Pointers and Arrays

adopted from KNK C Programming : A Modern Approach

Introduction

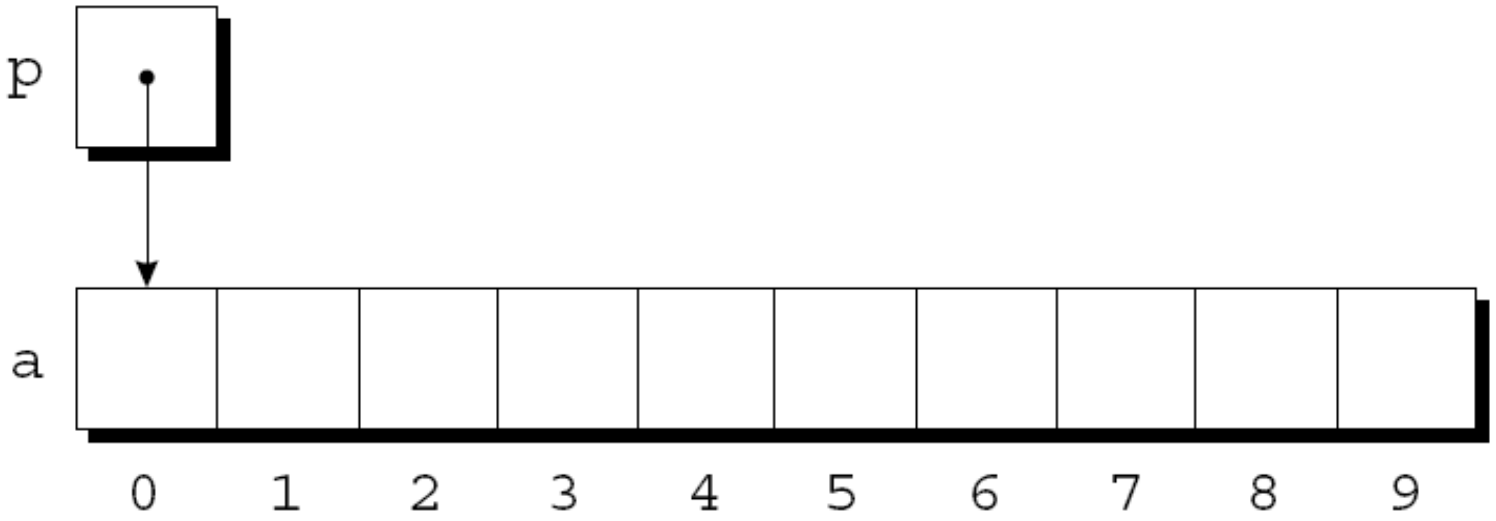
- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
  C는 배열 요소에 대한 포인터 덧셈 뺄셈을 지원함
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
  배열 첨자를 사용하지 않고 포인터로 배열을 조작할 수 있음
- The relationship between pointers and arrays in C is a close one.
  C에서 포인터와 배열은밀접한 관계가 있음
- Understanding this relationship is critical for mastering C.
  이 관계를 이해하는 것이 C를 마스터하는데 중요함

Pointer Arithmetic 포인터 연산

- Chapter 11 showed that pointers can point to array elements:
  11장에서 포인터는 배열의 요소를 가리킬 수 있음을 보였음

```
int a[10], *p;
p = &a[0];
```

