시스템 프로그래밍 기초

Introduction to System Programming

ICT융합학부 조용우

6. Arrays, Pointers, and Strings

배열(Arrays)

- 같은 자료형의 자료를 여러 개 생성
- (예) 성적처리를 위한 변수 선언

```
int grade0, grade1, grade2 ;
int grade[3] ; /* grade[0], grade[1], grade[2] */
```

→ 3:배열 원소의 개수, 배열 원소의 첨자는 0부터 시작

```
type var_name[size]
```

→ type: 기본 자료형 + 사용자 정의 자료형

→ size: 상수식

배열의 크기

```
#define SIZE 100

int a[SIZE] ; /* space for a[0], ..., a[99]
                 is allocated*/
```

- lower bound = 0
- upper bound = SIZE - 1
- SIZE = upper bound + 1

배열 초기화(Initialization)

- *one_dimensional_array_initializer* ::= { *initializer_list* }
- *initializer_list* ::= *initializer* { , *initializer* }$_{0+}$
- *initializer* ::= *constant_integral_expression*

배열 초기화(Initialization)

- 1차원배열 초기화

```
float f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
```

→ f[0] = 0.0
→ f[1] = 1.0
→ f[2] = 2.0
→ f[3] = 3.0
→ f[4] = 4.0

배열 초기화(Initialization)

- 초기화 리스트가 배열원소 개수보다 적으면 나머지는 0으로 초기화

```
int a[100] = {10};
```

→ a[0] = 10
→ a[1] = 0
→ a[2] = 0
...
→ a[99] = 0

배열 초기화(Initialization)

- 크기가 명시되지 않은 배열: 초기값의 수가 암시적 배열크기

```
int a[] = {2, 3, 4, 5};
int a[4] = {2, 3, 4, 5};
```

- 문자 배열에도 적용

```
char s[] = "abc";
char s[] = {'a', 'b', 'c', '\0'};
```

→ 주의: s배열의 크기는 3이 아니라 4

첨자(Subscripting)

- a가 배열이면, a의 원소를 접근할 때, a[expr]과 같이 씀

```
a[expr]
```

  - → expr은 정수수식
  - → expr을 a의 첨자(subscript), 혹은 색인(index)이라 함

예제

```
int i, a[N];
...
x = a[i];
...
```

- 첨자 i의 범위
  - → 0 ≤ i ≤ N-1
- 선언한 범위는 프로그래머가 책임져야 함
- 범위를 벗어났을 때의 효과는 시스템마다 다름
  - → 대체로 실행시간 오류 발생

포인터(Pointers)

- 포인터
  - → 변수의 메모리 상의 저장주소를 다룰 수 있게 함
- 주소연산자(Address Operator)

```
&
```

  - → v가 변수라면, &v는 이 변수의 값이 저장된 메모리 위치, 또는 주소
- (예)

```
int v;

&v /* location, or address, in memory of v */
```

포인터 변수(Pointer Variables)

- 포인터 변수(Pointer Variables)
  - → 주소를 값으로 가짐

- 선언 방법

```
int *p;
```

  - → int형 변수의 주소를 가지는 포인터 변수
  - → 변수 이름 앞에 *를 붙임

- 포인터의 범위
  - → 특수 주소 0과 주소공간에 있는 양의 정수

포인터 변수(Pointer Variables)

- 포인터 변수 배정 예제
- p = 0;
- p = NULL; /* equivalent to p = 0 ; */
- p = &I ; /* i의 주소를 갖는 포인터 변수 */
- p = (int *) 1776; /* an absolute address in memory */

역참조 연산자(Dereferencing Operator)

