은 상수 형태의 문자열이라 하였다,

char * str = "Const String";

그렇다면 어떠한 순서를 거쳐서 이 문장은 처리가 될까? 위의 문장이 실행되면 먼저 문자열이 메모리 공간에 저장된다. 그리고 그 메모리의 주소 값이 반환된다. 즉 문자열이 0x1234 번지에 저장되었다고 가정하면, 위의 문장은 문자열이 저장된 이후에 다음의 형태가 된다.

char * str = 0x1234;

그리하여 포인터 변수 str에는 문자열의 주소 값 0x1234 가 저장되는 것이다. 그렇다면 다음과 같이 함수의 호출과정에서 선언이 되는 문자열은 어떻게 처리가 될까?

printf("Show your string");

이 경우도 마찬가지이다. 큰 따옴표로 묶어서 표현되는 문자열은 그 형태에 상관없이 메모리 공간에 저장된 후 그 주소 값이 반환된다. 따라서 위의 함수호출 문장도 메모리 공간에 문자열이 저장된 이후에 다음의 형태가 된다(문자열이 0x1234 번지에 저장되었다고 가정)

printf(0x1234);

이렇듯 printf 함수는 문자열은 통째로 전달받는 함수가 아닌, 문자열의 주소 값을 전달받는 함수이다.

따라서 다음의 함수 호출문을 보면,

WhoAreYou("Hong"); // WhoAreYou라는 이름의 함수호출

이 함수의 매개변수 선언이 다음과 같음을 짐작할 수 있다(반환형은 void라 가정하였다). 실제로 전달되는 값은 문자 H의 주소 값이기 때문이다.

voud WhoAreYou(char * str) { . . . . }

4. 포인터 변수로 이뤄진 배열: 포인터 배열

지금까지는 기본 자료형의 변수를 요소로 지니는 배열만 선언해왔다. 하지만 포인터 변수도 변수이니 이를 대상으로도 배열을 선언할 수 있다.

- 포인터 배열의 이해

포인터 변수로 이뤄진, 그래서 주소 값의 저장이 가능한 배열을 가리켜 '포인터 배열'이라 한다. 그리고 이러한 배열의 선언방식은 다음과 같다.

int * arr1[20]; // 길이가 20 인 Int형 포인터 배열 arr1

double * arr2[30]; // 길이가 30 인 double형 포인터 배열 arr2

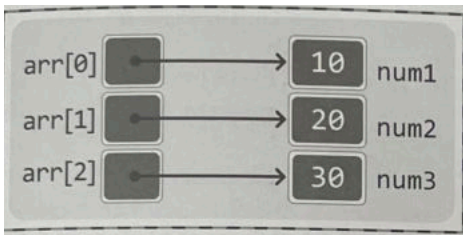
위의 문장에서 보이듯이 포인터 배열의 선언방식은 기본 자료형 배열의 선언방식과 동일하다. 배열의 이름 앞에 배열요소의 자료형 정보를 선언하면 된다. 즉, 배열이름 arr1 의 앞에 선언된 Int * 가 int형 포인터를 의미하고, arr2 의 앞에 선언된 double*가 double형 포인터를 의미한다. 그럼 다음 예제를 통해서 포인터 배열의 선언과 활용을 간단히 보이겠다.

```
#include <stdio.h>

int main(void
{
        int num1=10, num2=20, num3=20;
        int * arr[3] = {&num1, &num2, &num3};

        printf("%d \n", *arr[0]);
        printf("%d \n", *arr[1]);
        printf("%d \n", *arr[2]);
        return 0;
}
```

6 행: int형 포인터 배열을 선언하고 5 행에서 선언한 변수의 주소 값으로 초기화하고 있다.
8~10 행: 배열요소가 가리키는 변수에 저장된 값을 출력하고 있다.

```
10
20
30
```

위 예제의 6 행에 선언된 배열과 5 행에 선언된 변수 num1, num2, num3 의 관계를 그림으로 표현하면 다음과 같다.



이렇듯 포인터 배열도 기본 자료형 배열과 별반 다르지 않다. 다만 주소 값을 지정할 수 있도록 포인터 변수를 대상으로 선언된 배열일 뿐이다.

- 문자열을 저장하는 포인터 배열

이번에는 문자열 배열을 소개하고자 한다. 그런데 이는 문자열의 주소 값을 저장할 수 있는 배열로서 사실상 char형 포인터 배열이다. 즉, 여기서 말하는 문자열 배열은 다음과 같다.

