프로그래밍 1
Lecture Note #13

백윤철
ybaek@smu.ac.kr

내용

- 포인터와 배열
- 포인터로 문자열 표현
- 포인터 배열

배열의 이름은 무엇을 의미하는 가?

- 아래의 예제에서 보이듯이, 배열의 이름은 배열의 시작 주소 값
- 메모리 접근에 사용되는 *연산 사용 가능

```
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]; 컴파일 에러를 발생시킴 */
  return 0;
}
```

배열의 이름은 변수가 아닌 상수 형태의 포인터이기에 대입연산이 불가능하다.

배열의 이름은 무엇을 의미하는 가?

실행결과

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

배열 요소간 주소 값의 크기는 4바이트임을 알 수 있다(모든 요소가 붙어있다는 의미).

배열의 이름은 무엇을 의미하는 가?

- 배열 이름과 포인터 변수의 비교

비교조건 \ 비교대상	포인터 변수	배열의 이름
이름이 존재하는가?	존재한다	존재한다
무엇을 나타내거나 저장하는가?	메모리의 주소 값	메모리의 주소 값
주소 값의 변경이 가능한가?	가능하다	불가능하다.

1차원 배열 이름의 포인터 형

- 1차원 배열 이름의 포인터 형 결정하는 방법
  - 배열의 이름이 가리키는 변수의 자료형을 근거로 판단
  - int형 변수를 가리키면 int *
  - double형 변수를 가리키면 double *
  - int arr1[5];에서 arr1은 ~~int *~~ 주소
  - double arr2[7];에서 arr2는 ~~double *~~

1차원 배열 이름의 포인터 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;
}
```

배열 이름을 대상으로 포인터 연산을 하고 있음에 주목!

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

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

실행결과

```
1 1.1
101 121.6
```

포인터를 배열의 이름처럼 사용할 수 있음

```
int main(void) {
  int arr[3] = { 1, 2, 3 };
  arr[0] += 5;
  arr[1] += 7;
  arr[2] += 9;
  return 0;
}
```

- arr은 int형을 가르키는 주소값. 배열 접근을 위한 [idx]연산을 진행한 셈
- 실제로 포인터 변수 ptr을 대상으로 ptr[0], ptr[1], ptr[2]와 같은 방식으로 메모리 공간에 접근 가능

포인터를 배열의 이름처럼 사용할 수 있음

- 포인터 변수를 이용해서 배열의 형태로 메모리 공간에 접근하고 있음

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

실행결과

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

포인터를 대상으로 하는 증가 및 감소 연산

