



제2장

Case Study: 전화번호부

전화번호부.v1

실행 예

프로그램을 실행하면 화면에 프롬프트(\$)를 출력하고 사용자의 명령을 기다린다.

```
$ add John 01076769898
John was added successfully.
$ add David 0517778888
David was added successfully.
$ find Henry
No person named 'Henry' exists.
$ find David
0517778888
$ status
John 01076769898
David 0517778888
Total 2 persons.
$ delete Jim
No person named 'Jim' exists.
$ delete John
John was deleted successfully.
$ status
David 0517778888
Total 1 person.
$ exit
```

새로운 사람을 추가한다.

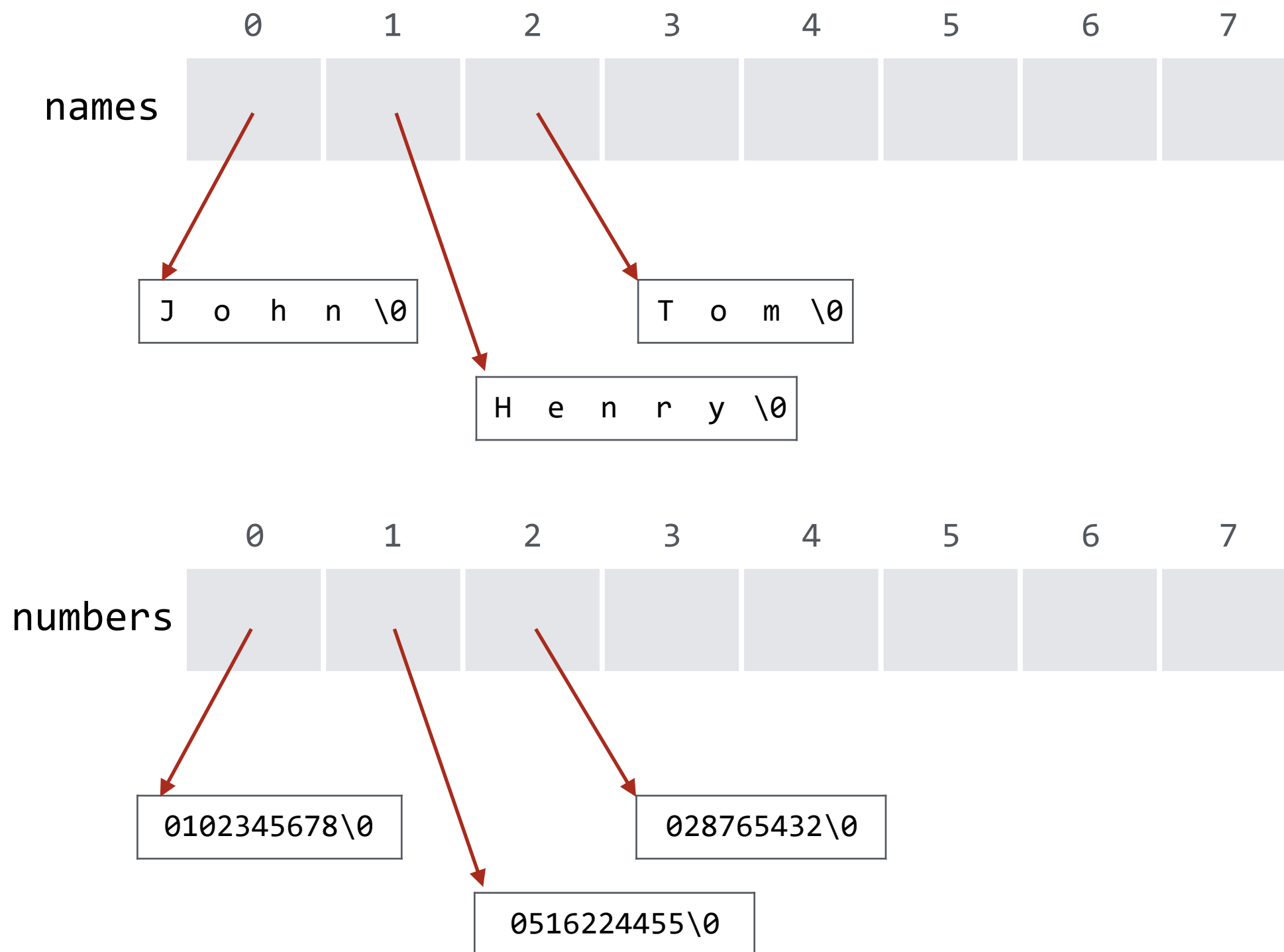
이름으로 전화번호를 검색한다.

