# 포인터와 배열

## - 배열 이름과 포인터, 포인터 연산, 포인터 배열 -

성공회대학교 IT융합자율학부
소프트웨어공학전공
홍 성 준

# 포인터와 배열의 관계

## ◎ 배열 이름과 포인터의 관계

- 배열 이름은 메모리 상에서 배열이 시작되는 주소 값을 나타내는 포인터 상수
  - 한 번 선언된 배열의 위치(주소)는 변경할 수 없음
- 배열 이름에 * 연산자를 사용하여 메모리에 접근 가능
- ArrayNameType.c

```
int main(void)
{
    int arr[3]={0, 1, 2};
    printf("배열의 이름: %p \n", arr);
    printf("첫 번째 요소: %p \n", &arr[0]);
    printf("두 번째 요소: %p \n", &arr[1]);
    printf("세 번째 요소: %p \n", &arr[2]);
    // arr = &arr[i];  // 이 문장은 컴파일 에러를 일으킨다.
    return 0;
}
```

```
배열의 이름: 0012FF50
첫 번째 요소: 0012FF50
두 번째 요소: 0012FF54
세 번째 요소: 0012FF58
```


- 배열의 모든 원소는 메모리 공간에 연속적으로 할당됨
- int 형 배열 요소 간의 주소 값의 차이는 4바이트

# 1차원 배열 이름의 포인터 형

◎ 1차원 배열 이름의 포인터 형을 결정하는 방법

- 배열 이름이 가리키는 변수의 자료 형을 근거로 판단

· int형 변수를 가리키면 int * 형

· double형 변수를 가리키면 double * 형

int arr1[5]; 에서 arr1은 int * 형

double arr2[7]; 에서 arr2는 double * 형

```
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;
}
```

```
1 1.1
101 121.6
```

arr1이 int형 포인터이므로 * 연산의 결과로 4바이트 메모리 공간에 정수를 저장

arr2는 double형 포인터이므로 * 연산의 결과로 8바이트 메모리 공간에 실수를 저장

# 포인터를 사용하여 배열의 원소 접근하기

◎ 배열 이름과 포인터 변수의 값은 모두 포인터

- 배열 이름과 포인터 변수 모두 주소 값을 다루기 때문에 포인터 변수에 정의된 연산은 배열 이름에도 정의 되어 있고, 배열 이름으로 할 수 있는 연산은 포인터 변수로도 연산 가능함

◎ ArrayNameIsPointer.c

- 포인터 변수를 이용하여 배열의 형태로 메모리 공간에 접근하기

```
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;
}
```

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

# 포인터 연산

◎ 포인터를 대상으로 하는 증가 연산과 감소 연산

- 포인터 변수에 저장된 값을 대상으로 하여 증가 연산과 감소 연산을 진행할 수 있음
- int 형 포인터 변수에 n 만큼 증감 연산을 하면 n * sizeof(int)의 크기 만큼 증감
- double 형 포인터 변수에 n 만큼 증감 연산을 하면 n * sizeof(double)의 크기 만큼 증감

◎ PointerOperationResult.c

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

    printf("%p %p \n", ptr1+1, ptr1+2);
    printf("%p %p \n", ptr2+1, ptr2+2);

    printf("%p %p \n", ptr1, ptr2);
    ptr1++;
    ptr2++;
    printf("%p %p \n", ptr1, ptr2);
    return 0;
}
```

▶ **type형 포인터 변수 대상으로 n의 크기 만큼 증감 시,**
**n * sizeof(type)의 크기만큼 값이 증감**