```
int main(void) {
  int i; double d;
  int *ptr1 = &i;
  double *ptr2 = &d;
  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;
}
```

포인터를 대상으로 하는 증가 및 감소 연산

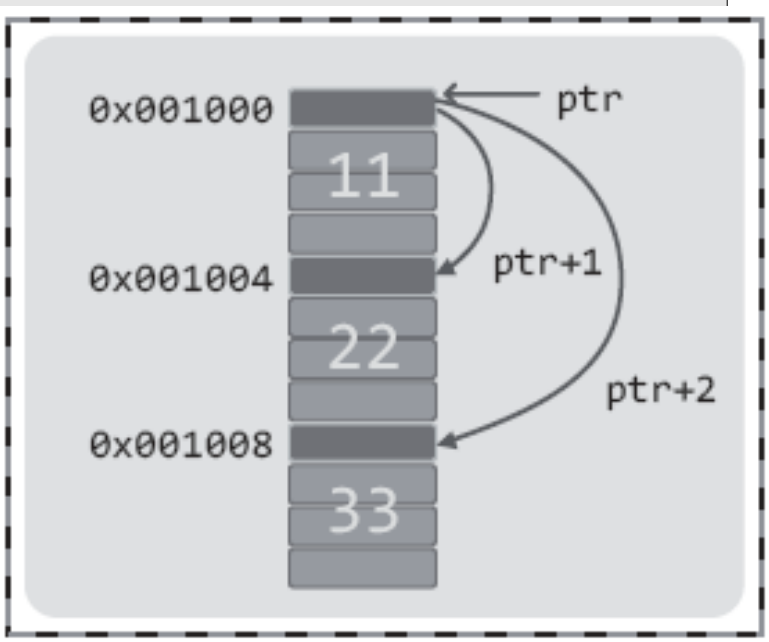
- 예제의 실행결과를 통해서 다음을 알 수 있음
  - int형 포인터 변수 대상의 증가 감소 연산 시 sizeof(int)의 크기만큼 값이 증가 및 감소
  - double형 포인터 변수 대상의 증가 감소 연산 시 sizeof(double)의 크기만큼 값이 증가 및 감소

일반화

  - TYPE형 포인터 변수 대상의 증가 감소 연산 시 sizeof(type)의 크기만큼 값이 증가 및 감소

포인터를 대상으로 하는 증가 및 감소 연산

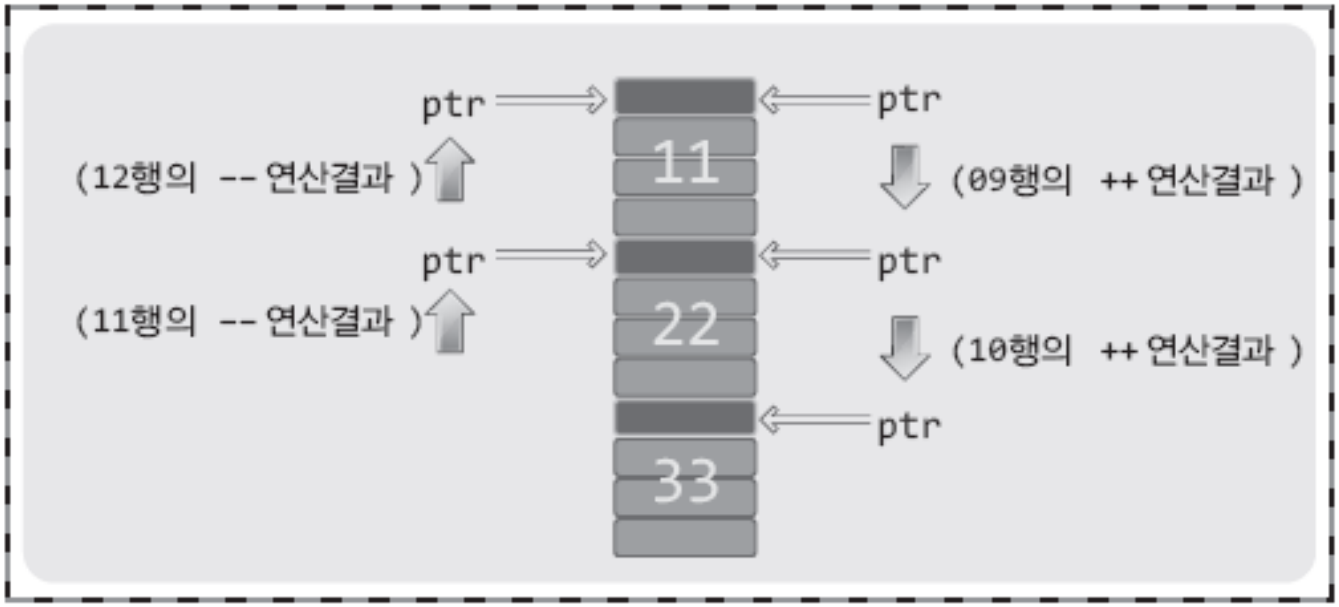
```
int main(void) {
  int arr[3] = { 11, 22, 33 };
  int *ptr = arr; /* int *ptr = &arr[0]; */
  printf("%d %d %d\n", *ptr, *(ptr+1), *(ptr+2));
  printf("%d ", *ptr); ptr++;
  printf("%d ", *ptr); ptr++;
  printf("%d ", *ptr); ptr--;
  printf("%d ", *ptr); ptr--;
  printf("%d\n", *ptr);
  return 0;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



실행결과

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

포인터를 대상으로 하는 증가 및 감소 연산



- int형 포인터 변수의 값은 4씩 증가 및 감소하니 int형 포인터 변수가 int형 배열을 가리키면, int형 포인터 변수의 값을 증가 및 감소시켜서 배열 요소에 순차적으로 접근 가능