전화번호부에 저장된 모든 사람을 출력한다.

전화번호부에서 삭제한다..

프로그램을 종료한다.

자료구조



phonebook01.c

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

```
#define CAPACITY 100
#define BUFFER_SIZE 20
```

← 최대 100명을 저장한다.

names와 numbers는 char * 타입의 배열이다.

```
char * names[CAPACITY]; /* names */
char * numbers[CAPACITY]; /* phone numbers */
int n = 0; /* number of people in phone directory */

void add();
void find();
void status();
void remove();
```

phonebook01.c

```
int main() {
    char command[BUFFER_SIZE];
    while (1) {
        printf("$ ");
        scanf("%s", command);

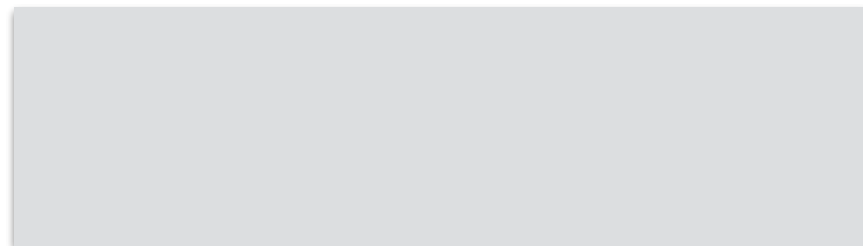
        if (strcmp(command, "add")==0)
            add();
        else if (strcmp(command, "find")==0)
            find();
        else if (strcmp(command, "status")==0)
            status();
        else if (strcmp(command, "delete")==0)
            remove();
        else if (strcmp(command, "exit")==0)
            break;
    }
    return 0;
}
```

strcmp함수는 두 문자열이 동일하면 0을 반환한다.



phonebook01.c

```
void add() {  
    char buf1[BUFFER_SIZE], buf2[BUFFER_SIZE];  
    scanf("%s", buf1);  
    scanf("%s", buf2);
```



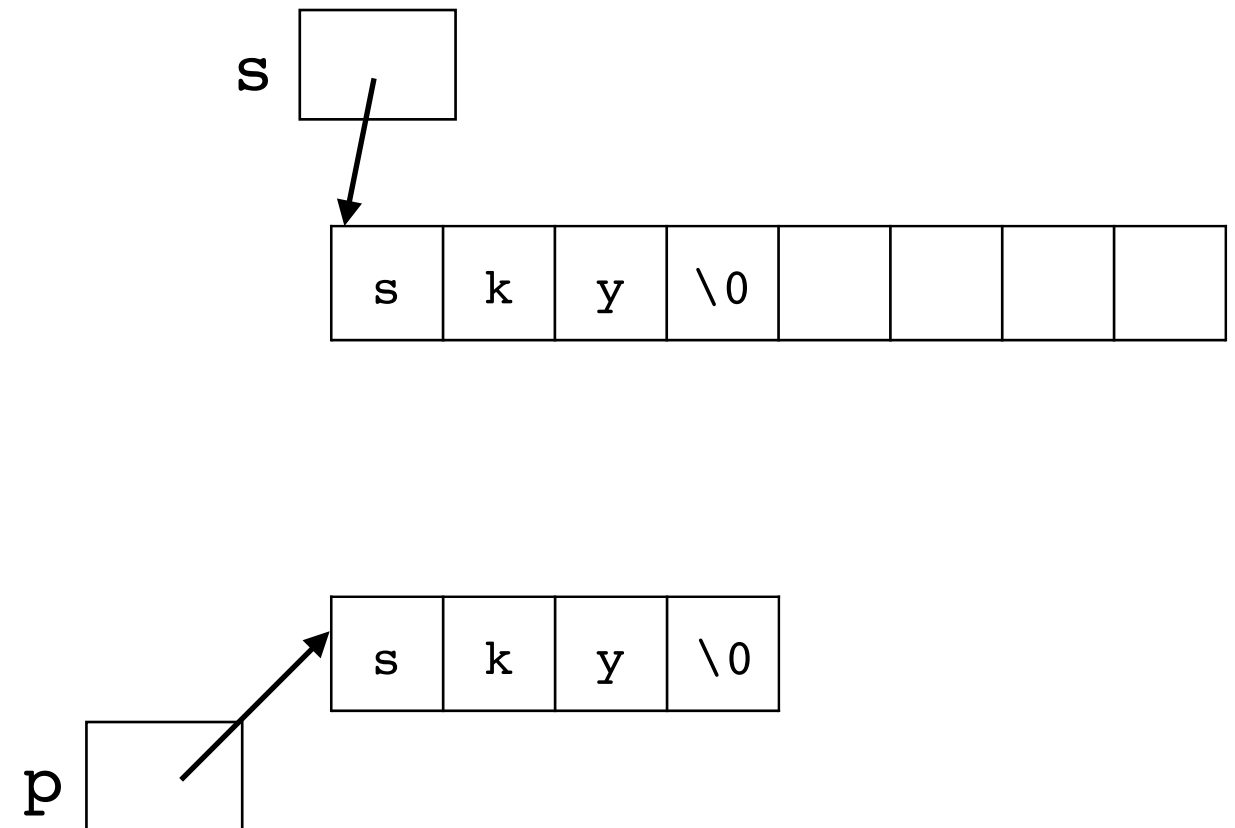
← strdup와 strcpy의 차이는 ?

```
    printf("%s was added successfully.\n", buf1);  
}
```

