Arrays, Functions

adopted from KNK C Programming : A Modern Approach

Arrays

adopted from KNK C Programming : A Modern Approach

Program: Reversing a Series of Numbers

- The `reverse.c` program prompts the user to enter a series of numbers, then writes the numbers in reverse order: 사용자에게 수를 입력받아 반대로 출력하자

```
 Enter 10 numbers: 34 82 49 102 7 94 23 11 50 31
In reverse order: 31 50 11 23 94 7 102 49 82 34
```

- The program stores the numbers in an array as they're read, then goes through the array backwards, printing the elements one by one.
- 이 프로그램은 사용자 입력 숫자를 배열에 저장한 후, 배열 뒤에서부터 앞으로 하나씩 출력한다

reverse.c

```
/* Reverses a series of numbers */

#include <stdio.h>

#define N 10

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

  printf("Enter %d numbers: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);

  printf("In reverse order:");
  for (i = N - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

Program: Checking a Number for Repeated Digits

- The `repdigit.c` program checks whether any of the digits in a number appear more than once.

  입력한 수에 0..9까지의 숫자가 다시 나타나는지 체크

- After the user enters a number, the program prints either `Repeated digit` or `No repeated digit`:

  반복이 되면 `Repeated digit` 를 아니면 `No repeated digit` 를 출력

```
Enter a number: 28212
Repeated digit
```

- The number 28212 has a repeated digit (2); a number like 9357 doesn't.

  예를 들어 28212에서는 2가 반복

Program: Checking a Number for Repeated Digits

- The program uses an array of 10 Boolean values to keep track of which digits appear in a number. 어떤 0…9 숫자가 나타나는지를 파악하기 위해 10자리 배열을 사용할 수 있다.

- Initially, every element of the `digit_seen` array is false.

  초기에 배열의 모든 요소는 0이다.

Program: Checking a Number for Repeated Digits

- When given a number `n`, the program examines `n`’s digits one at a time, storing the current digit in a variable named `digit`.
- 수 n에 대하여 프로그램은 각 자리 숫자를 조사한다.

  - If `digit_seen[digit]` is true, then `digit` appears at least twice in `n`.
  - 배열의 0..9이 0이 아니면, 최소한 한번 이상 존재했다.

  - If `digit_seen[digit]` is false, then `digit` has not been seen before, so the program sets `digit_seen[digit]` to `true` and keeps going.
  - 배열의 0..9이 0이면, 해당 숫자는 아직 나타난 적이 없다.

repdigit.c

```
/* Checks numbers for repeated digits */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

int main(void)
{
  bool digit_seen[10] = {false};
  int digit;
  long n;

  printf("Enter a number: ");
  scanf("%ld", &n);
  while (n > 0) {
    digit = n % 10;
    if (digit_seen[digit])
      break;
    digit_seen[digit] = true;
    n /= 10;
  }
  if (n > 0)
    printf("Repeated digit\n");
  else
    printf("No repeated digit\n");

  return 0;
}
```

Program: Computing Interest

- The `interest.c` program prints a table showing the value of $100 invested at different rates of interest over a period of years.
- 100달러에 대해서 수년간 다른 이자율에 대한 테이블을 출력한다.

- The user will enter an interest rate and the number of years the money will be invested.
- 사용자는 이자율과 투자할 기간(년)을 입력한다.

- The table will show the value of the money at one-year intervals—at that interest rate and the next four higher rates—assuming that interest is compounded once a year.
- 테이블은 1년 단위로 투자 결과를 보여준다.
- 이자는 복리로 계산한다.

Program: Computing Interest

- Here's what a session with the program will look like:

```
Enter interest rate: 6
Enter number of years: 5


Years     6%     7%     8%     9%    10%
  1    106.00 107.00 108.00 109.00 110.00
  2    112.36 114.49 116.64 118.81 121.00
  3    119.10 122.50 125.97 129.50 133.10
  4    126.25 131.08 136.05 141.16 146.41
  5    133.82 140.26 146.93 153.86 161.05
```

Program: Computing Interest

- The numbers in the second row depend on the numbers in the first row, so it makes sense to store the first row in an array.
- 두번째 연도의 수는 첫번째 연도의 수에 의존한다. 따라서 첫번째 연도의 결과를 배열로 저장한다.
  - The values in the array are then used to compute the second row.
  - 배열의 값은 두 번째 연도의 결과를 계산하는데 사용한다.
  - This process can be repeated for the third and later rows.
  - 이러한 과정은 세번째 이후의 연도의 결과를 계산하는데 사용

- The program uses nested `for` statements. 중첩 for문 사용
  - The outer loop counts from 1 to the number of years requested by the user.
  - 처음 for문은 연도의 증가
  - The inner loop increments the interest rate from its lowest value to its highest value. 두번째 for문은 이자율의 증가

interest.c