중요한 결론! 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));
  …
}
```

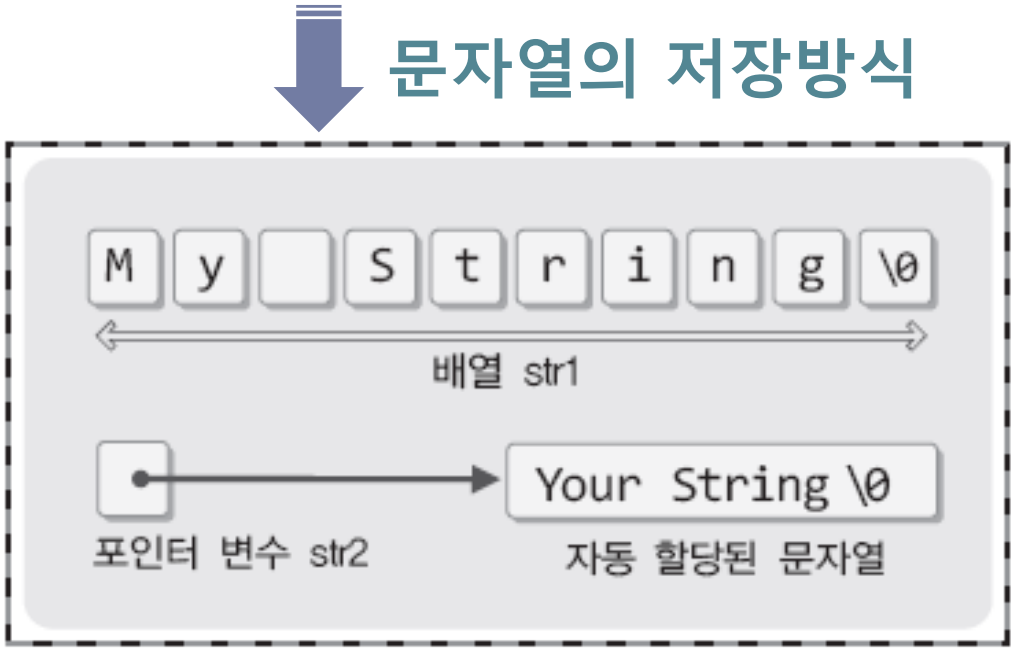
중요한 결론! arr[i] == *(arr + i)

- 배열 이름은?
- 따라서 포인터 변수를 이용한 배열의 접근 방식을 배열의 이름을 이용해서 사용 가능
- 배열의 이름을 이용한 접근방식도 포인터 변수를 대상으로 사용 가능

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

두 가지 형태의 문자열 표현

```
char str1[] = "My string";
char *str2 = "Your string";
```



- str1은 문자열이 저장된 배열
- str2는 문자열의 주소 값을 저장 (자동 할당된 문자열의 주소 값 저장)

두 가지 형태의 문자열 표현

```
int main(void) {
   char *str = "Your team";
   str = "Our team"; /* 의미 있음 */
   …
}

int main(void) {
   char str[] = "Your team";
   str = "Our team"; /* 의미 없음 */
   …
}
```

두 가지 형태의 문자열 표현

```
int main(void) {
   char str1[] = "My string";/* str1 불변, 문자 가변*/
   char* str2 = "Your string"; /* str2 가변, 문자 불변*/
   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;
}
```

str1에 저장된 변수 성향의 문자열은 변경이 가능!
반면 str2에 저장된 상수 성향의 문자열은 변경이 불가능!
간혹 상수 성향의 문자열 변경도 허용하는 컴파일러가 있으나, 이러한 형태의 변경은 바람직하지 못하다!

어디서든 정의할 수 있는 상수 형태의 문자열

```
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 정의

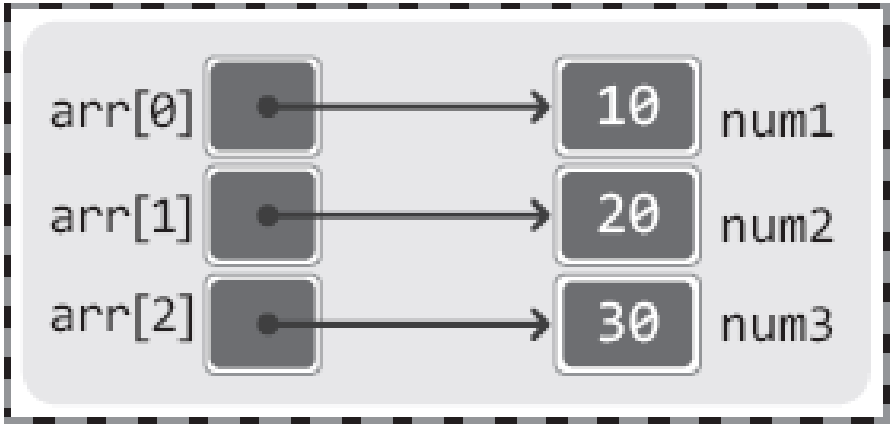
```
void WhoAreYou(char *str) {
…
}
```

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

포인터 배열의 이해

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

```
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;
}
```



실행결과

```
10
20
30
```

포인터 배열의 이해

- 포인터 배열이라 해서 일반 배열의 정의와 다를 바 없음
- 변수의 자료형을 표시하는 위치에 int나 double을 대신해서 int *나 double *이 존재함

```
typedef int * INTP; /* 새로운 형을 생성 */
INTP arr1[20];
typedef double * DOUBLEP; /* 새로운 형을 생성 */
DOUBLEP arr2[30];
```

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;
}
```

실행결과

```
Simple
String
Array
```

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

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