- A graphical representation: 그림으로 표현해보자



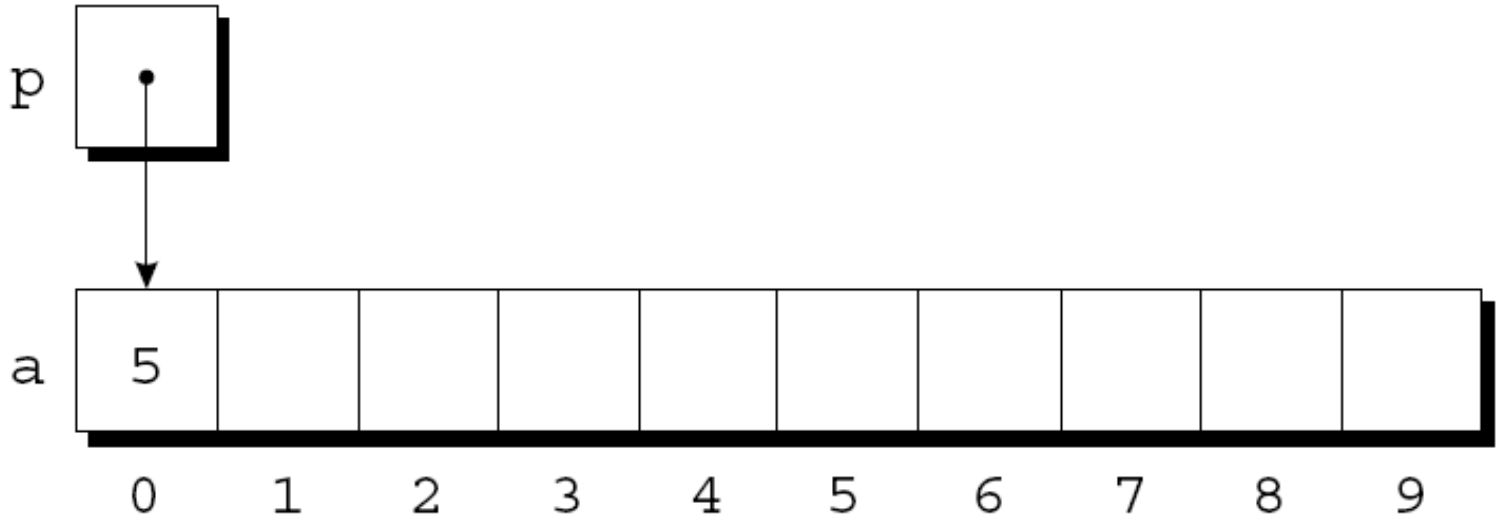
Pointer Arithmetic

- We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```

a[0]을 포인터로 접근할 수 있음; 그리고 상기 문장을 통해 a[0]을 접근할 수 있음

- An updated picture: 위 문장을 반영한 그림



Pointer Arithmetic

- If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
  p가 배열 a의 요소를 가리킬 때 포인터 연산을 통해 a의 다른 요소도 접근 가능
- C supports three (and only three) forms of pointer arithmetic:
  c에서 포인터는 3가지만 있음
  - Adding an integer to a pointer 포인터에 정수 더하기
  - Subtracting an integer from a pointer 포인터에 정수 빼기
  - Subtracting one pointer from another 포인터 빼기 포인터

Adding an Integer to a Pointer 포인터에 정수 더하기

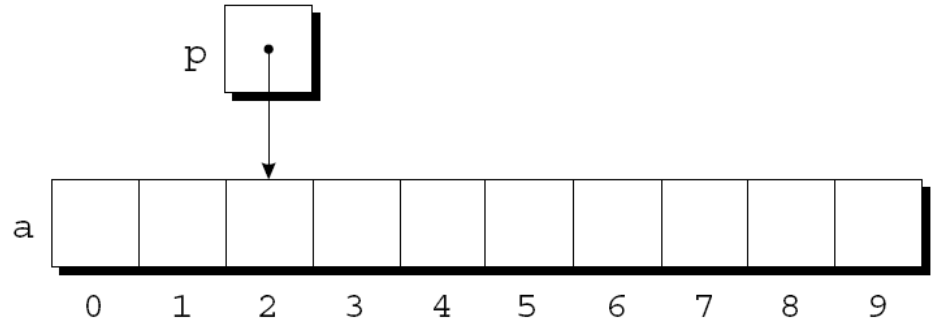
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
  포인터 p에 정수 j 를 더하면 p 에서 j번 째 다음 요소를 가리킴
- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
  P가 a[i]요소를 가리킬 때, p+j는 a[i+j]를 가리킴
- Assume that the following declarations are in effect:
  다음과 같이 선언했다고 하자.

```
int a[10], *p, *q, i;
```

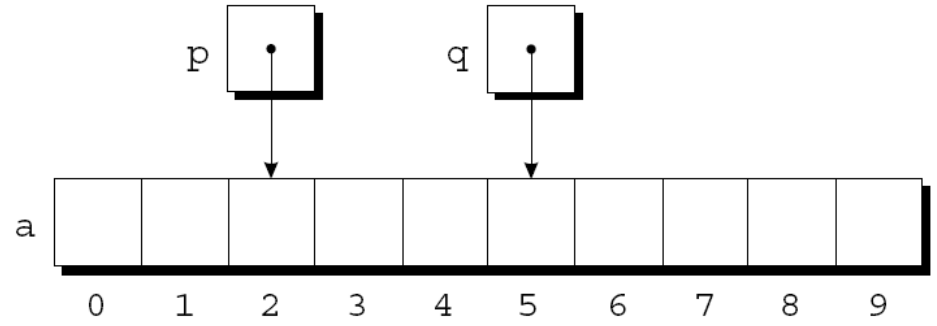
Adding an Integer to a Pointer

- Example of pointer addition: 포인터 덧셈의 예

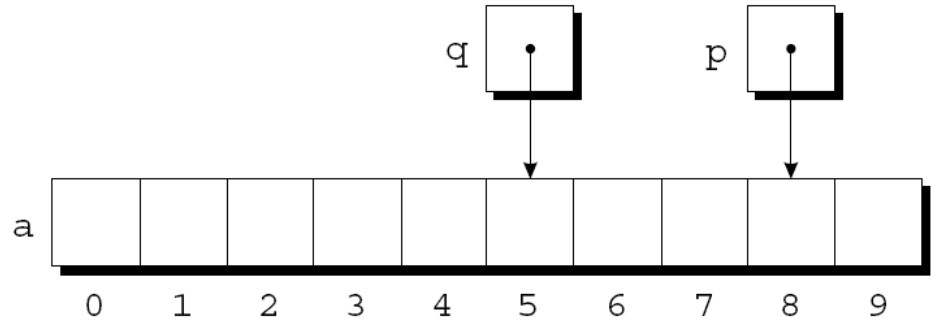
```
p = &a[2];
```




```
q = p + 3;
```