strdup

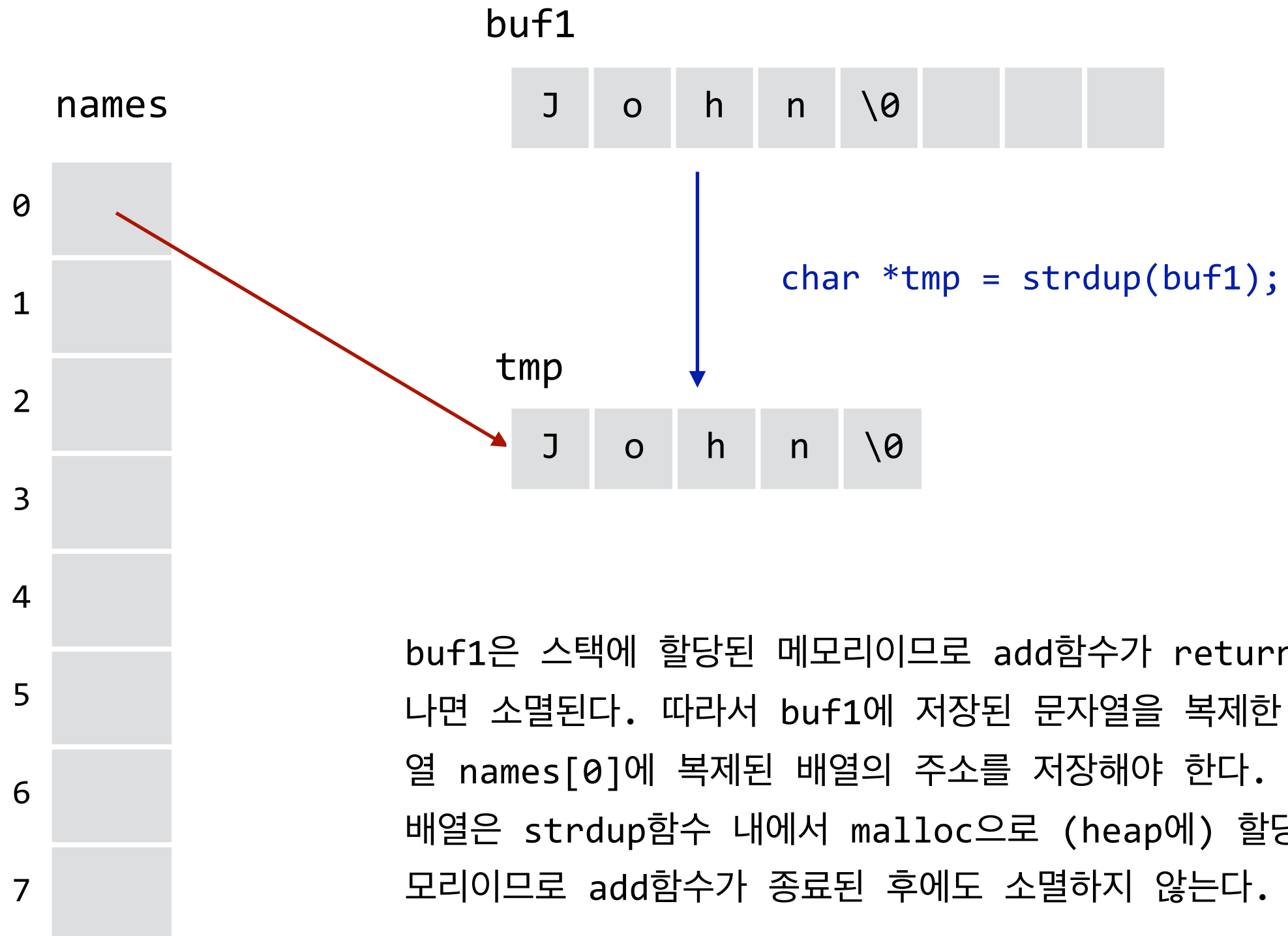
strdup는 string.h 라이브러리가 제공하므로
직접 구현할 필요는 없다. 아래의 코드는 참조용이다.

```
char *strdup(char *s)
{
    char *p;
    p = (char *)malloc(strlen(s)+1);
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```



strdup는 배열을 만들고
매개변수로 받은 하나의 문자열을 거기에 복사하여 반환한다.
strcpy와의 차이는 ?

strdup가 필요한 이유



`buf1`은 스택에 할당된 메모리이므로 `add`함수가 `return`되고 나면 소멸된다. 