```
/* Prints a table of compound interest */

#include <stdio.h>

#define NUM_RATES ((int) (sizeof(value) / sizeof(value[0])))
#define INITIAL_BALANCE 100.00

int main(void)
{
  int i, low_rate, num_years, year;
  double value[5];

  printf("Enter interest rate: ");
  scanf("%d", &low_rate);
  printf("Enter number of years: ");
  scanf("%d", &num_years);
  printf("\nYears");
  for (i = 0; i < NUM_RATES; i++) {
    printf("%6d%%", low_rate + i);
    value[i] = INITIAL_BALANCE;
  }
  printf("\n");

  for (year = 1; year <= num_years; year++) {
    printf("%3d    ", year);
    for (i = 0; i < NUM_RATES; i++) {
      value[i] += (low_rate + i) / 100.0 * value[i];
      printf("%7.2f", value[i]);
    }
    printf("\n");
  }

  return 0;
}
```

Program: Dealing a Hand of Cards

- The `deal.c` program illustrates both two-dimensional arrays and constant arrays.
- 2차원 상수 배열을 보이는 프로그램

- The program deals a random hand from a standard deck of playing cards.
- 카드를 랜덤하게 다룬다.

- Each card in a standard deck has a *suit* (clubs, diamonds, hearts, or spades) and a *rank* (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, or ace).
- 각 카드는 클럽, 다이아몬드, 하트, 스페이드로 구분되며 2부터 왕까지 그리고 에이스까지의 번호를 가진다.

Program: Dealing a Hand of Cards

- The user will specify how many cards should be in the hand:
  사용자는 얼마나 많은 카드를 가질지 입력한다.

```
Enter number of cards in hand: 5
Your hand: 7c 2s 5d as 2h
```

- Problems to be solved: 풀어야 할 문제
  - How do we pick cards randomly from the deck?
  - 어떻게 카드를 랜덤하게 고를 것인가?
  - How do we avoid picking the same card twice?
  - 어떻게 동일한 카드를 고르는 것을 피할 것인가?

Program: Dealing a Hand of Cards

- To pick cards randomly, we'll use several C library functions: 랜덤 함수를 위해 아래 라이브러리 사용
  - `time` (from `<time.h>`) – returns the current time, encoded in a single number.
  - `srand` (from `<stdlib.h>`) – initializes C's random number generator.
  - `rand` (from `<stdlib.h>`) – produces an apparently random number each time it's called.
- By using the `%` operator, we can scale the return value from `rand` so that it falls between 0 and 3 (for suits) or between 0 and 12 (for ranks). 일정 구간 내 있도록 %연산 사용

Program: Dealing a Hand of Cards

- The `in_hand` array is used to keep track of which cards have already been chosen.
- 어떤 카드가 선택되었는지를 파악하기 위해 배열 사용
- The array has 4 rows and 13 columns; each element corresponds to one of the 52 cards in the deck.
- All elements of the array will be false to start with.
- Each time we pick a card at random, we'll check whether the element of `in_hand` corresponding to that card is true or false. 선택을 했을 때 이미 선택한 카드인지 체크
  - If it's true, we'll have to pick another card. 이미 선택했으면 다른 카드 고름
  - If it's false, we'll store `true` in that element to remind us later that this card has already been picked. 아니면 선택했음 기입

Program: Dealing a Hand of Cards

- Once we've verified that a card is "new," we'll need to translate its numerical rank and suit into characters and then display the card.
- 새로운 카드를 선택했으면, 그의 종류와 숫자 출력
- To translate the rank and suit to character form, we'll set up two arrays of characters—one for the rank and one for the suit—and then use the numbers to subscript the arrays. 이를 위해서는 2개의 배열이 필요
- These arrays won't change during program execution, so they are declared to be `const`.
- 종류와 숫자는 프로그램 실행 동안 변하지 않음. 따라서 상수로 선언

deal.c