Char * strArr[3]; // 길이가 3 인 char형 배열

위의 선언에서 보이듯이, char형 포인터 배열은 문자열의 주소 값을 저장할 수 있는 배열이다 보니 문자열 배열로 불릴 뿐이다. 그럼 이와 관련해서 다음 예제를 보자.

```
#include <stdio.h>

int main(void)
{
        char * strArr[3]={"Simple", "String", "Array"};
        printf("%s ₩n", strArr[0]);
        printf("%s ₩n", strArr[1]);
        printf("%s ₩n", strArr[2]);
        return 0;
}
```

실행결과:

```
Simple
String
Array
```

위의 예제 5 행에는 다음의 형태로 문자열 배열이 선언되어 있다.

char * strArr[3]={"Simple", "String", "Array"};

"큰따옴표로 묶여서 표현되는 문자열은 그 형태에 상관없이 메모리 공간에 저장된 후 그 주소 값이 반환된다."

즉 위 문장이 실행되면 초기화 리스트에 선언된 문자열들은 메모리 공간에 저장되고, 그 위치에 저장된 문자열의 주소 값이 반환된다. 따라서 문자열이 저장된 이후에는 다음의 형태가 된다.

char * strArr[3]={0x1004, 0x1048, 0x2012}; // 반환된 주소 값은 임의로 결정하였다.

4. 포인터와 함수에 대한 이해

1. 함수의 인자로 배열 전달하기

함수는 인자를 전달받도록 정의할 수 있다. 사실 함수라는 이름이 붙은 이유도 인자의 전달과 값의 반환이 가능하기 때문 아닌가? 본론으로 들어가기에 앞서 인자의 전달원리에 대한 확실한 이해가 필요하다. 따라서 이에 대해 먼저 언급한 다음에 배열 대상의 인자 전달에 이야기하겠다.

- 인자전달의 기본방식은 값의 복사이다.

"함수호출 시 전달되는 인자의 값은 매개변수에 복사가 된다."

위의 문장에서 가장 중요한 단어는 복사이다. 즉, 복사가 되는 것뿐이기 때문에 함수가 호출하고 나면, 전달되는 인자와 매개변수는 별개가 된다. 이와 관련해서 다음 코드를 보자.

```
int SimpleFunc(int num) { . . . . }
int main(void)
{
        int age=17;
        SimpleFunc(age); // age에 저장된 값이 매개변수 num에 복사됨
        . . . .
}
```

위 코드의 simplefunc함수의 호출을 통해서 인자로 age를 전달하고 있다. 그러나 실제로 전달되는 것은 age가 아닌, age에 저장된 값이다. 그리고 그 값이 매개변수 num에 복사되는 것이다. 따라서 age와 num은 값을 주고받은 사이일 뿐 그 이상은 아무런 관계도 아니다. 그럼 다음 질문에 답을 해보자.

"SimpleFunc 함수 내에서 매개변수 num에 저장된 값을 1 증가시킬 경우, 변수 age의 값은 얼마가 증가하는가?"

매개변수 num의 값이 증가했다고 하여 변수 age의 값이 변화될 리 만무하다. num과 age는 별개의 변수이기 때문이다.

이어서 함수호출 시 인자로 배열을 통째로 전달하는 방법에 대해서 생각해보자. 하지만 애석하게도 함수를 호출하면서 매개변수에 배열을 통째로 넘겨주는 방법은 존재하지 않는다. 왜냐하면 매개변수로 배열을 선언할 수 없기 때문이다. 배열을 통째로 넘겨받으려면 매개변수로 배열을 선언할 수 있어야 한다. 하지만 이것이 허용되지 않으니 배열을 통째로 넘기는 것은 불가능한 일이다. 대신에 함수 내에서 배열에 접근할 수 있도록 배열의 주소 값을 전달하는 것은 가능하다.