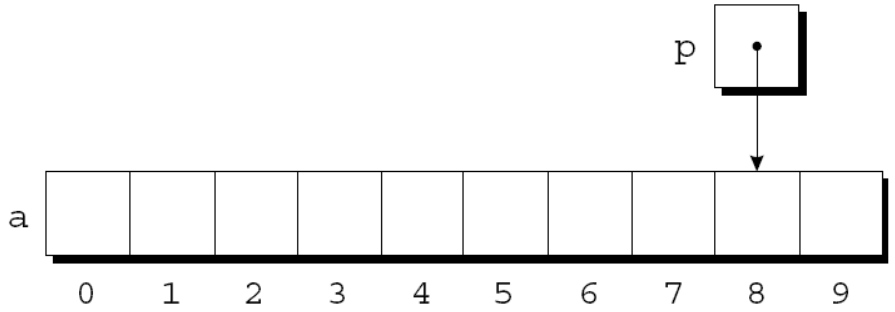

```
p += 6;
```



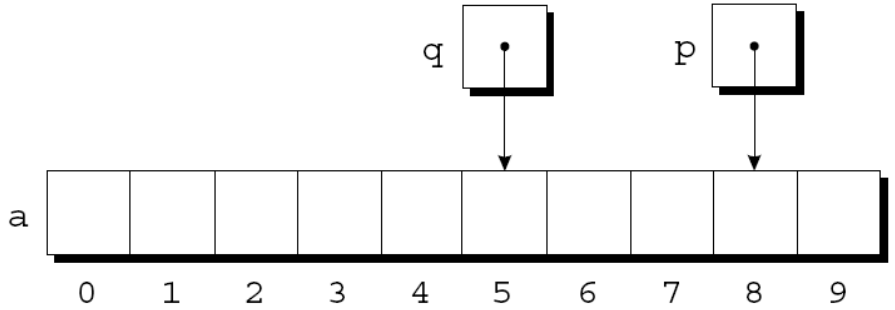
Subtracting an Integer from a Pointer 포인터 뺄셈

- If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.
  P가 a[i]를 가리킬 때 p-j는 a[i-j]를 가리킴
- Example:

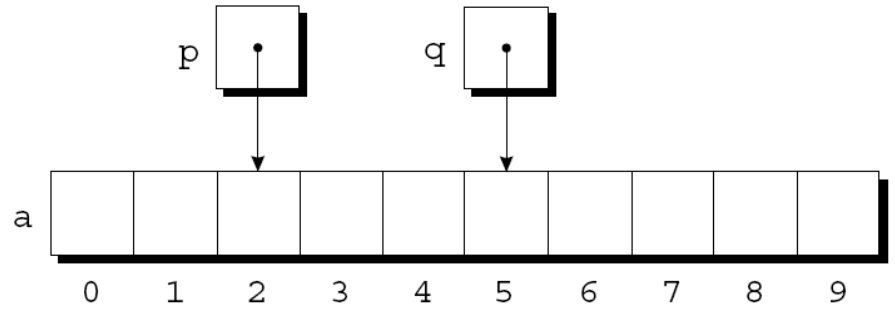
```
p = &a[8];
```




```
q = p - 3;
```



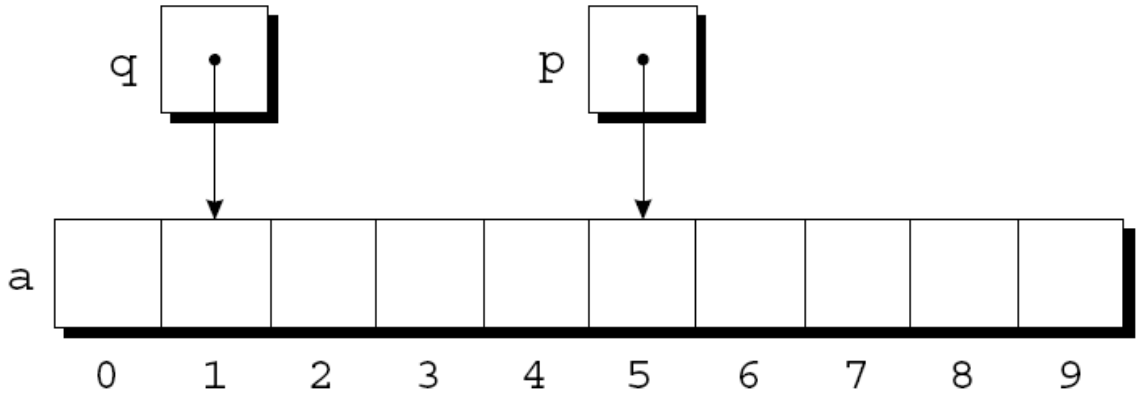

```
p -= 6;
```



Subtracting One Pointer from Another포인터-포인터

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
  포인터에서 포인터를 빼면 배열 요소를 단위로 한 포인터 간의 차를 구함
- If `p` points to `a[i]` and `q` points to `a[j]`, then `p – q` is equal to `i – j`. p가 a[i]를, q가 a[j]를 가리킬 때 p-q는 i-j와 같음
- Example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Subtracting One Pointer from Another