따라서 `buf1`에 저장된 문자열을 복제한 후 배열 `names[0]`에 복제된 배열의 주소를 저장해야 한다. 복제된 배열은 `strdup`함수 내에서 `malloc`으로 (heap에) 할당된 메모리이므로 `add`함수가 종료된 후에도 소멸하지 않는다.

C 언어에서 메모리 관리

● 전역변수 (global variable)

- 함수의 외부에 선언된 변수들
- 프로그램이 시작될 때 메모리가 할당되며 프로그램이 종료될 때 까지 유지된다
- Data section이라고 부르는 메모리 영역에 위치한다.

● 지역변수 (local variable)

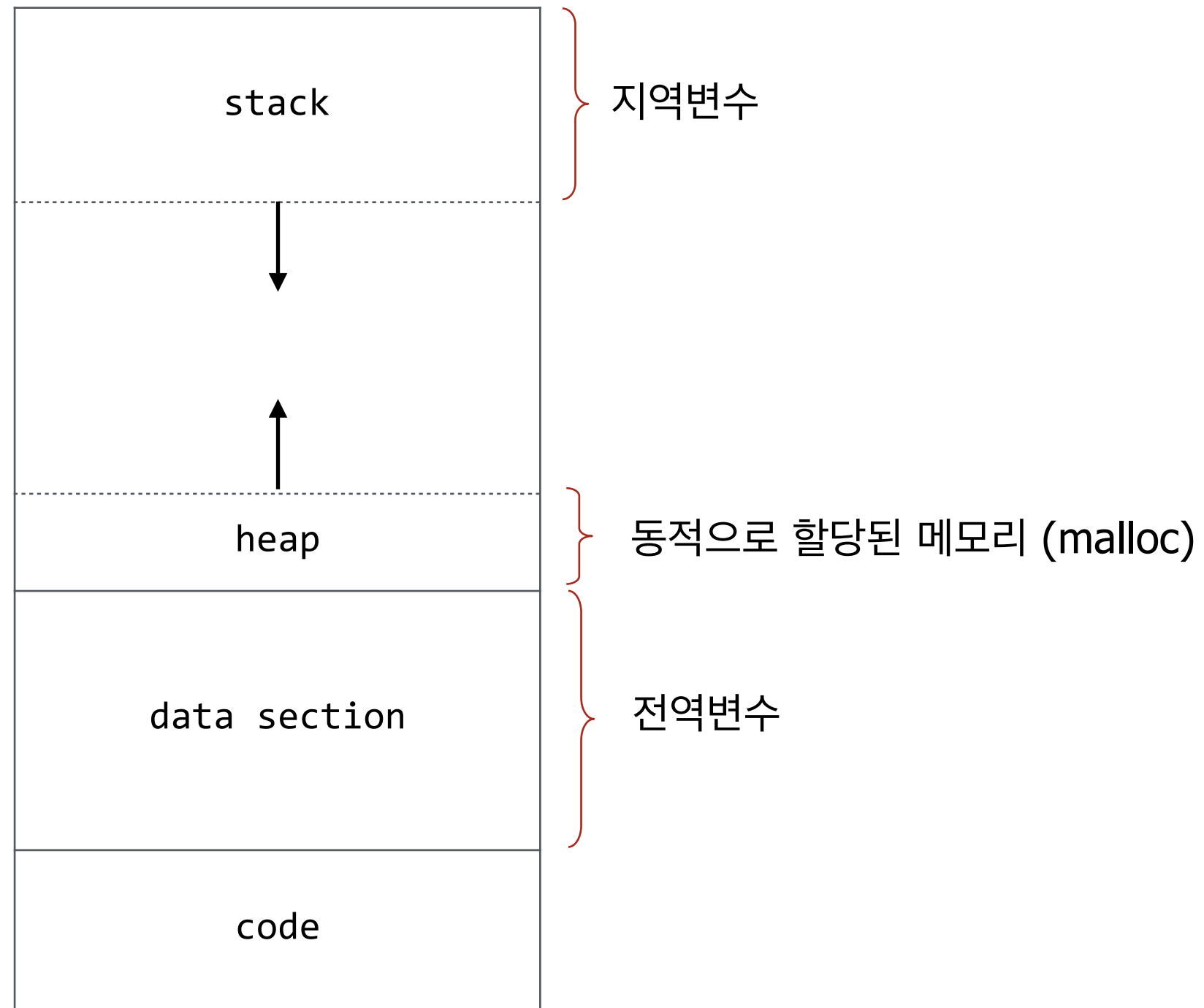
- 함수의 내부에 선언된 변수들
- 자신이 속한 함수가 호출될 때 메모리가 할당되며 함수가 return될 때 소멸된다.