```
00000014 00000018
00000018 00000020
00000010 00000010
00000014 00000018
```

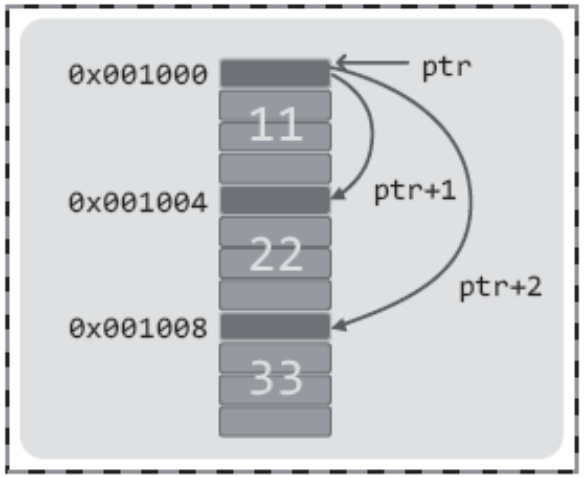
# 포인터 연산

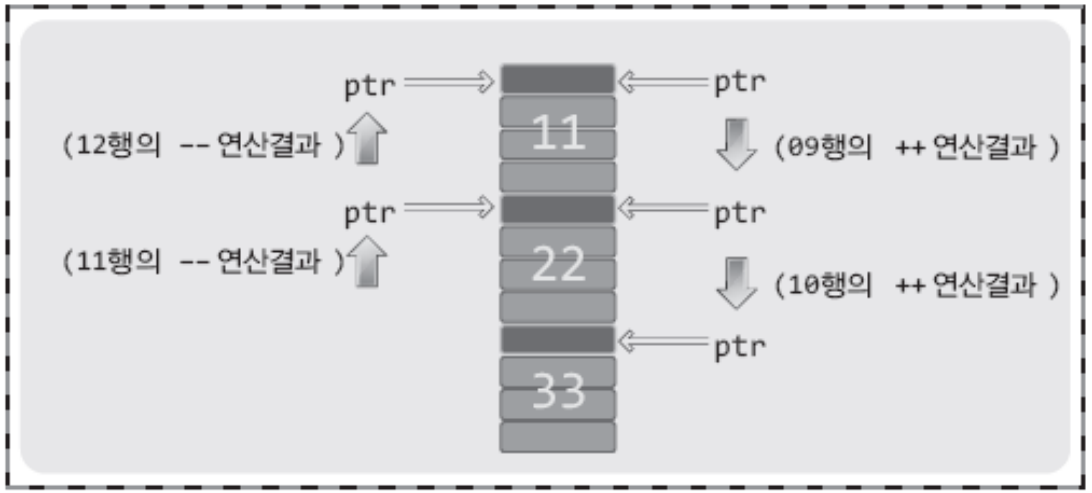
◎ PointerBaseArrayAccess.c

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

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



# 포인터 연산

◎ arr[i] == *(arr + i)

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

```
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 *arr과 같다.
printf("%d %d %d \n", arr[0], arr[1], arr[2]);
```

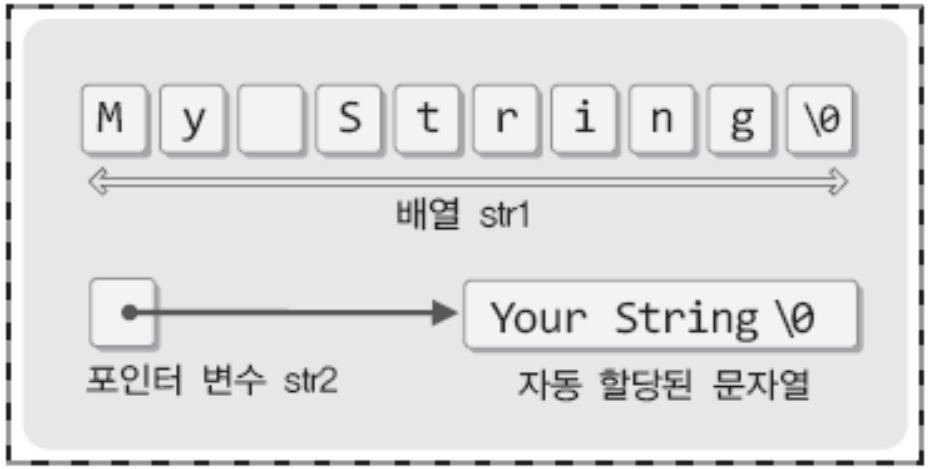
# 문자열의 형태

## ◎ 변수 형태의 문자열

- 배열을 기반으로 문자열을 선언

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

- 문자열의 일부를 다른 문자로 변경 가능
- 배열 이름은 포인터 상수이기 때문에 가리키는 대상은 바꿀 수 없음




## ◎ 상수 형태의 문자열

- 메모리에 있는 문자열 상수를 포인터가 가리키는 형태

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

- 메모리 공간에 문자열이 저장되고, 문자열의 첫번째 문자의 주소를 포인터가 저장
- 포인터 변수가 가리키는 대상은 언제든지 변경 가능함

# 문자열의 형태

◎ TwoStringType.c