- Operations that cause undefined behavior:
  정의되지 않은 동작을 일으키는 연산
  - Performing arithmetic on a pointer that doesn't point to an array element 배열의 요소를 가리키지 않는 포인터에 연산을 하는 경우
  - Subtracting pointers unless both point to elements of the same array 같은 배열을 가리키지 않는 포인터들을 서로 빼는 경우

Comparing Pointers 포인터 비교

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
  포인터는 관계 연산자나 등호 연산자로 비교 가능
  - Using relational operators is meaningful only for pointers to elements of the same array.
    관계 연산자는 같은 배열을 가리키는 포인터들에 한에 의미 있음
- The outcome of the comparison depends on the relative positions of the two elements in the array.
  비교에 결과는 배열의 두 요소의 상대적 위치에 의존함
- After the assignments 다음과 같은 할당문의 결과는

```
p = &a[5];    // p<=q 는 0
q = &a[1];    // p>=q는 1
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

Pointers to Compound Literals (C99)

- It's legal for a pointer to point to an element within an array created by a compound literal:
  포인터 변수로 배열을 선언할 수 있음

```
int *p = (int []){3, 0, 3, 4, 1};
```

- Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:
  이와 같은 방식으로 배열을 선언하면 배열을 선언하고 배열의 첫 요소를 포인터로 가리키도록 하는 수고를 덜 수 있음

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

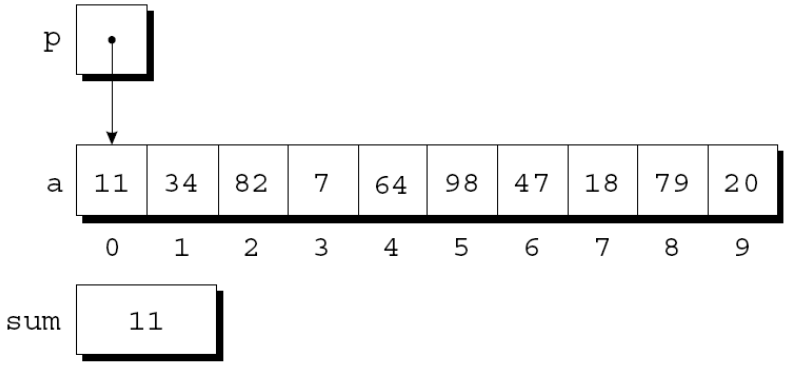
Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
  포인터 연산으로 포인터 변수를 증가시켜서 배열의 요소들을 순회할 수 있음
- A loop that sums the elements of an array `a`:
  배열 a의 요소들을 더하는 루프

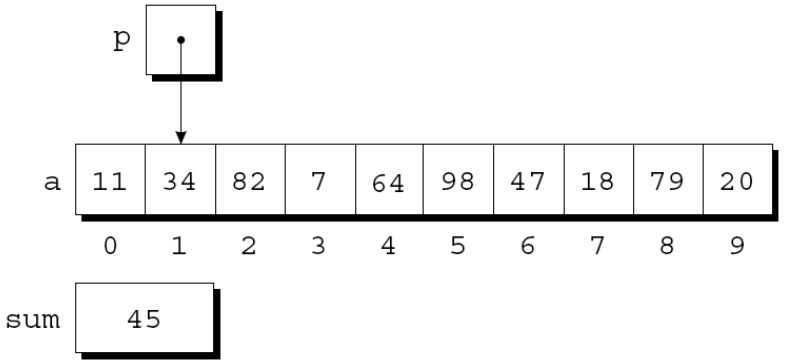
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

Using Pointers for Array Processing

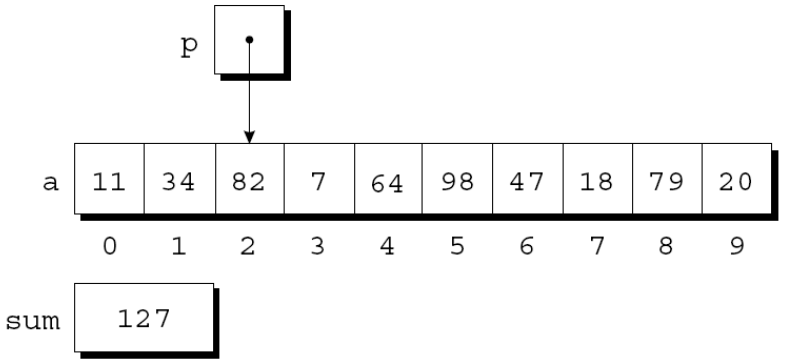
At the end of the first iteration:
첫 번째 반복의 끝



At the end of the second iteration:
두 번째 반복의 끝



At the end of the third iteration:
세 번째 반복의 끝



Using Pointers for Array Processing

- The condition `p < &a[N]` in the `for` statement deserves special mention. for문의 p <&[N] 비교 조건에 주의해야 함
- It's legal to apply the address operator to `a[N]`, even though this element doesn't exist.
  존재하지 않는 a[N]에 주소 연산자를 붙였지만 비교문장으로는 사용 가능
- Pointer arithmetic may save execution time.
  포인터 연산은 실행시간을 줄일 수 있음
- However, some C compilers produce better code for loops that rely on subscripting.
  어떤 c 컴파일러는 반복문에 첨자를 사용한 코드를 만들어 내기도 함

Combining the * and ++ Operators

- C programmers often combine the * (indirection) and ++ operators. c 개발자들은 *(역참조)와 ++ 연산자를 결합하기도 함
- A statement that modifies an array element and then advances to the next element: 다음 문장은 배열의 요소를 변경하고 다음 요소를 방문