- 스택(stack)이라고 부르는 영역에 위치한다.

C 언어에서 메모리 관리

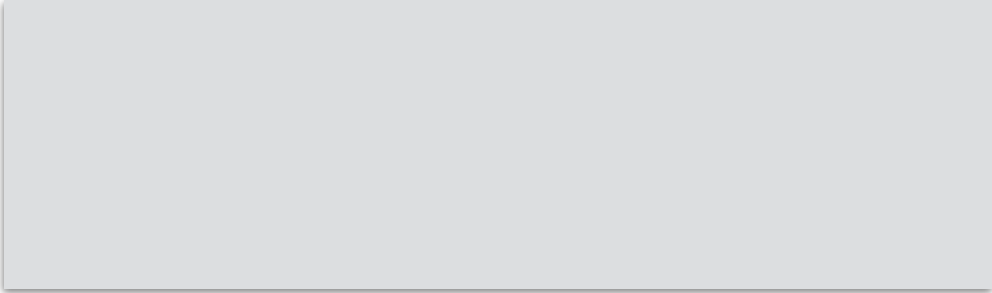
● 동적 메모리 할당 (dynamic memory allocation)

- 아무때나 malloc등의 함수를 호출하여 필요한 크기의 메모리를 할당할 수 있다. 이것을 동적 메모리 할당이라고 부른다.
- 동적으로 할당된 메모리는 힙(heap)이라고 부르는 영역에 위치한다.
- 동적으로 할당된 메모리는 명시적으로 free()함수를 호출하여 반환하지 않는 한 계속 유지된다.

C 언어에서 메모리 레이아웃



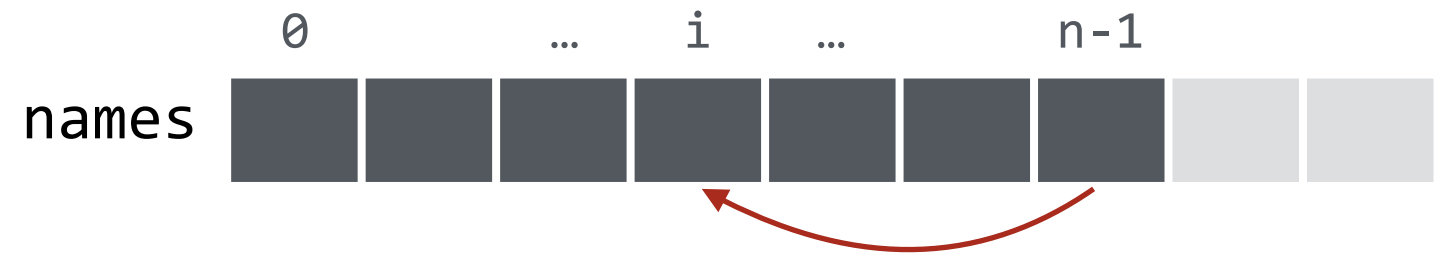
phonebook01.c