*****

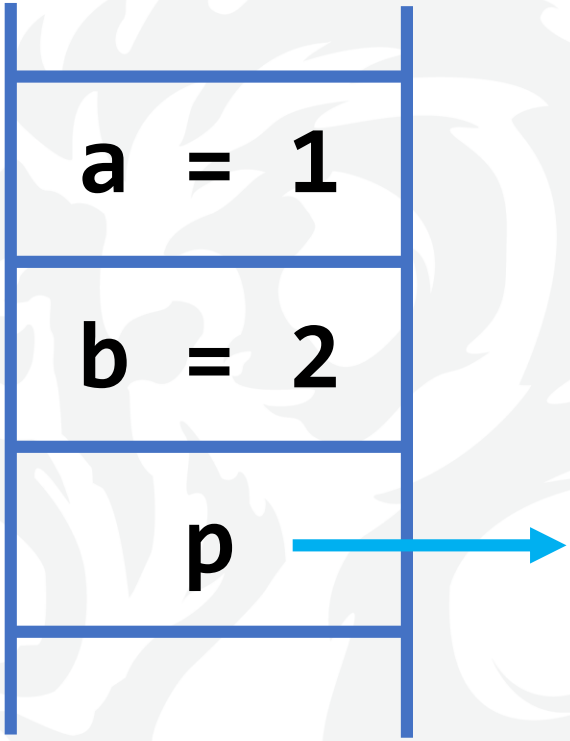
- 참조, 간접지정 혹은 역참조 연산자라고도 함
- 단항 연산자, R-->L 결합법칙
- &의 반대 연산자

***p**

- p가 포인터라면, *p는 p가 주소인 변수의 값을 나타냄

포인트 메커니즘

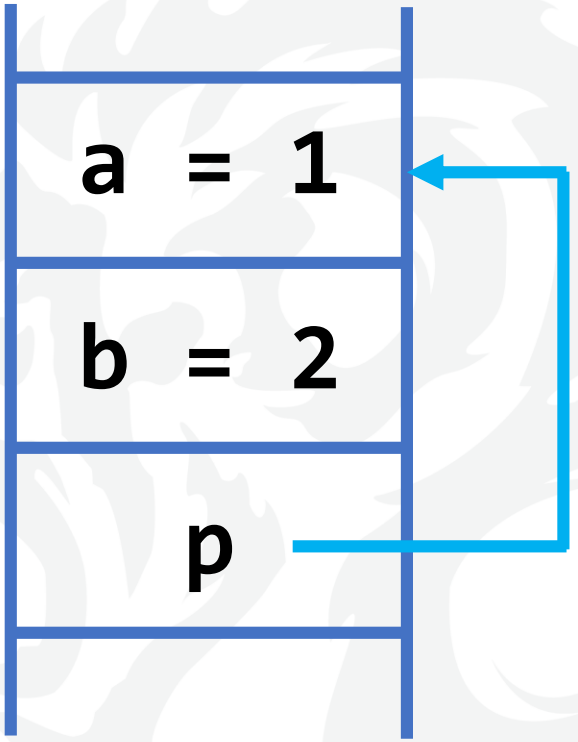
- `int a = 1, b = 2, *p;`



포인트 메커니즘

- `p = &a; /* p에 a의 주소 배정 */`
  - → `p`: a의 주소를 가리킴
  - → `*p`: a의 주소에 있는 변수값
- `b = *p;`

↔ `b = a;` ↔ `b = 1;`



(예) 포인터 주소

```
/* Printing an address, or location */
#include <stdio.h>

int main(void)
{
    int i = 7, *p = &i;
    printf("%s%d\n%s%p\n", "   Value of i: ", *p,
        "Location of i: ", p);
    return 0;
}
```

```
   Value of i: 7
Location of i: effffb24
```

```
Declarations and initializations
int     i = 3, j = 5, *p = &i, *q = &j, *r;
double  x;

Expression              Equivalent expression          Value
p == & i                p == (& i)                     1
* * & p                 * ( * (& p))                   3
r = & x                 r = (& x)                      /* illegal */
7 * * p / * q + 7       (((7 * ( * p))) / ( * q)) + 7  11
* (r = & j) *= * p      ( * (r = (& j))) *= ( * p)     15
```

```
Declarations
int       *p;
float     *q;
void      *v;

Legal assignments      Illegal assignments
p = 0;                 p = 1;
p = (int *) 1;         v = 1;
p = v = q;             p = q;
p = (int *) q;
```

지정할 수 없는 구조

- 상수
  - → &3 /* illegal */
- 일반식
  - → &(k + 99) /* illegal */
- register 변수
  - → register v;

    &v /* illegal */

Size of Pointers

```
#include <stdio.h>

int main(void)
{
    printf("size of char pointer : %d\n", sizeof(char *));
    printf("size of short pointer : %d\n", sizeof(short *));
    printf("size of int pointer : %d\n", sizeof(int *));
    printf("size of long pointer : %d\n", sizeof(long *));
    printf("size of signed pointer : %d\n", sizeof(signed *));
    printf("size of unsigned pointer : %d\n", sizeof(unsigned *));
    printf("size of float pointer : %d\n", sizeof(float *));
    printf("size of double pointer : %d\n", sizeof(double *));
    printf("size of longdouble pointer : %d\n", sizeof(long double *));
    return 0;
}
```