```
a[i++] = j;
```

- The corresponding pointer version: 포인터를 사용한 방법

```
*p++ = j;
```

- Because the postfix version of ++ takes precedence over *, the compiler sees this as
  연산자 우선 순위 상 ++가 *보다 우선 순위가 높기 때문에 다음과 같이 해석됨

```
*(p++) = j;
```

Combining the * and ++ Operators

- Possible combinations of * and ++: 결합 방법

Expression	*Meaning*
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later *p를 사용하고 p를 증가
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later *p를 사용하고 *p를 증가
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment p를 증가하고; 증가된 *p를 사용
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment *p를 증가하고; 증가된 *p를 사용

Combining the * and ++ Operators

- The most common combination of `*` and `++` is `*p++`, which is handy in loops. 가장 흔한 * 와 ++의 결합 방식은 *p++으로 루프에 유용함
- Instead of writing 배열 a의 요소를 합산을 위해 아래와 같이 작성하는 대신

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array `a`, we could write
다음과 같이 작성 할 수 있음

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

Combining the * and ++ Operators

- The `*` and `--` operators mix in the same way as `*` and `++`.
  *과 --연산자도 *와 ++의 결합과 같은 방식으로 사용 가능
- For an application that combines `*` and `--`, let's return to the stack example of Chapter 10. 10장의 스택을 활용하여 *와 --를 응용해보자
- The original version of the stack relied on an integer variable named `top` to keep track of the "top-of-stack" position in the contents array. 스택의 top의 위치를 추적하기 위해 정수 변수를 사용했었음
- Let's replace `top` by a pointer variable that points initially to element 0 of the `contents` array:
  top을 포인터 변수로 바꾸고 contents배열의 0번 요소를 가리키도록 하자

```
int *top_ptr = &contents[0];
```

Combining the * and ++ Operators

- The new `push` and `pop` functions:
  새로운 push 와 pop 함수

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
  포인터 연산이 배열과 포인터를 연관지어줌
- Another key relationship: 또 다른 관계

  The name of an array can be used as a pointer to the first element in the array.
  배열의 이름은 배열의 첫 번째 요소에 대한 포인터로 쓸 수 있음
- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.
  이 관계가 포인터 연산을 단순하게 만들고 배열과 포인터를 좀 더 유용하게 만듬

Using an Array Name as a Pointer

- Suppose that `a` is declared as follows: 다음과 같이 a를 선언했음

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

- Examples of using `a` as a pointer: a를 포인터로 쓰는 방법

```
*a = 7;    /* stores 7 in a[0] */
*(a+1) = 12;    /* stores 12 in a[1] */
```

- In general, `a + i` is the same as `&a[i]`. 일반적으로 a+i는 &a[i]와 같음
  - Both represent a pointer to element `i` of `a`.
    둘 다 배열 a의 i번째 요소에 대한 포인터를 나타냄
- Also, `*(a+i)` is equivalent to `a[i]`. *(a+i)는 a[i]와 같음
  - Both represent element `i` itself. 둘 다 요소 i를 가리킴

Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.
  배열 이름을 포인터로 쓸 수 있기 때문에 이를 활용하여 배열의 요소를 순회 가능
- Original loop: 일반적 루프

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

- Simplified version: 간략화 버전

```
for (p = a; p < a + N; p++)
  sum += *p;
```

Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it's not possible to assign it a new value.
  배열의 이름을 포인터로 쓸 수는 있지만, 새로운 값을 할당할 수는 없음
- Attempting to make it point elsewhere is an error:
  다른 위치를 가리키려고 하면 오류 발생

```
while (*a != 0)
  a++;                    /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:
  대신 포인터 변수에 배열 a를 복사하는 것으로 활용

```
p = a;
while (*p != 0)
  p++;
```