- 배열을 함수의 인자로 전달하는 방식

아파트를 보고 싶어 하는 사람 앞에 아파트를 통째로 복사해다 놓을 수 없다면, 아파트의 주소를 가르쳐줘서 직접 찾아가게 하면 된다. 이와 유사하게 배열을 통째로 전달하는 것이 불가능하다면, 배열의 주소 값을 인자로 전달해서 이를 통해서 접근하도록 유도하는 방법을 생각해볼 수 있다. 예를 들어서 다음과 같이 선언된 배열이 있다면,

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

다음의 형태로 함수를 호출하면서 배열의 주소 값을 전달할 수 있다.

```
simplefunc(arr); // simplefunc 함수를 호출화면서 배열 arr의 주소 값 전달
```

그렇다면 simplefunc 함수의 매개변수는 어떻게 선언되어야 하겠는가?

```
int main(void)
{
        int arr[3]={1,2,3};
        int * ptr=arr; // 배열이름 arr은 int형 포인터
        . . . .
}
```

따라서 simplefunc함수의 매개변수는 다음과 같이 int형 포인터 변수로 선언되어야 한다.

```
void simplefunc(int * param) { . . . . }
```

그럼 매개변수 param을 이용해서 배열에는 어떻게 접근해야 하는가? 이와 관련해서는 앞서 이미 공부하였다. 포인터 변수를 이용해도 배열의 형태로 접근이 가능하니, 다음과 같이 접근이 가능하다고 이미 공부하였다.

```
printf("%d %d", param[1], param[2]); // 두 번째, 세 번째 요소 출력
```

자, 그럼 지금까지 설명한 내용을 모아서 하나의 예제로 완성해 보이겠다.

```
#include <stdio.h>

void sap(int * param, int len)
{
        int i;
        for(i=0; i<len; i++)
                printf("%d", param[i]);
        printf("₩n");
}

int main(void)
{
        int arr1[3]={1, 2, 3};
        int arr2[5]={4, 5, 6, 7, 8};
        sap(arr1, sizeof(arr1) / sizeof(int));
        sap(arr2, sizeof(arr2) / sizeof(int));
        return 0;
}
```

3 행: 두 번째 인자로 배열의 길이정보를 전달받도록 정의
7 행: int형 포인터 변수의 이름을 대상으로 배열형태의 접근을 진행하고 있다.
15, 16 행: 길이가 다른 두 배열을 대상으로 sap함수를 호출하고 있다.

실행결과:

```
1 2 3
4 5 6 7 8
```

위 예제에서는 sap함수 내에서 외부에 선언된 배열에 접근하여 그 값을 출력하였다. 그렇다면 값의 출력이 아닌, 값의 변경도 가능하겠는가? 물론 가능하다. 주소 값만 알면 해당 메모리 공간에 접근이 가능하기 때문이다. 그럼 이와 관련해서 다음 예제를 제시하겠다.

```
#include <stdio.h>

void show(int * param, int len)
{
        int i;
        for(i=0; i<len; i++)
                printf("%d", param[i]);
        printf("₩n");
}

void add(int * param, int len, int add)
{
        int 1;
        for(i=0; i<len; i++)
                param[i] += add;
}

int main(void)
{
        int att[3]={1, 2, 3};
        add(arr, sizeof(arr) / sizeof(int), 1);
        show(arr, sizeof(arr) / sizeof(int));

        add(arr, sizeof(arr) / sizeof(int), 2);
        show(arr, sizeof(arr) / sizeof(int));

        add(arr, sizeof(arr) / sizeof(int), 3);
        show(arr, sizeof(arr) / sizeof(int));
        return 0;
}
```

실행결과:

2 3 4

4 5 6

7 8 9

위 예제 7 행에서는 20 행에 선언된 배열에 저장된 값을 참조만 하는 방면 15 행에서는 배열에 저장된 값을 변경하고 있다. 이렇듯 배열의 주소 값만 안다면 어디서든 배열에 접근하여 저장된 값을 참조하고 변경할 수 있다.

- 배열을 함수의 인자로 전달받는 함수의 또 다른 선언

앞서 보인 예제의 다음 두 함수에는 int형 배열의 주소 값을 인자로 전달할 수 있도록 int형 포인터 변수가 선언되어 있다.

void show(int * param, int len)

void add(int * param, int len, int add)

그런데 이를 대신하여 다음과 같이 선언하는 것도 가능하다.

void show(int param[], int len)

void add(int param[], int len, int add)

즉, int param[]과 int *param은 완전히 동일한 선언이다. 그런데 전자의 선언이, 배열의 인자로 전달된다는 느낌을 더 강하게 주는 선언이다. 따라서 일반적으로 배열의 주소 값이 인자로 전달될 때에는 int param[] 형태의 선언을 주로 많이 사용한다. 하지만 이 둘이 같은 선언으로 간주되는 경우는 매개변수의 선언으로 제한된다. 따라서 다음의 코드에서,

```
int main(void)
{
        int arr[3]={1, 2, 3};
        int * ptr=arr; // int ptr[]=arr; 로 대체 불가능
        . . . .
}
```

main 함수의 두 번째 문장인 int * ptr=arr을 int ptr[]=arr로 대체할 수 없다.