참조에 의한 호출

- C는 기본적으로 Call-by-Value(값에 의한 호출) 사용
- Call-by-Reference(참조에 의한 호출)의 효과를 얻기 위해서는 함수 정의의 매개변수 목록에서 포인터를 사용

(예) 참조에 의한 호출

```
#include <stdio.h>

void swap(int *p, int*q)
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}

int main(void)
{
    int     i = 3, j = 5;

    swap(&i, &j);
    printf("%d  &d\n", i, j);        /* 5 3 is printed */
    return 0;
}
```

"Call-by-Reference"의 효과를 얻는 방법

1. 함수 매개변수를 포인터형으로 선언
2. 함수 호출 시 주소를 인자로 전달
3. 함수 구현에서 역참조 포인터 사용

배열과 포인터의 관계

- 배열 이름 그 자체는 주소 또는 포인터 값이고, 배열과 포인터에는 둘 다 첨자를 사용할 수 있음
- 포인터 변수는 다른 주소들을 값으로 가질 수 있음
- 반면에 배열 이름은 고정된 주소 또는 포인터임

(예)

```
int      *p, *q;
int      a[4];
p = a;          /* is equivalent to   p = &a[0]; */
q = a + 3;      /* is equivalent to   q = &a[3]; */
```

배열과 포인터의 관계

- a와 p는 포인터이고 둘 다 첨자를 붙일 수도 있음

```
a[i] is equivalent to *(a + i)
p[i] is equivalent to *(p + i)
```

- 포인터 변수는 다른 값을 가질 수 있음

  하지만, 배열 이름은 값이 고정된(상수) 포인터이다.

```
p = a + i;
a = q; /* Illegal */
```

(예) 배열의 합

```
#define     N   100
int         *p, a[N], sum;

/* Version 1 */
for (i = 0, sum = 0; i < N; ++i)
    sum += a[i];    /* or sum += *(a + i); */

/* Version 2 */
for (p = a, sum = 0; p < &a[N]; ++p)
    sum += p;

/* Version 3 */
for (p = a, i = 0, sum = 0; i < N; ++i)
    sum += p[i];
```

포인터 연산과 원소 크기

- 포인터 연산은 C의 강력한 특징 중 하나
- 변수 p를 특정형에 대한 포인터라고 하면, 수식 p + 1은 그 형의 다음 변수를 나타냄
- p와 q가 모두 한 배열의 원소들을 포인트하고 있다면, p - q는 p와 q 사이에 있는 배열 원소의 개수를 나타내는 int 값을 생성함

포인터 수식과 산술 수식은 형태는 유사하지만, 완전히 다름

```
double  a[2], *p, *q;
p = a;                              /* points to base of array */
q = p + 1;                          /* equivalent to q = &a[1]; */
printf("%d\n", q - p);              /* 1 is printed */
printf("%d\n", (int) q - (int) p);  /* 8 is printed */
```