Program: Reversing a Series of Numbers (Revisited)

- The `reverse.c` program of Chapter 8 reads 10 numbers, then writes the numbers in reverse order.
  8장에서 reverse.c프로그램은 숫자를 역순으로 출력함
- The original program stores the numbers in an array, with subscripting used to access elements of the array.
  원 프로그램은 수를 배열에 저장하였고, 역순으로 배열 첨자를 순회하였음
- `reverse3.c` is a new version of the program in which subscripting has been replaced with pointer arithmetic.
  reverse3.c는 첨자 대신 포인터 연산을 활용함

reverse3.c

```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

배열을 함수의 인자로 전달

Array Arguments (Revisited)

- When passed to a function, an array name is treated as a pointer.
  배열을 함수로 전달할 때, 배열의 이름이 포인터로 처리됨
- Example:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

- A call of `find_largest`: find_largest 호출문은 다음과 같음

```
largest = find_largest(b, N);
```

  This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied. 이 호출문은 배열 b의 첫 요소의 포인터를 a에 할당

Array Arguments (Revisited)

- The fact that an array argument is treated as a pointer has some important consequences.
  배열을 인자로 쓸 때 포인터로 처리된다는 것에는 중요한 의미가 있음
- *Consequence 1:* When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.
  첫 번째 중요성: 일반 변수는 함수로 전달되면 값이 복사됨; 매개변수는 원래 변수에 영향을 주지 않음
- In contrast, an array used as an argument isn't protected against change.
  반면, 배열이 인자로 사용되면 원래 배열의 값이 매개변수에 의해 변경됨

Array Arguments (Revisited)

- For example, the following function modifies an array by storing zero into each of its elements:
  다음 함수는 배열의 모든 요소에 0을 저장함

```
void store_zeros(int a[], int n)
{
  int i;

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

Array Arguments (Revisited)

- To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:
  매개변수 배열이 원래 배열을 못 바꾸도록 하려면 const라는 키워드를 선언에 넣자

```
int find_largest(const int a[], int n)
{
  …
}
```

- If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.
  const 키워드가 존재하면, 컴파일러가 find_largest함수 내용에 a의 요소에 어떤 값을 할당하는지 검사함

Array Arguments (Revisited)

- *Consequence 2:* The time required to pass an array to a function doesn't depend on the size of the array.
  두 번째 중요성: 배열을 함수의 인자로 전달하는데 배열의 크기는 관계 없음
- There's no penalty for passing a large array, since no copy of the array is made.
  아주 긴 배열을 전달하더라도 배열의 복사본을 만드는 것이 아니기 때문에 오버헤드가 없음

Array Arguments (Revisited)

- *Consequence 3:* An array parameter can be declared as a pointer if desired.
  세 번째 중요성: 매개변수를 배열이 아니라 포인터로 선언할 수 있음
- `find_largest` could be defined as follows: 다음과 같이 변경 가능

```
int find_largest(int *a, int n)
{
  …
}
```

- Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.
  이 경우 a를 포인터로 선언하는 것은 배열로 선언하는 것과 똑같은 의미를 같음; 컴파일러가 둘을 동일하게 처리함

Array Arguments (Revisited)

- Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*.
  단, 포인터를 선언하는 것이 배열을 매개변수로 쓰는 것과 같은 효과가 있지만, 같은 것은 아님
- The following declaration causes the compiler to set aside space for 10 integers:
  다음 선언문은 만나면, 컴파일러는 10개의 정수를 저장할 공간을 만듬

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

- The following declaration causes the compiler to allocate space for a pointer variable:
  다음 선언문을 만나면 컴파일러는 정수형 포인터 변수를 저장할 공간을 만듬

```
int *a;
```

Array Arguments (Revisited)

- In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results.
  두 번째 경우 a는 배열이 아니기 때문에, 배열처럼 값을 저장하려고 하면 절대 안됨
- For example, the assignment

```
*a = 0;     /*** WRONG ***/
```

  will store 0 where `a` is pointing.
  위의 문장은 a가 가리키는 위치에 값을 0으로 저장함
- Since we don't know where `a` is pointing, the effect on the program is undefined.
  a가 어디를 가리키는지 알 수 없음으로, 결과적으로 정의되지 않은 동작을 하게 됨

Array Arguments (Revisited)

- *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.
  네 번째 중요성: 배열의 일부만 함수에 전달할 수 있음
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:
  다음의 예제는 find_largest에 배열의 5부터 14까지의 요소만 전달함

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

- C allows us to subscript a pointer as though it were an array name:
  c는 배열의 이름인 것처럼 포인터에 첨자를 사용 가능