2. Call-by-value vs Call-by-reference

call-by-value와 call-by-reference는 함수의 호출방식을 의미한다.

- 값을 전달하는 형태의 함수호출: call-by-value

함수를 호출할 때 단순히 값을 전달하는 형태의 함수호출을 가리켜 call-by-value라 하고, 메모리의 접근에 사용되는 주소 값을 전달하는 형태의 함수호출을 가리켜 call-by-reference라 한다. 즉, call-by-value와 call-by-reference를 구분하는 기준은 함수의 인자로 전달되는 대상에 있다.

지금까지 우리가 정의한 함수는 대부분 call-by-value였다. 하지만 앞서 정의한 다음 함수는 call-by-reference이었다.

```
void sae(int * param, int len)
{
        int I;
        for(i=0; i<len; i++)
                printf("%d", param[i]);
        printf("₩n");
}
```

이 함수는 첫 번째 인자로 배열의 주소 값을 전달받도록 정의되었으니, 이 함수의 호출 형태는 call-by-reference이다.

call-by-value와 call-by-reference를 구분하는 이유는 다음과 같은 실수를 막기 위함이다. 다음 예제의 문제점을 보자.

```
#include <stdio.h>

void swap(int
n1, int n2)
{
        int temp=n1;
        n1=n2;
        n2=temp;
        printf("n1 n2: %d %d ₩n", n1, n2);
}

int main(void)
{
        int num1=10;
        int num2=20;
        printf("num1 num2: %d %d ₩n", num1, num2);

        swap(num1, num2); // num1 과 num2 에 저장된 값이 서로 바뀌길 기대
        printf("num1 num2: %d %d ₩n", num1, num2);
        return 0;
}
```

실행결과:
num1 num2: 10 20
n1 n2: 20 10
num1 num2: 10 20

위 예제 17 행의 주석에 쓰여있는 대로, 17 행의 함수호출 결과로 num1 과 num2 에 저장된 값이 변경되기를 기대하면서 swap 함수를 호출하고 있으며, 그 결과로 num1 과 num2 에 저장된 값이 매개변수 n1 과 n2 에 복사된다. 그리고 swap 함수 내에서는 실제로 다음 그림에서 보이는 과정을 거쳐서 n1 과 n2 에 저장된 값의 변경이 일어난다. 이는 위의 출력결과를 통해서도 확인이 가능하다.
하지만 이는 매개변수 n1 과 n2 에 저장된 값을 변경시키는 것일 뿐, num1 과 num2 에 저장된 값의 변경으로까지 이어지지는 않는다. num1, num2 는 n1, n2 와 완전히 별개이기 때문이다. 결국 두 매개변수 사이에서만 값의 반환이 일어난 것이다.

- 주소 값을 전달하는 형태의 함수호출: call-by-reference
그렇다면 앞서 보인 예제를 어떻게 변경해야만 원하는 결과를 얻을 수 있겠는가? 다시 말해서 어떻게 해야 swap함수의 호출결과로 num1 과 num2 의 값을 서로 바꿀 수 있겠는가? 답은 call-by-reference 형태의 함수 정의에 있다. 즉, num1 과 num2 의 주소 값을 swap 함수로 전달해서 swap 함수 내에서 num1 과 num2 에 직접 접근이 가능하도록 하는 것이다. 그럼 이에 대한 예제를 제시하겠다.

```
#include <stdio.h>

void swap(int * ptr1, int * ptr2)
{
        int temp = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 = temp;
}

int main(void)
{
        int num1=10;
        int num2=20;
        printf("num1 num2: %d %d ₩n", num1, num2);
        swap(&num1, &num2);
        printf("num1 num2: %d %d ₩n", num1, num2);
        return 0;
}
```