```
void find() {  
    char buf[BUFFER_SIZE];  
    scanf("%s", buf);  
  
    int i;  
    for (i=0; i<n; i++) {  
          
    }  
    printf("No person named '%s' exists.\n", buf);  
}
```

phonebook01.c

```
void status() {  
    int i;  
    for (i=0; i<n; i++)  
        printf("%s %s\n", names[i], numbers[i]);  
    printf("Total %d persons.\n", n);  
}
```

phonebook01.c



```
void remove() {  
    char buf[BUFFER_SIZE];  
    scanf("%s", buf);
```

```
    int i;  
    for (i=0; i<n; i++) {
```

맨 마지막 사람을 삭제된 자리로 옮긴다.

```
    }  
    printf("No person named '%s' exists.\n", buf);  
}
```

전화번호부.v2

파일로 저장하고 로드하기, 알파벳 순으로 정렬

실행 예

파일로부터 데이터를 읽어온다.

항상 알파벳 순으로 정렬된 상태를 유지한다.

```
$ read directory.txt
$ status
David 0517778888
Henry 0243737788
John 0103452374
Sean 34527354
Total 4 persons.
$ add Anderson 0103245257
Anderson was added successfully.
$ delete David
David was deleted successfully.
$ status
Anderson 0103245257
Henry 0243737788
John 0103452374
Sean 34527354
Total 4 persons.
$ save as directory.txt
$ exit
```

directory.txt (before)

```
David 0517778888
Henry 0243737788
John 0103452374
Sean 34527354
```

directory.txt (after)

```
Anderson 0103245257
Henry 0243737788
John 0103452374
Sean 34527354
```

파일에 데이터를 저장한다.

phonebook02.c

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

#define CAPACITY 100
#define BUFFER_SIZE 20

char * names[CAPACITY];
char * numbers[CAPACITY];
int n = 0;

/* names */
/* phone numbers */
/* number of people in phone directory */

void add();
void find();
void status();
void remove();
void load();
void save();
```

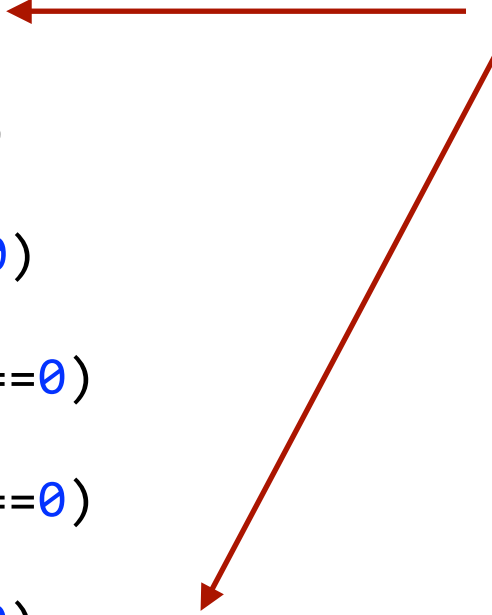
자료구조는 **phonebook01.c**와 완전히 동일하다.