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

The compiler treats `p[i]` as `*(p+i)`.
컴파일러는 p[i]를 *(p+i)로 처리함

심화 내용

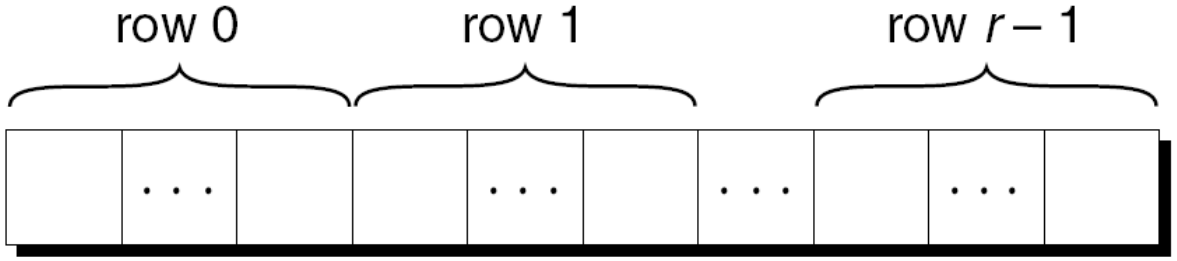
다차원 배열

Pointers and Multidimensional Arrays

- Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays.
  포인터가 일차원 배열의 요소를 가리킬 수 있듯이 다차원 배열의 요소도 가리킴
- This section explores common techniques for using pointers to process the elements of multidimensional arrays.
  포인터를 다차원 배열에 사용하는 일반적인 기법을 설명함

Processing the Elements of a Multidimensional Array

- Chapter 8 showed that C stores two-dimensional arrays in row-major order.
  8장에서 2차원 배열은 줄 단위로 데이터를 저장한다고 했음
- Layout of an array with *r* rows: r개의 줄을 갖는 배열의 모습



- If `p` initially points to the element in row 0, column 0, we can visit every element in the array by incrementing `p` repeatedly.
  최초에 p가 배열의 0, 0 위치의 요소를 가리킨다면, p를 반복적으로 증가시켜서 배열의 요소를 순회할 수 있음

Processing the Elements of a Multidimensional Array

- Consider the problem of initializing all elements of the following array to zero: 다음의 배열의 모든 요소를 0으로 초기화한다고 하자

```
int a[NUM_ROWS][NUM_COLS];
```

- The obvious technique would be to use nested `for` loops: for루프를 사용할 수 있을 것임

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

- If we view `a` as a one-dimensional array of integers, a single loop is sufficient: 만약 a를 1차원 배열로 인식한다면 루프 하나면 충분히 초기화 가능

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

Processing the Elements of a Multidimensional Array

- Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers.
  2차원 배열을 1차원 배열처럼 다루는 것이 이상해 보여도 많은 컴파일러가 허용함
- Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency.
  이렇게 쓰면 프로그램을 읽기가 어려워지겠지만, 옛날 컴파일러에서는 성능이 좋아질 수 있음
- With many modern compilers, though, there's often little or no speed advantage.
  현대의 컴파일러는 성능이득을 찾아보기 어려움

Processing the Rows of a Multidimensional Array

- A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.
  포인터 변수 p는 2차원 배열의 줄의 시작만 가리키도록 할 수 있음
- To visit the elements of row `i`, we'd initialize `p` to point to element 0 in row `i` in the array `a`:
  i번째 줄의 요소를 방문하기 위해 p가 배열 a의 i번째 줄의 0번째 요소를 가리키도록 하면 됨

```
p = &a[i][0];
```

or we could simply write
또는 간단하게 다음처럼 쓸 수 있음

```
p = a[i];
```

Processing the Rowsof a Multidimensional Array

- For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first element in row `i`.
  모든 2차원 배열 a에 대해, a[i]는 i번째 줄의 첫 번째 요소에 대한 포인터임
- To see why this works, recall that `a[i]` is equivalent to `*(a + i)`.
  a[i]는 *(a+i)로 바꿔 쓸 수 있다고 했음
- Thus, `&a[i][0]` is the same as `&(*(a[i] + 0))`, which is equivalent to `&*a[i]`.
  그러므로 &a[i][0]은 &(*(a[i]+0))과 같고 정리하면 &*a[i]가 됨
- This is the same as `a[i]`, since the `&` and `*` operators cancel. &와 *연산자가 서로 상쇄하므로 a[i]가 됨

Processing the Rows of a Multidimensional Array

- A loop that clears row `i` of the array `a`:
  배열 a의 i번째 줄을 초기화하는 루프

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

- Since `a[i]` is a pointer to row `i` of the array `a`, we can pass `a[i]` to a function that's expecting a one-dimensional array as its argument.
  a[i]가 배열 a의 i번째 줄의 포인터이기 때문에, a[i]를 함수에 전달한다는 의미는 1차원 배열을 인자로 전달한다는 의미로 쓸 수 있음
- In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.
  다시 말하면 1차원 배열에 사용가능한 함수에 2차원 배열을 전달해도 쓸 수 있음

Processing the Rows of a Multidimensional Array

- Consider `find_largest`, which was originally designed to find the largest element of a one-dimensional array.
  find_largest는 1차원 배열에서 가장 큰 값을 찾는 함수였음
- We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:
  다음처럼 2차원 배열의 한 줄을 인자로 전달하여 i번째 줄의 가장 큰 값을 찾도록 할 수 있음