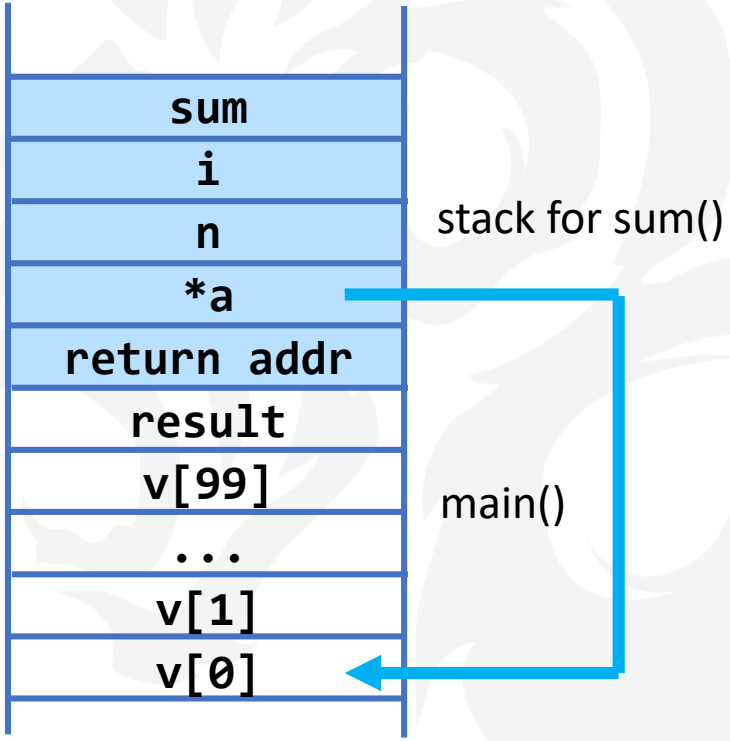
함수 인자로서의 배열

- 함수 정의에서 배열로 선언된 형식 매개변수는 실질적으로는 포인터임
- 함수의 인자로 배열이 전달되면, 배열의 기본 주소가 "값에 의한 호출"로 전달됨
- 배열 원소 자체는 복사되지 않음
- 표기 상의 편리성 때문에 포인터를 매개변수로 선언할 때 배열의 각 괄호 표기법을 사용할 수 있음

(예)

```
double sum(double a[], int n)   /* n is
the size of a[] */
{
    int     i;
    double  sum = 0.0;
    for (i = 0; i < n; ++i)
        sum += a[i];
    return sum;
}

int main(void)
{
    double v[100], result;
    ...
    result = sum(v, 100);
    ...
}
```



함수 헤드를 다음과 같이 정의해도 됨

```
double sum(double *a, int n)/* n is the size of a[] */
{
    ...
```

다양한 함수 호출 방법 및 의미

호출(Invocation)	계산 및 리턴되는 값
`sum(v, 100)`	`v[0] + v[1] + ... + v[99]`
`sum(v, 88)`	`v[0] + v[1] + ... + v[87]`
`sum(&v[7], k - 7)`	`v[7] + v[8] + ... + v[k - 1]`
`sum(v + 7, 2 * k)`	`v[7] + v[8] + ... + v[2 * k + 6]`

calloc()과 malloc()

- stdlib.h에 정의되어 있음
  - → calloc: contiguous allocation
  - → malloc: memory allocation
- calloc()과 malloc()을 이용해 배열, 구조체, 공용체를 위한 메모리 공간을 동적으로 생성할 수 있음

calloc()과 malloc()

- 각 원소의 크기가 el_size인 n개의 원소를 할당하는 방법
  - → calloc(n, el_size);
  - → malloc(n * el_size);
- calloc()은 모든 원소를 0으로 초기화하는 반면 malloc()은 하지 않음
- 할당 받은 메모리를 반환하기 위해서 free() 사용

(예)

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int     *a;                 /* to be used as an array */
    int     n;                  /* the size of the array */
    ...                         /* get n from somewhere, perhaps
                                interactively from the user */
    a = calloc(n, sizeof(int)); /* get space for a */
    ...
    free(a);
    ...
}
```

문자열

- char 형의 1차원 배열
- 문자열은 끝의 기호인 \0, 또는 널 문자로 끝남
- 널 문자: 모든 비트가 0인 바이트; 십진 값 0
- 문자열의 크기는 \0까지 포함한 크기

문자열

- 문자열 상수
  - 큰따옴표 안에 기술됨
  - 문자열 예 : "abc"
    - 마지막 원소가 널 문자이고 크기가 4인 문자 배열
- 주의 - "a"와 'a'는 다름
  - 배열 "a"는 두 원소를 가짐
  - 첫 번째 원소는 'a', 두 번째 원소는 '\0'

문자열

- 컴파일러는 문자열 상수를 배열 이름과 같이 포인터로 취급