phonebook02.c

```
int main() {
    char buffer[BUFFER_SIZE];

    while (1) {
        printf("$ ");
        scanf("%s", buffer);
        if (strcmp(buffer, "read")==0)
            load();
        else if (strcmp(buffer, "add")==0)
            add();
        else if (strcmp(buffer, "find")==0)
            find();
        else if (strcmp(buffer, "status")==0)
            status();
        else if (strcmp(buffer, "delete")==0)
            remove();
        else if (strcmp(buffer, "save")==0)
            save();
        else if (strcmp(buffer, "exit")==0)
            break;
    }
    return 0;
}
```

read와 save명령을 추가했고
각각을 처리하는 함수 load()와 save()를
추가하였다.



phonebook02.c

```
void load() {  
    char fileName[BUFFER_SIZE];  
    char buf1[BUFFER_SIZE];  
    char buf2[BUFFER_SIZE];
```

```
    scanf("%s", fileName);
```

← 파일 이름을 입력 받는다.

```
    FILE *fp = fopen(fileName, "r");  
    if (fp==NULL) {  
        printf("Open failed.\n");  
        return;  
    }
```

← 파일에 접근하기 위해서는 먼저
파일을 열어야(open)해야 한다.

← 파일의 끝에 도달할 때 까지 반복해서
이름과 전화번호를 읽어서 배열에 저장한다.

```
    fclose(fp);
```

← 볼일이 끝난 파일은 반드시 닫아 주어야 한다.

```
}
```

phonebook02.c

```
void save() {  
    int i;  
    char fileName[BUFFER_SIZE];  
    char tmp[BUFFER_SIZE];  
  
    scanf("%s", tmp);          // which is "as", discarded  
    scanf("%s", fileName);
```

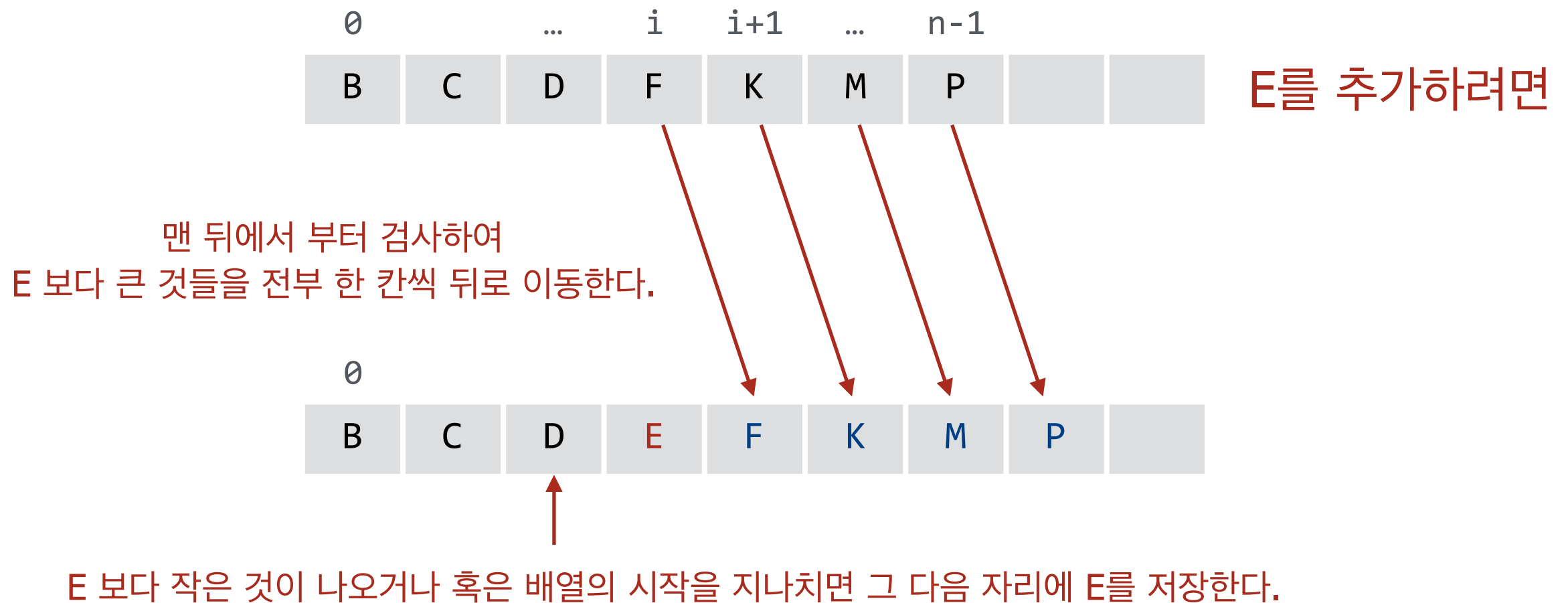
```
    FILE *fp = fopen(fileName, "w");  
    if (fp==NULL) {  
        printf("Open failed.\n");  
        return;  
    }
```

