Arrays

adopted from KNK C Programming : A Modern Approach

Scalar 변수 versus Aggregate 변수

- ***scalar :*** 한 번에 한 값만 저장 가능한 변수만 지금까지 다룸

- ***aggregate*** 집합 변수도 지원, 하나 이상의 다른 값 저장 가능

- 집합 변수는 두 종류가 있음, 배열(array), 구조체(structure)
  - 배열을 먼저 다룸

One-Dimensional Arrays 1차원 배열

- ***array*** 는 같은 타입의 여러 개의 값을 갖는 자료 구조임
- 이 값들은 요소 ***elements***라고 함.
  - 배열 내의 위치를 나타내는 값으로 요소를 선택함

- 1차원 배열이 가장 간단함

- 일차원 배열 a 는 개념적으로 일렬로 줄을 선 것처럼 표현됨

One-Dimensional Arrays

- 배열 선언 시 배열 형과 길이를 명시해야 함

```
int a[10];
```

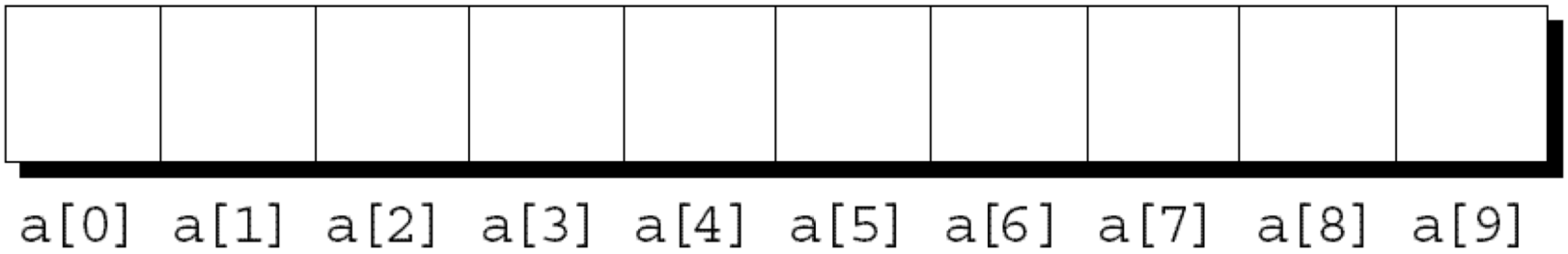
- 요소는 어떤 타입이어도 상관없음. 길이는 정수 배의 고정된 길이를 갖음

- 배열의 크기는 매크로로 정의하는 것이 바람직함

```
#define N 10
…
int a[N];
```

Array Subscripting 배열 첨자

- 배열의 요소를 참조/접근하기 위해 배열의 이름과 대괄호 안에 수를 넣음
- 대괄호 안에 수를 넣어 참조할 요소의 위치를 표시하는 것을 배열 첨자 ***subscripting*** 또는 인덱스***indexing***라고 함
  또 다른 말로, 참조할 배열의 위치를 인덱스라고 부름
- 배열의 크기가 n일 때 인덱스 시작과 끝 값은 0과 n-1임
- 배열 `a` 의 길이가 10일 때 각 요소의 인덱스는 `a[0]`, `a[1]`, …, `a[9]`:



Array Subscripting

- `a[i]` 은 lvalues이기 때문에 다른 변수들처럼 값을 할당 받을 수 있음

```
a[0] = 1;   // 인덱스가 실제 저장된 값이 아님!!
printf("%d\n", a[5]);
++a[i];
```

Array Subscripting

- 많은 프로그램들이 `for` 루프를 사용하여 배열의 모든 요소들을 처리함

- 길이 `N` 의 배열 `a` 을 다루는 예:

```
for (i = 0; i < N; i++)
  a[i] = 0;                 /* a 초기화*/

for (i = 0; i < N; i++)
  scanf("%d", &a[i]);       /* a 에 값 저장*/

for (i = 0; i < N; i++)
  sum += a[i];              /* a의 요소들의 합 계산 */
```

Array Subscripting

- C 는 인덱스 경계를 검사하지 않음; 범위 초과 시 오동작을 함

- 흔한 실수: 인덱스는 1부터가 아닌 0부터 $n - 1$

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

어떤 컴파일러의 경우 무한 루프에 빠지게 됨.