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the Columns of a Multidimensional Array

- Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.
  배열을 열단위로 처리하는 것은 간단한 문제가 아님; 배열요소가 줄단위로 저장되어 있기 때문임
- A loop that clears column `i` of the array `a`:
  배열 a의 i번째 열의 값을 초기화 하는 코드

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

Using the Name of a Multidimensional Array as a Pointer

- The name of *any* array can be used as a pointer, regardless of how many dimensions it has, but some care is required.
  배열의 차원과 관계 없이 배열의 이름을 포인터로 사용할 수 있음
- Example:

  ```
  int a[NUM_ROWS][NUM_COLS];
  ```

  `a` is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`.
  a는 a[0][0]에 대한 포인터가 아니라 a[0]에 대한 포인터임
- C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.
  C는 a를 일차원 배열로 인식함
- When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).
  포인터로 사용할 경우 a는 int (*)[NUM_COLS]로 이해함; 해석하면 NUM_COLS길이를 갖는 정수형 배열에 대한 포인터

Using the Name of a Multidimensional Array as a Pointer

- Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array.
  a[0]의 위치를 a가 가리킨다는 사실을 활용하면 2차원 배열의 순회하는 루프를 작성이 간단해짐
- Instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

  to clear column `i` of the array `a`, we can write
  배열 a의 i 번째 열을 초기화하기위해 아래처럼 쓸 수 있음

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```

Using the Name of a Multidimensional Array as a Pointer

- We can “trick” a function into thinking that a multidimensional array is really one-dimensional.
  2차원 배열이 마치 1차원 배열인것처럼 인식하도록 할 수 있음
- A first attempt at using using `find_largest` to find the largest element in `a`: find_largest함수로 a의 가장 큰 요소를 찾는 문장을 보자

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);
  /* WRONG */
```

  This an error, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`.
  위 문장에서 find_largest는 int * 타입을 인자로 기대하는데 a는 int (*)[NUM_COLS]의 타입을 갖고 있기 때문에 오류가 발생함

Using the Name of a Multidimensional Array as a Pointer

- We can "trick" a function into thinking that a multidimensional array is really one-dimensional.
  2차원 배열이 마치 1차원 배열인것처럼 인식하도록 할 수 있음

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);
  /* WRONG */
```

- The correct call:

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

  `a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler).
  제대로 부르려면 1차원 배열로 전달해야 함; a[0]은 0번 줄에 0번째 요소를 가리키고 있고 int * 타입을 갖음

포인터와 가변길이 배열

Pointers and Variable-Length Arrays (C99)

- Pointers are allowed to point to elements of variable-length arrays (VLAs).
  포인터는 가변 길이의 배열의 요소를 가리킬 수 있음
- An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:
  평범한 포인터 변수로 일차원 가변길이 배열의 요소를 가리킬 수 있음

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

Pointers and Variable-Length Arrays (C99)

- When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first.
  가변길이 배열이 다차원인 경우 포인터의 종류는 각 차원의 길이에 의존함
- A two-dimensional example: 2차원 가변 길이의 예제

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.***
p의 타입은 n에 의존하기 때문에 p는가변적으로 변형되는 타입이라 부름

Pointers and Variable-Length Arrays (C99)

- The validity of an assignment such as `p = a` can't always be determined by the compiler.
  p = a과 같은 할당은 컴파일러에 의해 유효성이 항상 검증되는 것은 아님
- The following code will compile but is correct only if `m` and `n` are equal:
  아래와 같은 문장은 컴파일은 되지만, m과 n과 동일한 경우만 제대로 동작함

```
int a[m][n], (*p)[m];
p = a;
```

- If `m` is not equal to `n`, any subsequent use of `p` will cause undefined behavior.
  m과 n이 동일하지 않d은데 p를 사용하게 되면 오동작을 함

Pointers and Variable-Length Arrays (C99)

- Variably modified types are subject to certain restrictions.
  가변적으로 변형되는 타입은 제한 사항이 있음
- The most important restriction: the declaration of a variably modified type must be inside the body of a function or in a function prototype.
  가장 중요한 제한사항: 가변길이 배열에 대한 선언은 함수 내용 중에 포함되어 있거나 함수 프로토타입에 정의되어야 함

Pointers and Variable-Length Arrays (C99)

- Pointer arithmetic works with VLAs.
  포인터 연산은 가변길이 배열에서도 적용됨
- A two-dimensional VLA: 2차원 가변길이 배열이 있다고 하자

```
int a[m][n];
```

- A pointer capable of pointing to a row of `a`:
  a의 어떤 줄을 가리킬 수 있는 p는 다음처럼 선언함

```
int (*p)[n];
```

- A loop that clears column `i` of `a`:
  a의 i번째 열의 초기화는 다음 같음

```
for (p = a; p < a + m; p++)
  (*p)[i] = 0;
```