```
char *p = "abc";
printf("%s %s\n", p, p + 1); /* abc bc is printed */
```

  - → 변수 p에는 문자 배열 "abc"의 기본 주소가 배정
  - → char 형의 포인터를 문자열 형식으로 출력하면, 그 포인터가 포인트하는 문자부터 시작하여 \0이 나올 때까지 문자들이 연속해서 출력됨

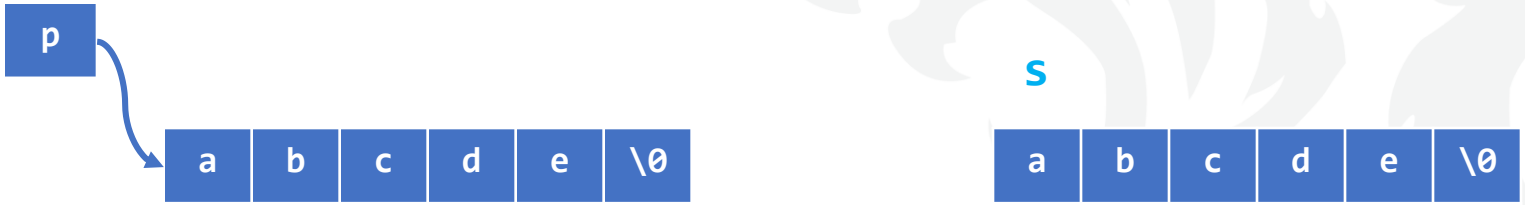
문자열

- "abc"와 같은 문자열 상수는 포인터로 취급되기 때문에 "abc"[1] 또는 *("abc" + 2)와 같은 수식을 사용할 수 있음

문자열

- 배열과 포인터의 차이

```
char *p = "abcde";
char s[ ] = "abcde";
// char s[ ] = {'a', 'b', 'c', 'd', 'e', '\0'};
```



(예)

```
#include <ctype.h>

int word_cnt(const char *s)
{
    int cnt = 0;
    while (*s != '\0') {
        while (isspace(*s))     // skip white space
            ++s;
        if (*s != '\0') {       // found a word
            ++cnt;
            while (!isspace(*s) && *s != '\0')
                                // skip the word
                ++s;
        }
    }
    return cnt;
}
```

문자열 조작 함수

```
char *strcat(char *s1, const char *s2);
```

→ 두 문자열 s1,s2를 결합하고, 결과는 s1에 저장

```
int strcmp(const char *s1, const char *s2);
```

→ s1과 s2를 사전적 순서로 비교하여, s1이 작으면 음수, 크면 양수, 같으면 0을 리턴

```
char *strcpy(char *s1, const char *s2);
```

→ s2의 문자를 \0이 나올 때까지 s1에 복사

```
size_t strlen(const char *s);
```

→ \0을 뺀 문자의 개수를 리턴

문자열 조작 함수 strlen()

```
size_t strlen(const char *s)
{
   size_t n;
   for (n = 0; *s != '\0'; ++s)
      ++n;
   return n;
}
```

문자열 조작 함수 strcpy()

```
char *strcpy(char *s1, register const char *s2)
{
   register char *p = s1;
   while (*p++ = *s2++)
      ;
   return s1;
}
```

6.11 String-Handling Functions in the Standard Library

```
Declarations and initializations
char    s1[] = "beautiful big sky country",
        s2[] = "how now brown cow";

Expression      Value
strlen(s1)      25
strlen(s2 + 8)  9
strcmp(s1, s2)  negative integer


Statements                  What gets printed
printf("%s", s1 + 10);      big sky country
strcpy(s1 + 10, s2 + 8);
strcat(s1, "s!");
printf("%s", s1);           beautiful brown cows!
```

다차원 배열

- C 언어는 배열의 배열을 포함한 어떠한 형의 배열도 허용함
- 2차원 배열은 두 개의 각괄호로 만듬
- 이 개념은 더 높은 차원의 배열을 만들 때에도 반복적으로 적용됨

배열 선언의 예	설명
`int a[100];`	1차원 배열
`int b[2][7];`	2차원 배열
`int c[5][3][2];`	3차원 배열

2차원 배열

- 2차원 배열은 행과 열을 갖는 직사각형의 원소의 집합으로 생각하는 것이 편리함
  - → 사실 원소들은 하나씩 연속적으로 저장됨
- 선언

```
int a[3][5];
```

	1열	2열	3열	4열	5열
1행	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
2행	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
3행	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

2차원 배열 (a[i][j]와 같은 표현들)

```
*(a[i] + j)

(*(a + i)) [j]

*((*(a + i)) + j)

*(&a[0][0] + 5 * i + j)
```

기억장소 사상 함수

- 배열에서 포인터 값과 배열 첨자 사이의 사상
- 예

```
int a[3][5];
```

→ 배열 a의 a[i][j]에 대한 기억장소 사상 함수:

```
*(&a[0][0] + 5 * i + j)
```

형식 매개변수 선언

- 함수 정의에서 형식 매개변수가 다차원 배열일 때, 첫 번째 크기를 제외한 다른 모든 크기를 명시해야 함
  - → 기억장소 사상 함수를 위해

(예) int a[3][5];으로 선언되어 있을 때

```
int sum(int a[][5]){      /* int sum(int a[3][5])
                  or int sum(int (*a)[5]) */
   int i, j, sum=0;
   for (i = 0; i < 3; ++i)
      for (j = 0; j < 5; ++j)
         sum += a[i][j];
   return sum;
}
```

3차원 배열

- 3차원 배열 선언 예

```
int a[7][9][2];
```

→ a[i][j][k]를 위한 기억장소 사상 함수:

```
*(&a[0][0][0] + 9 * 2 * i + 2 * j + k)
```

- 함수 정의 헤더에서 다음은 다 같음

```
int sum(int a[][9][12])
int sum(int a[7][9][12])
int sum(int (*a)[9][12])
```

초기화

- 다차원 배열 초기화 방법

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
int a[ ][3] = {{1, 2, 3}, {4, 5, 6}};
```

  - 내부 중괄호가 없으면, 배열은 a[0][0], a[0][1], ..., a[1][2] 순으로 초기화되고, 인덱싱은 행 우선 임
  - 배열의 원소 수보다 더 적은 수의 초기화 값이 있다면, 남는 원소는 0으로 초기화됨
  - 첫 번째 각괄호가 공백이면, 컴파일러는 내부 중괄호 쌍의 수를 그것의 크기로 함
  - 첫 번째 크기를 제외한 모든 크기는 명시해야 함

초기화

- 초기화 예

```
int a[2][2][3]={
        {{1, 1, 0}, {2, 0, 0}},
        {{3, 0, 0}, {4, 4, 0}}
    };
```

- 이것은 다음과 같음

```
int a[][2][3]={{{1, 1}, {2}}, {{3}, {4, 4}}};
```

- 모든 배열 원소를 0으로 초기화 하기

```
int a[2][2][3] = {0};
    /* all element initialized to zero */
```

typedef 사용 예

```
#define N 3
typedef double scalar;
typedef scalar vector[N];
typedef scalar matrix[N][N];
/* typedef vector matrix[N]; */
```

→ 의미있는 이름을 사용하는 형 이름을 정의하여 가독성을 높임

포인터 배열

- 배열의 원소의 형은 포인터형을 포함하여 임의의 형이 될 수 있음
- 포인터 배열은 문자열을 다룰 때 많이 사용됨

main() 함수의 인자

- main()은 운영체제와의 통신을 위해 argc와 argv라는 인자를 사용함

예제 코드

```
void main(int argc, char *argv[]) {
int i;
printf("argc = %d\n", argc);
for (i = 0; i < argc; ++i)
printf("argv[%d] = %s\n", i, argv[i]);
}
```

→ argc : 명령어 라인 인자의 개수를 가짐
→ argv : 명령어 라인을 구성하는 문자열들을 가짐

main() 함수의 인자

- 앞의 프로그램을 컴파일하여 my_echo로 한 후, 다음 명령으로 실행:

```
$ my_echo a is for apple
```

- 출력:

```
argc = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```

래기드 배열