실행결과:
num1 num2: 10 20
num1 num2: 20 10

실행결과를 통해서 이번에는 원하는 바가 이뤄졌음을 확인할 수 있다. 그럼 위 예제에서 가장 핵심이 되는 swap함수를 살펴보자.

```
void swap(int * ptr1, int * ptr2)
{
        int temp = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 = temp;
}
```

위 함수는 변수의 주소 값을 인자로 받아서 해당 변수에 직접 접근하는 형태를 띤다. 따라서 main 함수가 다음과 같이 실행되면서 swap 함수가 호출되면,

```
int main(void)
{
        int num1=10;
        int num2=20;
        . . . .
        swap(&num1, &num2);
        . . . .
}
```

swap 함수의 매개변수인 ptr1 과 ptr2 는 각각 num1 과 num2 를 가리키는 상황이 된다. 따라서 *ptr1 은 num1 을

의미하게 되고, *ptr2 는 num2 를 의미하게 되어 결과적으로 num1 에 저장된 값과 num2 에 저장된 값이 서로 바뀐다.

- 이제 scanf 함수호출 시 & 연산자를 붙이는 이유를 알 수 있다.

위의 제목처럼 이제는 scanf 함수 호출 시 & 연산자를 붙이는 이유를 알 수 있을 것이다. 이와 관련해서 다음 코드를 보자.

```
int main(void)
{
        int num;
        scanf("%d", &num); // 변수 num의 주소 값을 scanf 함수에 전달
        . . . .
}
```

위의 scanf 함수호출이 완료되면 변수 num에는 값이 채워진다. 즉 프로그램 사용자로부터 값을 입력 받아서 변수 num에 그 값을 채우는 일을 scanf 함수가 하는 것이다. 그리고 이를 위해서 scanf 함수는 변수 num의 주소 값을 알아야 한다. 그래야 변수 num에 접근을 해서 값을 채워 넣을 수 있기 때문이다. 그래서 scanf 함수호출 시에 변수 num의 주소 값을 전달하고 있다. 이제 변수 num의 앞에 & 연산자를 붙이는 이유를 알겠는가? 이렇듯 scanf 함수의 호출도 call-by-reference 형태의 함수호출에 해당한다. 그렇다면 문자열을 입력 받을 때에는 &연산자를 붙여주지 않았던 것일까? 이와 관련해서 다음 코드를 보자.

```
int main(void)
{
        char str[30];
        scanf("%s", str); // scanf("%s", &str); 는 잘못된 문장 구성이다.
        . . . .
}
```

이전에 보였듯이 문자열을 위와 같은 방식으로 입력 받는다. 그리고 문자열이 저장될 배열의 이름 str의 앞에는 &연산자를 붙이지 않는다. 왜냐하면 str은 그 자체로 배열의 주소 값이기 때문이다. 따라서 &연산자를 붙일 이유가 없다.

3. 포인터 대상의 const 선언

const 선언은 변수를 상수화하는 목적이 있다.

- 포인터 변수가 참조하는 대상의 변경을 허용하지 않는 const 선언

아래의 코드에서 보이는 바와 같이 포인터 변수 ptr을 대상으로도 const 선언을 할 수 있다.

```
int main(void)
{
        int num = 20;
        const int * ptr=&num;
        *ptr=30; // 컴파일 에러
        num=40; // 컴파일 성공
        . . . .
}
```

위의 const 선언에서 여러분이 주의 깊게 봐야할 것은 const의 선언 위치이다. 이렇듯 맨 앞부분에서 선언이 되면, 포인터 변수 ptr을 대상으로 다음의 의미가 담겨진다.

"포인터 변수 ptr을 이용해서 ptr이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않는다."
때문에 이어서 등장하는 다음 문장에 컴파일 에러가 발생한다.

```
*ptr=30; // 컴파일 에러
```

그렇다고 해서 포인터 변수 ptr이 가리키는 변수 num이 상수화되는 것은 아니다. 따라서 다음과 같이 변수 num에 저장된 값을 변경하는 것은 허용이 된다.

```
num=40;
```

이렇듯 위의 const 선언은 값을 변경하는 방법에 제한을 두는 것이지 무엇인가를 상수로 만드는 선언은 아니다.