```
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;
}
```

- str2[0] = 'x';
  - 컴파일러에 따라 컴파일 오류가 발생하거나, 연산이 무시되거나, 런타임 오류가 발생함

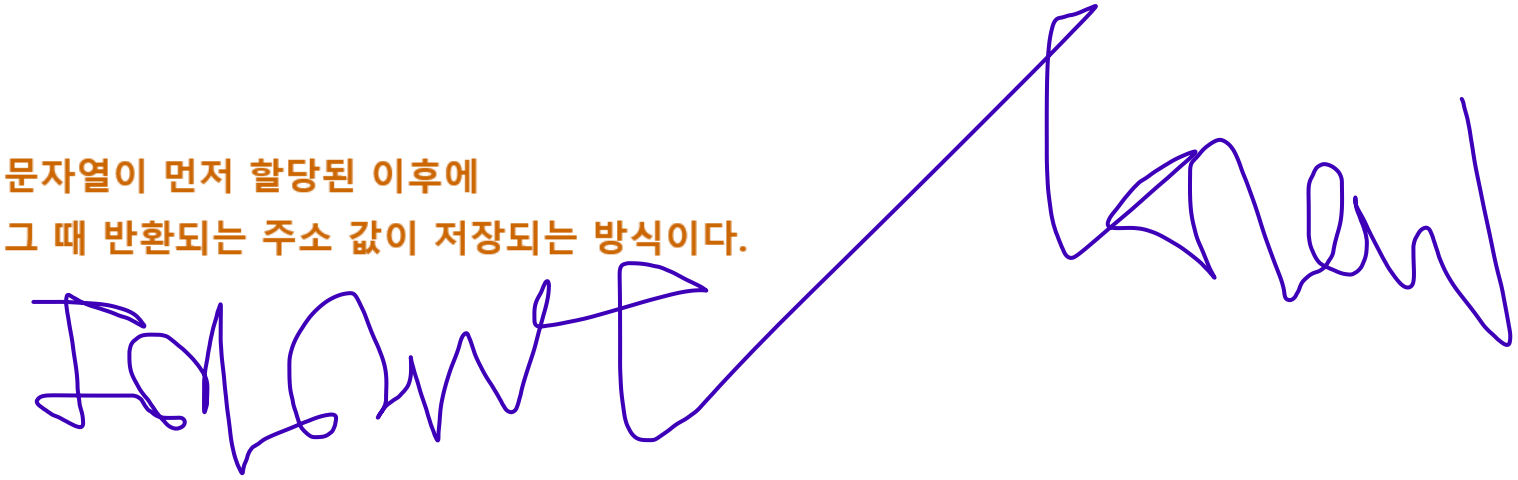
# 상수 형태의 문자열의 처리 과정

```
char * str = "Const String";
```

문자열 저장 후 주소 값 반환

```
char * str = 0x1234;
```

문자열이 먼저 할당된 이후에
그 때 반환되는 주소 값이 저장되는 방식이다.



```
printf("Show your string");
```

문자열 저장 후 주소 값 반환

```
printf(0x1234);
```

위와 동일하다.
문자열은 선언 된 위치로 주소 값이 반환된다.

```
WhoAreYou("Hong");
```

문자열을 전달받는 함수의 선언

```
void WhoAreYou(char * str) { . . . . }
```

문자열의 전달만 보더라도
함수의 매개변수 형(type)을 짐작할 수 있다.

# 포인터 배열의 이해

## ◎ 포인터 배열

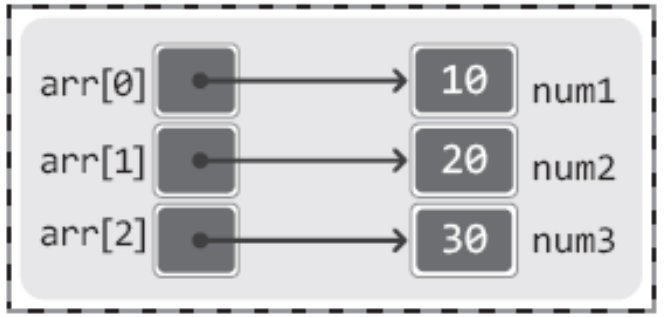
- 주소 값을 저장할 수 있는 포인터 변수를 원소로 갖는 배열
- 선언 방식

```
int * arr1[20];    // 길이가 20인 int형 포인터 배열 arr1
double * arr2[30];     // 길이가 30인 double형 포인터 배열 arr2
```

```
int main(void)
{
    int num1=10, num2=20, num3=30;
    int* arr[3]={&num1, &num2, &num3};

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

```
10
20
30
```

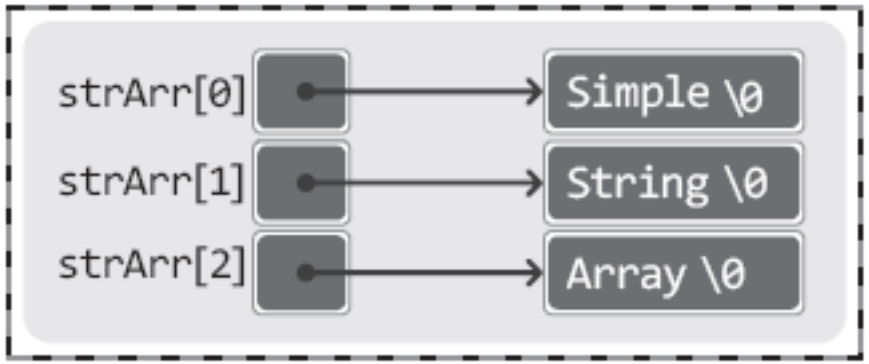


# 포인터 배열의 이해

◎ 문자열을 저장하는 포인터 배열

```
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;
}
```

```
Simple
String
Array
```




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

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