Array Subscripting

- 인덱스는 정수 표현식으로 사용 가능

```
a[i+j*10] = 0;
```

- 인덱스를 표현한 수식에 사이드 이펙트가 있어도 됨

```
i = 0;
while (i < N)
  a[i++] = 0;
```

Array Subscripting

- 인덱스 값에 사이드 이펙트가 있는 경우 주의해야 함

```
i = 0;
while (i < N)
  a[i] = b[i++];
```

- `a[i] = b[i++]` 는 `i` 의 값을 활용하기도 하고 변경하기도 하기 때문에 정의되지 않은 동작을 함

- 인덱스에서 증감 연산자를 분리하면 문제 해결 가능

```
for (i = 0; i < N; i++)
  a[i] = b[i];
```

배열 초기화

- 변수 선언시 값을 할당하여 초기화할 수 있음

- 흔한 초기화 방법: 중괄호와 값 그리고 쉼표로 구분

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Array Initialization

- 최기화하는 값들이 배열의 길이보다 짧은 경우 나머지 값들은 0으로 초기화 됨:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

- 이 기능을 써서 0으로 쉽게 초기화 할 수 있음:

```
int a[10] = {0};
/* initial value of a is {0, 0, 0, 0, 0, 0, 0, 0, 0, 0} */
```

  괄호 안에 0이 하나 있는데, 아무 값도 없이 초기화하는 것은 규칙에 위배됨

- 배열의 길이보다 많은 값들로 초기화하는 것도 규칙 위반

Array Initialization

- 초기화 값들이 있다면 길이는 생략 가능:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- 초기화에 제공된 값들로 컴파일러가 배열의 길이를 판단함

위치 지정 초기화 (C99 표준)

- 배열의 모든 요소를 초기화하는 경우보다 한 두 요소를 초기화하는 경우가 많이 있음
- 예:

```
int a[15] =
  {0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

- 긴 배열의 경우 이런 초기화 방식은 오류 발생 여지가 높음

Designated Initializers (C99)

- C99의 ***designated initializers 위치 지정 초기화*** 로 문제 해결.

- 앞의 예를 위치 지정 초기화로 표현한 예:

```
int a[15] = {[2] = 29, [9] = 7, [14] = 48};
```

- 괄호 안의 숫자가 ***designator 지정자 역할을 함.***

Designated Initializers (C99)

- 위치 지정 초기화는 짧고 읽기가 쉬움
- 위치 지정의 순서 상관 없음

- 앞의 예를 적는 또 다른 예:

```
int a[15] = {[14] = 48, [9] = 7, [2] = 29};
```

Designated Initializers (C99)

- 지정자는 정수형 표현식을 갖어야 함
- 초기화 하는 배열의 길이가 n이면 지정자의 값도 0 과 $n - 1$ 사이에 있어야 함.
- 배열의 길이가 명시되지 않으면 지정자는 양수이면 됨.
  - 컴파일러가 가장 큰 지정자를 기준으로 길이를 판단함

- 다음의 배열은 길이가24 임:

```
int b[] = {[5] = 10, [23] = 13, [11] = 36, [15] = 29};
```

Designated Initializers (C99)

- C99 표준 이전의 방식을 겸해서 쓸 수 있음:

```
int c[10] = {5, 1, 9, [4] = 3, 7, 2, [8] = 6};
```

배열에 `sizeof` 연산자 사용하기

- `sizeof` 연산자는 배열의 길이를 바이트 단위로 계산함
- 배열 `a` 의 길이가 정수 10 개라면 `sizeof(a)` 는 40을 알려줌 (단, 정수가 4바이트인 경우).
- `sizeof` 를 배열의 요소 하나 `a[0]`의 크기를 알 때도 씀
- 이 두 값을 나누면 배열의 길이를 알 수 있음:

```
sizeof(a) / sizeof(a[0])
```

Using the **`sizeof`** Operator with Arrays

- 많은 사람들이 이 표현식을 배열의 길이를 얻기 위해 씀.

- 배열 a의 값을 초기화하는 예:

```
for (i = 0; i < sizeof(a) / sizeof(a[0]); i++)
  a[i] = 0;
```

이렇게 쓰면 나중에 배열의 길이가 바뀌더라도 수정 불필요

Using the **`sizeof`** Operator with Arrays

- 어떤 컴파일러는 다음 문장에 경고 메시지를 출력하기도 함
  `i < sizeof(a) / sizeof(a[0])`.
- 변수 `i` 는 `int` (a signed type)형인데, `sizeof` 는 정수형이기는 하지만 부호가 없는 `size_t` (an unsigned type) 형이기 때문.
- 부호 있는 정수형과 없는 정수형을 비교하는 것은 문제가 있지만, 이 경우는 안전함

Using the **`sizeof`** Operator with Arrays

- 경고 메시지를 피하고 싶으면 cast를 해서 부호 있도록 해야 함
  기존의 `sizeof(a) / sizeof(a[0])`을

```
for (i = 0; i < (int) (sizeof(a) / sizeof(a[0])); i++)
  a[i] = 0;
```

- 매크로를 쓰면 좀 더 유용함:

```
#define SIZE ((int) (sizeof(a) / sizeof(a[0])))

for (i = 0; i < SIZE; i++)
  a[i] = 0;
```

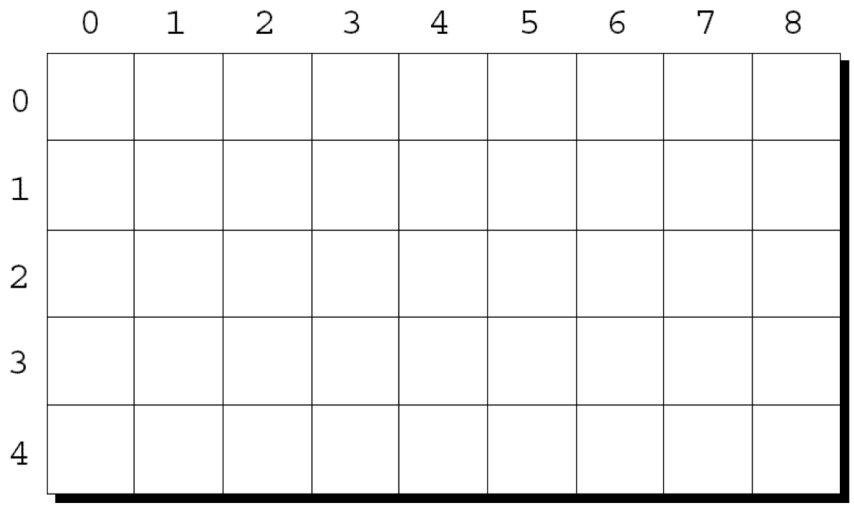
Multidimensional Arrays 다차원 배열

- 배열에 쓸 수 있는 차원에는 제한 없음

- 다음의 문장은 2차원 배열을 선언함 (수학 용어로는 *matrix*):

```
int m[5][9];
```

- `m` 은 5줄 9칸으로 이루어 졌고 둘다 인덱스는 0에서 시작함



Multidimensional Arrays

- 배열 m 의 i번째 줄, j번째 칸 요소는 m[i][j]으로 표현.
- m[i] 은 m의 i번째 줄을 뜻하고, m[i][j] 에서 j가 그 줄의 j번째 칸의 요소를 뜻함

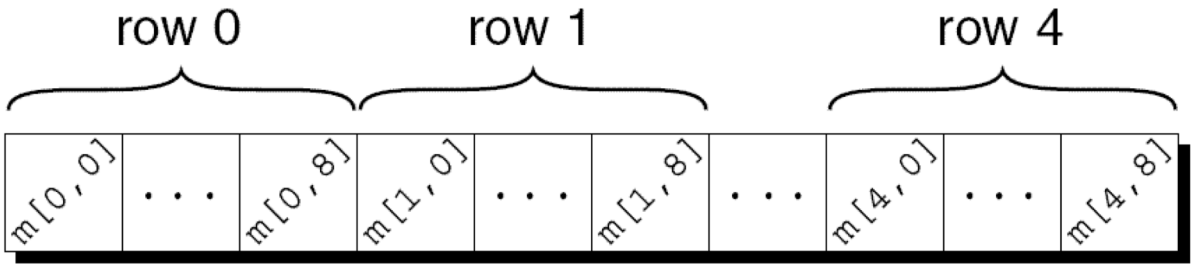
- m[i,j]와 같이 쓰면 안되고 꼭 m[i][j]으로 써야 함.
- C 에서 m[i,j]의 쉼표는 연산자로 쓰여서 m[j]가 된다.

Multidimensional Arrays

- 그림으로 2차원 배열을 표현하기는 하지만, 컴퓨터 메모리에서는 그림처럼 배치가 되지 않음

- C 에서는 2차원 이상의 배열을 ***row-major order (줄을 기준으로 정렬함),*** 0번 줄의 요소들을 순서대로 배치하고 1 번줄의 요소들을 배치하는 식임.

- 배열 m이 배치된 모습:



Multidimensional Arrays

- 중첩 `for` 루프는 다차원 배열을 처리하는 데 이상적임

- 단위 행렬을 초기화하는 문제를 생각해보자. 두 개의 중첩 `for` 루프로 충분히 해결할 수 있음:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Multidimensional Array 의 초기화

- 2차원 배열의 초기화를 위해 1차원 배열 초기화를 중첩 시키면 됨:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- 더 높은 차원의 배열도 같은 방식으로 초기화 할 수 있음

- C 는 다차원 배열 초기화를 위해 몇 가지 방법을 제공함

Initializing a Multidimensional Array

- 제공된 초기화 값이 다차원 배열을 모두 채우지 못한다면 그 나머지는 0을로 채움

- 다음과 같은 초기화 문장은 배열 m의 첫 3줄만 초기화하고 나머지 두 줄은 0으로 초기화 함

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

Initializing a Multidimensional Array

- 칸에 해당하는 요소들이 배열의 칸수 보다 적은 경우, 명시된 값들 외에는 0으로 초기화 됨:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1},
               {0, 1, 0, 1, 1, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 0, 1},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Initializing a Multidimensional Array

- 내부에 중괄호를 생략하고 초기화 할 수 있음:

```
int m[5][9] = {1, 1, 1, 1, 1, 0, 1, 1, 1,
               0, 1, 0, 1, 0, 1, 0, 1, 0,
               0, 1, 0, 1, 1, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 0, 1, 0,
               1, 1, 0, 1, 0, 0, 1, 1, 1};
```

  이런 경우 한 줄을 먼저 다 채우고 그 다음 줄을 채우기 시작함

- 내부에 중괄화를 생략하면 어디까지 채웠는지 알 수가 없어서 더 많은 수를 썼거나 적게 쓰는 실수를 범할 수 있음

Initializing a Multidimensional Array

- C99의 지정 초기화도 다차원 배열에서 쓸 수가 있음

- 2 × 2 단위 행렬을 초기화 하는 예:

```
double ident[2][2] = {[0][0] = 1.0, [1][1] = 1.0};
```

  명시되지 않은 나머지는 모두 0으로 채움

Constant Arrays 상수 배열

- 배열의 값을 변경하지 못하게 상수 취급을 할 수 있음. 선언을 하기 위해 앞에 `const` 를 붙이면 됨:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- 배열이 `const` 로 선언되면 프로그램에서 변경이 불가능함

Constant Arrays

- 배열을 `const` 로 선언하는 장점:
  - 배열 값을 프로그램이 변경할 수 없음을 문서화함
  - 배열 값을 변경하는 코드가 있다면 컴파일러가 오류를 알림

- `const` 가 배열에만 국한된 키워드가 아니지만, 특히 배열에 유용함

Variable-Length Arrays (C99) 가변 길이 배열

- C89 표준에서는 배열의 길이는 상수 표현식으로 선언되어야 했음

- C99 표준에서는 상수가 아닌 식으로 선언 가능하도록 허용됨
- 예:

```
int i, n;

scanf("%d", &n);

int a[n];
```

Variable-Length Arrays (C99)

- 앞의 예에서 `a` 가 ***variable-length array*** (or ***VLA***) 가변길이배열
- VLA의 길이는 프로그램이 실행된 후에 결정이 됨
- VLA 의 가장 큰 장점은 필요한 만큼 길이를 정할 수 있다는 것
- 쓰지 않는 메모리를 길게 할당하거나, 너무 적게 할당하는 실수를 줄일 수 있음

Variable-Length Arrays (C99)

- 한 변수에 의해서 VLA 의 길이가 정해지는 것이 아님. 다음과 같은 식도 가능:

```
int a[3*i+5];
int b[j+k];
```

- 다차원 배열도 VLA로 선언할 수 있음:

```
int c[m][n];
```

- 제한 요소들:
  - 정적 저장기간(static storage duration을 쓸 수 없음 (교재 18장)
  - 초기화가 불가능