```
/* Deals a random hand of cards */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUM_SUITS 4
#define NUM_RANKS 13

int main(void)
{
  bool in_hand[NUM_SUITS][NUM_RANKS] = {false};
  int num_cards, rank, suit;
  const char rank_code[] = {'2','3','4','5','6','7','8',
                            '9','t','j','q','k','a'};
  const char suit_code[] = {'c','d','h','s'};
  srand((unsigned) time(NULL));

  printf("Enter number of cards in hand: ");
  scanf("%d", &num_cards);

  printf("Your hand:");
  while (num_cards > 0) {
    suit = rand() % NUM_SUITS;     /* picks a random suit */
    rank = rand() % NUM_RANKS;     /* picks a random rank */
    if (!in_hand[suit][rank]) {
      in_hand[suit][rank] = true;
      num_cards--;
      printf(" %c%c", rank_code[rank], suit_code[suit]);
    }
  }
  printf("\n");

  return 0;
}
```

Variable-Length Arrays (C99)

Variable-Length Arrays (C99)

- In C89, the length of an array variable must be specified by a constant expression.
- In C99, however, it’s sometimes possible to use an expression that’s *not* constant.
- The `reverse2.c` program—a modification of `reverse.c`—illustrates this ability.

reverse2.c

```
/* Reverses a series of numbers using a variable-length
   array - C99 only */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("How many numbers do you want to reverse? ");
  scanf("%d", &n);

  int a[n];   /* C99 only - length of array depends on n */

  printf("Enter %d numbers: ", n);

  for (i = 0; i < n; i++)

    scanf("%d", &a[i]);
 printf("In reverse order:");
  for (i = n - 1; i >= 0; i--)
    printf(" %d", a[i]);
  printf("\n");

  return 0;
}
```

Variable-Length Arrays (C99)

- The array `a` in the `reverse2.c` program is an example of a ***variable-length array*** (or ***VLA***).
- The length of a VLA is computed when the program is executed.
- The chief advantage of a VLA is that a program can calculate exactly how many elements are needed.
- If the programmer makes the choice, it's likely that the array will be too long (wasting memory) or too short (causing the program to fail).

Variable-Length Arrays (C99)

- The length of a VLA doesn't have to be specified by a single variable. Arbitrary expressions are legal:

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

- Like other arrays, VLAs can be multidimensional:

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

- Restrictions on VLAs:
  - Can't have static storage duration (discussed in Chapter 18).
  - Can't have an initializer.

Functions

adopted from KNK C Programming : A Modern Approach

Defining and Calling Functions

- Before we go over the formal rules for defining a function, let’s look at three simple programs that define functions.
  구체적인 규칙을 보기 전에, 예를 살펴보자

Program: Computing Averages

- A function named `average` that computes the average of two `double` values: average라는 함수는 두 double 타입의 값의 평균을 구함

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- The word `double` at the beginning is the ***return type*** of `average`. 함수의 시작 위치의 double이라는 것은 함수 종료시 리턴 타입
- The identifiers `a` and `b` (the function's ***parameters***) represent the numbers that will be supplied when `average` is called.
  a 와 b는 함수의 매개변수로서 average함수가 호출될 때 전달되는 값

Program: Computing Averages

- Every function has an executable part, called the ***body,*** which is enclosed in braces. 모든 함수에는 내용(body)이 중괄호 내에 존재함
- The body of `average` consists of a single `return` statement. average의 경우 내용으로는 하나의 리턴문만 존재
- Executing this statement causes the function to "return" to the place from which it was called; the value of `(a + b) / 2` will be the value returned by the function.
  이 리턴 문을 실행하면 호출한 곳에 (a+b)/2의 계산 결과를 전달

Program: Computing Averages

- A function call consists of a function name followed by a list of ***arguments.*** 함수 호출문은 함수의 이름과 인자들로 구성됨
  - `average(x, y)` is a call of the `average` function.
    average(x, y)라고 쓰면 average라는 함수를 호출함
- Arguments are used to supply information to a function.
  인자는 함수에 정보를 전달할 때 사용함
  - The call `average(x, y)` causes the values of `x` and `y` to be copied into the parameters `a` and `b`.
    average(x, y)를 호출하면 x와 y의 값이 매개변수 a와 b에 각각 복사됨
- An argument doesn't have to be a variable; any expression of a compatible type will do. 수식과 숫자도 인자로 인정
  - `average(5.1, 8.9)` and `average(x/2, y/3)` are legal.

Program: Computing Averages

- We'll put the call of `average` in the place where we need to use the return value. 평균의 결과과 필요한 위치에 average를 호출함
- A statement that prints the average of `x` and `y`: 두 수의 평균을 구함