- 포인터 변수의 상수화

아래의 문장에서 보이듯이 const선언은 포인터 변수의 이름 앞에 올 수도 있다.

```
int * const ptr = &num;
```

그리고 이렇게 되면 포인터 변수 ptr은 상수가 된다. 포인터 변수 ptr이 상수라는 뜻은 한번 주소 값이 저장되면 그 값의 변경이 불가능하다는 뜻이며, 이는 한번 가리키기 시작한 변수를 끝까지 가리켜야 한다는 뜻으로도 이해할 수 있다. 이와 관련해서 다음 코드를 보자.

```
int main(void)
{
        int num1=20;
        int num2=30;
        int * const ptr=&num1;
        ptr=&num2; // 컴파일 에러
        *ptr=40;
        . . . .
}
```

위의 코드를 보면, 상수화된 포인터 변수 ptr이 num1 을 가리키고 잇다. 그런데 그 다음 행에서, 가리키는 대상을 num2 로 바꾸기 위한 연산을 진행하고 있다. 하지만 ptr은 상수이기 때문에 이 부분에서 컴파일 에러가 발생한다. 물론 ptr이 상수일 뿐이니, 다음과 같이 ptr이 가리키는 대상에 저장된 값을 변경하는 연산은 문제가 되지 않는다.

```
*ptr=40;
```

지금까지 포인터 변수를 대상으로 하는 두 가지 형태의 const 선언에 대해서 설명하였는데, 다음과 같이 하나의 포인터 변수를 대상으로 이 두가지 형태의 const 선언을 동시에 할 수도 있다.

```
const int * const ptr=&num;
```

그리고 이렇게 선언이 되면 맨 앞에 const 선언으로 인해서 다음의 연산이 불가능해지고,

```
*ptr=40; // 컴파일 에러
```

포인터 변수 ptr앞의 const선언으로 인해서 다음의 연산도 동시에 불가능해진다.

```
ptr=&age;
```

- const 선언이 갖는 의미

const선언이 특별한 기능을 제공하는 것은 아니기 때문에 그 중요성을 인식하지 못하는 경우가 많다. 반면 if문이나 for문은 제공하는 고유의 기능이 있기 때문에 중요하게 여긴다. 사실 const는 처음부터 c언어에 존재하던 키워드는 아니다. const는 c++에만 존재하던 키워드였는데, c언어의 표준을 재정립하는 과정에서 c언어의 일부가 된 것이다. const 선언을 많이 하면 그만큼 프로그램 코드의 안정성은 높아진다. 이와 관련해서 다음 코드를 보자.

```
int main(void)
{
        double PI=3.1415;
        double rad;
        PI=3.07; // 실수로 잘못 삽입된 문장, 컴파일 시 발견 안됨
        scanf("&lf", &rad);
        printf("circle area %f ₩n", rad*rad*PI);
        return 0;
}
```

위의 main 함수에서는 원의 넓이를 계산하여 출력한다. 그런데 원주율에 해당하는 값이 저장된 변수 PI의 값을 변경하는 실수를 범하고 있다. 물론 이 프로그램을 작성한 사람도 PI에 저장된 값을 변경하면 안 된다는 것은 알고 있을 것이다. 그럼에도 불구하고 이러한 실수가 발생하였다.
하지만 이보다 더 큰 문제는 컴파일러가 이러한 문제점을 발견하지 못한다는게 있다. 이것이 왜 큰 문제인가? 컴파일러가 발견하지 못하는 오류의 상황은 쉽게 발견되지 않기 때문이다. 혹 실행결과를 통해서 문제가 있다고 판단을 했어도 무엇이 문제인지 쉽게 찾아낼 수가 없다. 예를 들어서 약 8천 라인 정도되는 프로그램을 작성하고 컴파일까지 완료한 상태에서 실행을 해보니 그 결과가 조금 이상하다면 어디서부터 시작해서 문제를 찾아낼 것인가? 바로 이러한 문제점 때문에 현명한 프로그래머는 다음과 같이 코드를 작성한다.

```
int main(void)
{
        const double PI=3.1415;
        double rad;
        PI=3.07; // 컴파일 시 발견되는 오류상황
        scanf("&lf", &rad);
        printf("circle area %f ₩n", rad*rad*PI);
        return 0;
}
```

앞서 보인 코드와의 유일한 차이점은 변수 PI를 대상으로 하는 const 선언이다. 하지만 이로 인해서 PI에 대한 안전성은 확보가 되었다. PI에 저장된 값이 변경되는 오류를 실행파일이 생성되기 전에 막을 수 있으니, 그만큼 코드의 안정성이 높아진 것이다.