```
#include <stdio.h>

int main(void)
{
    char a[2][15] = {"abc:", "a is for apple"};
    char *p[2] = {"abc:", "a is for apple"};
    printf("%c%c%c %s %s\n",
        a[0][0], a[0][1], a[0][2], a[0], a[1]);
    printf("%c%c%c %s %s\n",
        p[0][0], p[0][1], p[0][2], p[0], p[1]);
    return 0;
}
```

출력

- abc abc: a is for apple
- abc abc: a is for apple

래기드 배열: 식별자 a

- 2차원 배열
- 30개의 char 형을 위한 공간이 할당
- 즉, a[0]과 a[1]은 15개 char의 배열
- 배열 a[0]은 다음으로 초기화됨: {'a', 'b', 'c', ':', '₩0'}
- 5개의 원소만 명시되어 있기 때문에, 나머지는 0(널 문자)으로 초기화됨
- 이 프로그램에서 배열의 모든 원소가 사용되지는 않지만, 모든 원소를 위한 공간이 할당됨
- 컴파일러는 a[i][j]의 접근을 위해 기억장소 사상 함수를 사용
- 즉, 각 원소를 접근하기 위해서는 한 번의 곱셈과 한 번의 덧셈이 필요함

래기드 배열: 식별자 p

- char 포인터의 1차원 배열
- 이 선언으로 두 포인터를 위한 공간이 할당
- p[0] 원소는 "abc :"를 포인트하도록 초기화되고, 이 문자열은 5개의 char를 위한 공간을 필요로 함
- p[1] 원소는 "a is ..."를 포인트하도록 초기화되고, 이 문자열은 15개의 char를 위한 공간을 필요로 함
- 즉, p는 a보다 더 적은 공간을 사용
- p[i][j] 접근을 위해 기억장소 사상 함수 사용하지 않음
- (p를 사용하는 것이 a를 사용하는 것보다 빠름)
- a[0][14]는 유효한 수식이지만, p[0][14]는 그렇지 않음
- p[0]과 p[1]은 상수 문자열을 포인트함 - 변경할 수 없음

래기드 배열

- 래기드 배열 : 배열의 원소인 포인터가 다양한 크기의 배열을 포인트하는 것
- 앞의 프로그램에서 p의 행들은 다른 길이를 갖기 때문에, p를 래기드 배열이라고 할 수 있음

인자로서의 함수

- 함수의 포인터는 인자로서 전달될 수 있고, 배열에서도 사용되며, 함수로부터 리턴될 수도 있음

인자로서의 함수

```
double sum_square(double f(double x), int m, int n)
{
   int k;
   double sum = 0.0;
   for (k = m; k <= n; ++k)
      sum += f(k) * f(k);
   return sum;
}
```

인자로서의 함수

- 앞의 코드에서 식별자 x는 사람을 위한 것으로, 컴파일러는 무시함 즉, 다음과 같이 해도 됨

```
double sum_square(double f(double), int m, int n)
{
    ...
```

인자로서의 함수

- 포인터 f를 함수처럼 취급할 수도 있고, 또는 포인터 f를 명시적으로 역참조할 수도 있음
  - → 즉, 다음 두 문장은 같음

```
sum += f(k) * f(k);
sum += (*f)(k) * (*f)(k);
```

인자로서의 함수

(*f)(k)

- f 함수에 대한 포인터
- *f 함수 그 자체
- (*f)(k) 함수 호출

const

- const는 선언에서 기억영역 클래스 뒤와 형 앞에 옴
- 사용 예

```
static const int k = 3;
```

  → 이것은 "k는 기억영역 클래스 static인 상수 int이다"라고 읽음

- const 변수는 초기화될 수는 있지만, 그 후에 배정되거나, 증가, 감소, 또는 수정될 수 없음
- 변수가 const로 한정된다 해도, 다른 선언에서 배열의 크기를 명시하는 데는 사용될 수 없음

(예) const#1

```
const int n = 3;
int v[n];
/* any C compiler should complain */
```

(예) const#2

```
const int a = 7;
int *p = &a; /* the compiler will complain */
```

- p는 int를 포인트하는 보통의 포인터이기 때문에, 나중에 ++*p와 같은 수식을 사용하여 a에 저장되어 있는 값을 변경할 수 있음

(예) const#3

```
const int a = 7;
const int *p = &a;
```

- 여기서 p 자체는 상수가 아님
- p에 다른 주소를 배정할 수 있지만, *p에 값을 배정할 수는 없음

(예) const#4

```
int a;
int * const p = &a;
```

- p는 int에 대한 상수 포인터임
- 따라서, p에 값을 배정할 수는 없지만, *p에는 가능함

(예) const#5

```
const int a = 7;
const int *const p = &a;
```

- p는 상수 int를 포인트하는 상수 포인터임
- 이제 p와 *p 모두는 배정될 수 없고, 증가나 감소도 안됨

volatile

- volatile 객체는 하드웨어에 의하여 어떤 방법으로 수정될 수 있음

```
extern const volatile int real_time_clock;
```

  - 한정자 volatile은 하드웨어에 의해 영향을 받는 객체임을 나타냄
  - 또한 const도 한정자이므로, 이 객체는 프로그램에서 증가, 감소, 또는 배정될 수 없음
  - 즉, 하드웨어는 변경할 수 있지만, 코드로는 변경할 수 없음

Homework

- Exercises #1, 4, 5, 12, 20, 23, 31, 35