```
printf("Average: %g\n", average(x, y));
```

  The return value of `average` isn't saved; the program prints it and then discards it. 평균 계산 결과는 printf문에서 활용 후 버림

- If we had needed the return value later in the program, we could have captured it in a variable: 결과 재활용시 변수에 결과를 할당

```
avg = average(x, y);
```

Program: Computing Averages

- The `average.c` program reads three numbers and uses the `average` function to compute their averages, one pair at a time:
  세 숫자를 입력받아 두 수 씩 평균을 구함

```
Enter three numbers: 3.5 9.6 10.2
Average of 3.5 and 9.6: 6.55
Average of 9.6 and 10.2: 9.9
Average of 3.5 and 10.2: 6.85
```

average.c

```c
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
  return (a + b) / 2;
}

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}
```

Program: Printing a Countdown

- To indicate that a function has no return value, we specify that its return type is `void`: 함수에 리턴 값이 없으면 void로 표기

```
void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}
```

- `void` is a type with no values. void는 값 없음을 뜻함
- A call of `print_count` must appear in a statement by itself:

```
print_count(i); 독립적인 문장으로 호출됨
```

- The `countdown.c` program calls `print_count` 10 times inside a loop. 10회 반복

countdown.c

```
/* Prints a countdown */

#include <stdio.h>

void print_count(int n)
{
  printf("T minus %d and counting\n", n);
}

int main(void)
{
  int i;

  for (i = 10; i > 0; --i)
    print_count(i);

  return 0;
}
```

Program: Printing a Pun (Revisited)

- When a function has no parameters, the word `void` is placed in parentheses after the function’s name:
  매개변수가 없으면 함수 명 뒤의 괄호 안에 void로 표기

  ```
  void print_pun(void)
  {
    printf("To C, or not to C: that is the question.\n");
  }
  ```

- To call a function with no arguments, we write the function’s name, followed by parentheses: 매개변수 없는 함수 호출시 함수명과 괄호만

  ```
  print_pun();
  ```

  The parentheses *must* be present. 괄호가 꼭 필요

- The `pun2.c` program tests the `print_pun` function.

pun2.c

```
/* Prints a bad pun */

#include <stdio.h>

void print_pun(void)
{
  printf("To C, or not to C: that is the question.\n");
}

int main(void)
{
  print_pun();
  return 0;
}
```

Program: Testing Whether a Number Is Prime

- The `prime.c` program tests whether a number is prime:
  소수 찾는 프로그램

```
Enter a number: 34
Not prime
```

- The program uses a function named `is_prime` that returns `true` if its parameter is a prime number and `false` if it isn't.
  함수 is_prime은 매개 변수가 소수이면 true 아니면 false를 리턴
- `is_prime` divides its parameter `n` by each of the numbers between 2 and the square root of `n`; if the remainder is ever 0, `n` isn't prime.
  매개 변수 n을 2 부터 n의 제곱근 사이의 수로 나눔; 나머지가 0이면 n은 소수

prime.c

```
/* Tests whether a number is prime */

#include <stdbool.h>   /* C99 only */
#include <stdio.h>

bool is_prime(int n)
{
  int divisor;

  if (n <= 1)
    return false;
  for (divisor = 2; divisor * divisor <= n; divisor++)
    if (n % divisor == 0)
      return false;
  return true;
}
```

```
int main(void)
{
  int n;

  printf("Enter a number: ");
  scanf("%d", &n);
  if (is_prime(n))
    printf("Prime\n");
  else
    printf("Not prime\n");
  return 0;
}
```

Array Arguments

- A function is allowed to change the elements of an array parameter, and the change is reflected in the corresponding argument.
- A function that modifies an array by storing zero into each of its elements:

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

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

The Quicksort Algorithm

- Recursion is most helpful for sophisticated algorithms that require a function to call itself two or more times.
  자기 스스로를 두 번 이상 불러야 하는 복잡한 알고리즘에서 재귀 호출이 유용함
- Recursion often arises as a result of an algorithm design technique known as ***divide-and-conquer,*** in which a large problem is divided into smaller pieces that are then tackled by the same algorithm.
  분할정복(divide-and-conquer) 기법에서 재귀법이 활용됨. 큰 문제를 작은 단위의 문제로 나누어 풀고 결과를 합침

The Quicksort Algorithm

- A classic example of divide-and-conquer can be found in the popular ***Quicksort*** algorithm.
  분할정복 기법의 고전인 quicksort(퀵소트 정렬 기법)을 살펴보자
- Assume that the array to be sorted is indexed from 1 to *n.*
  정렬할 배열의 인덱스는 1에서 n까지라 하자

Quicksort algorithm

1. Choose an array element *e* (the "partitioning element"), then rearrange the array so that elements 1, …, *i* – 1 are less than or equal to *e*, element *i* contains e, and elements *i* + 1, …, *n* are greater than or equal to *e*.
   임의의 배열의 원소 e를 선택, e 왼쪽은 작은 값, e오른쪽은 큰 값을 배치
2. Sort elements 1, …, *i* – 1 by using Quicksort recursively.
   왼편을 재귀법을 써서 정렬
3. Sort elements *i* + 1, …, *n* by using Quicksort recursively.
   오른편을 재귀법을 써서 정렬

The Quicksort Algorithm

- Step 1 of the Quicksort algorithm is obviously critical.
  1단계가 매우 중요함
- There are various methods to partition an array.
  파티션하는 방법은 다양함
- We’ll use a technique that’s easy to understand but not particularly efficient.
  이해하는 쉽지만 아주 효율적이지는 않은 방법을 사용하여 설명
- The algorithm relies on two “markers” named *low* and *high*, which keep track of positions within the array.
  두 개의 표식(low와 high)을 써서 배열의 값들을 처리함

The Quicksort Algorithm

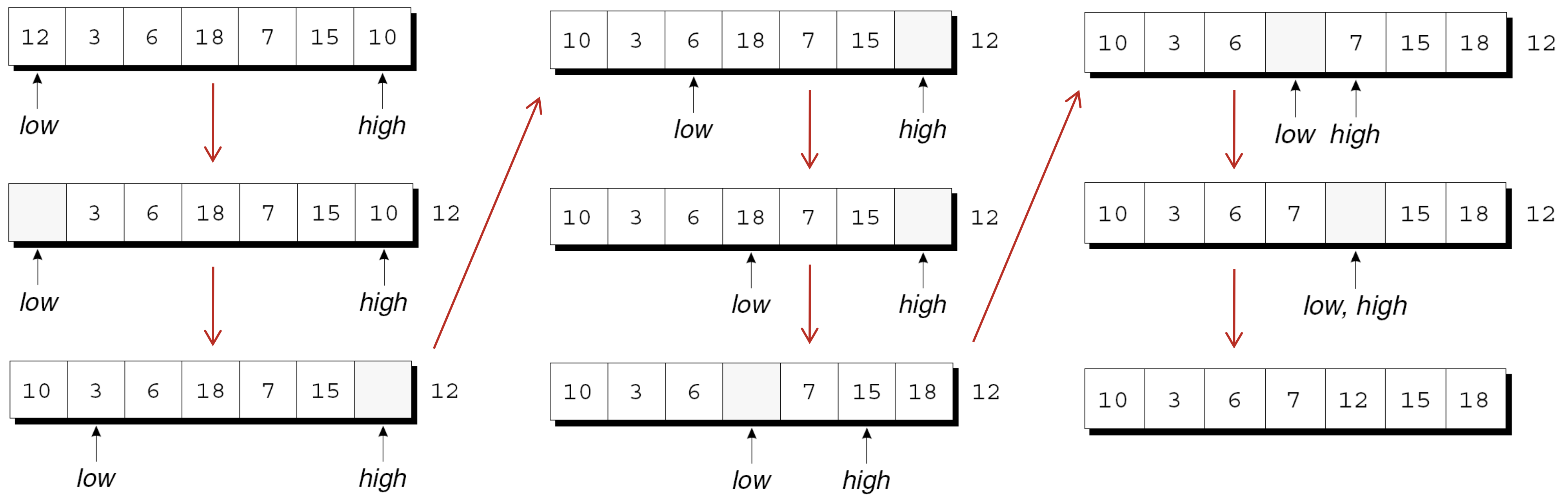
- Initially, *low* points to the first element; *high* points to the last.
  최초에는 low는 첫번째 원소, high는 마지막 원소를 가리킴
- We copy the first element (the partitioning element) into a temporary location, leaving a "hole" in the array.
  첫 원소를 임시 위치에 저장하여 배열에 구멍을 만듬
- Next, we move *high* across the array from right to left until it points to an element that's smaller than the partitioning element.
  다음, 파티션 값 e보다 작은 값을 만날 때까지 high 위치의 값을 오른쪽에서 왼쪽으로 이동

The Quicksort Algorithm

- We then copy the element into the hole that *low* points to, which creates a new hole (pointed to by *high*).
  작은 값을 만나면 그 값을 low가 가리키던 위치로 이동; 새로운 구멍이 생김
- We now move *low* from left to right, looking for an element that's larger than the partitioning element. When we find one, we copy it into the hole that *high* points to.
  파티션 값보다 큰 값을 찾을 때까지 low를 왼쪽에서 오른쪽으로 이동, high가 만든 구멍으로 이동
- The process repeats until *low* and *high* meet at a hole.
  low와 high가 서로 같은 구멍에서 만날 때까지 반복
- Finally, we copy the partitioning element into the hole.
  파티션 값을 구멍에 복사

The Quicksort Algorithm

- Example of partitioning an array: 예시



The Quicksort Algorithm

- By the final figure, all elements to the left of the partitioning element are less than or equal to 12, and all elements to the right are greater than or equal to 12.
  파티션 값 12보다 작거나 같은 값은 모두 왼쪽, 파티션 보다 큰 값은 모두 오른쪽
- Now that the array has been partitioned, we can use Quicksort recursively to sort the first four elements of the array (10, 3, 6, and 7) and the last two (15 and 18).
  파티션 후, 퀵소트를 재귀적으로 호출하여 첫 4개의 원소를 정렬하고, 15와 18도 퀵소트로 정렬함

Program: Quicksort

- Let's develop a recursive function named `quicksort` that uses the Quicksort algorithm to sort an array of integers.
  재귀 함수 quicksort를 써서 퀵소트 정렬을 구현해보자
- The `qsort.c` program reads 10 numbers into an array, calls `quicksort` to sort the array, then prints the elements in the array:
  10개의 숫자를 배열에 저장한 후 quicksort를 호출하여 정렬
  정렬후 결과 출력

```
Enter 10 numbers to be sorted: 9 16 47 82 4 66 12 3 25 51
In sorted order: 3 4 9 12 16 25 47 51 66 82
```

- The code for partitioning the array is in a separate function named `split`.
  배열을 파티션하는 것은 split이라는 다른 함수를 씀

qsort.c

```
/* Sorts an array of integers using Quicksort algorithm */

#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

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

  printf("Enter %d numbers to be sorted: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
  quicksort(a, 0, N - 1);

  printf("In sorted order: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

```
void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}
```

```
int split(int a[], int low, int high)
{
  int part_element = a[low];

  for (;;) {
    while (low < high && part_element <= a[high])
      high--;
    if (low >= high) break;
    a[low++] = a[high];

    while (low < high && a[low] <= part_element)
      low++;
    if (low >= high) break;
    a[high--] = a[low];
  }

  a[high] = part_element;
  return high;
}
```

Program: Quicksort

- Ways to improve the program's performance:
  성능 개선 방법
  - Improve the partitioning algorithm.
    파티션 알고리즘 개선
  - Use a different method to sort small arrays.
    배열의 크기가 작은 경우 다른 방법을 씀
  - Make Quicksort nonrecursive.
    퀵소트를 비재귀적으로 만듬