← 파일에 쓸 때는 모드를 "w"로 하고 열어야 한다.

```
    for (i=0; i<n; i++) {  
        fprintf(fp, "%s %s\n", names[i], numbers[i]);  
    }  
  
    fclose(fp);  
}
```

데이터를 정렬된 상태로 유지하려면

- bubblesort 등의 정렬(sorting) 알고리즘을 사용하는 방법
 - 새로운 데이터가 계속적으로 추가되는 우리의 상황에서는 부적절
- 새로운 데이터가 추가될 때 마다 제자리를 찾아서 삽입하는 방법



phonebook02.c

```
void add() {  
    char buf1[BUFFER_SIZE], buf2[BUFFER_SIZE];  
    scanf("%s", buf1);  
    scanf("%s", buf2);
```

사전식 순서로 나보다 큰 항목들은
모두 한 칸씩 뒤로 이동시키고,
처음으로 나보다 작은 항목이 나오면
그것 바로 뒤에 삽입한다.

```
        n++;  
        printf("%s was added successfully.\n", buf1);  
    }
```

phonebook02.c

```
void remove() {
    char buf[BUFFER_SIZE];
    scanf("%s", buf);

    int index = search(buf); /* returns -1 if not exists */
    if (index == -1) {
        printf("No person named '%s' exists.\n", buf);
        return;
    }

    n--;
    printf("'%' was deleted successfully. \n", buf);
}
```


phonebook02.c

```
void find() {  
    char buf[BUFFER_SIZE];  
    scanf("%s", buf);  
    int index = search(buf);  
    if (index == -1)  
        printf("No person named '%s' exists.\n", buf);  
    else  
        printf("%s\n", numbers[index]);  
}
```

phonebook02.c

```
int search(char *name) {
    int i;
    for (i=0; i<n; i++) {
        if (strcmp(name, names[i])==0) {
            return i;
        }
    }
    return -1;
}

void status() {
    int i;
    for (i=0; i<n; i++)
        printf("%s %s\n", names[i], numbers[i]);
    printf("Total %d persons.\n", n);
}
```

전화번호부.v3

배열 재할당, 라인 단위 입력과 문자열 **tokenizing**

실행 예

잘못된 명령어에 대해서 적절히 반응한다.

```
$ read
File name required
$ read directory.txt
$ status
David 0517778888
Henry 0243737788
John 0103452374
Sean 34527354
Total 4 persons.
$ add Anderson
Invalid arguments.
$ add Anderson 0103245257
Anderson was added successfully.
$ save directory.txt
Invalid command format.
$ save as directory.txt
$ exit
```

저장된 사람의 수가
배열의 용량을 초과할 경우
동적 메모리 할당으로
배열의 크기를 키운다.

phonebook03.c

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

#define INIT_CAPACITY 3    /* 배열 재할당을 테스트하기 위해서 일부러 아주 작은 값으로 */
#define BUFFER_SIZE 50

char ** names;
char ** numbers;          ← char * 타입의 배열의 이름이므로 char ** 타입의 변수이다.

int capacity = INIT_CAPACITY; /* size of arrays */
int n = 0;                  /* number of people in phone directory */

/* function prototypes here */

char delim[] = " ";

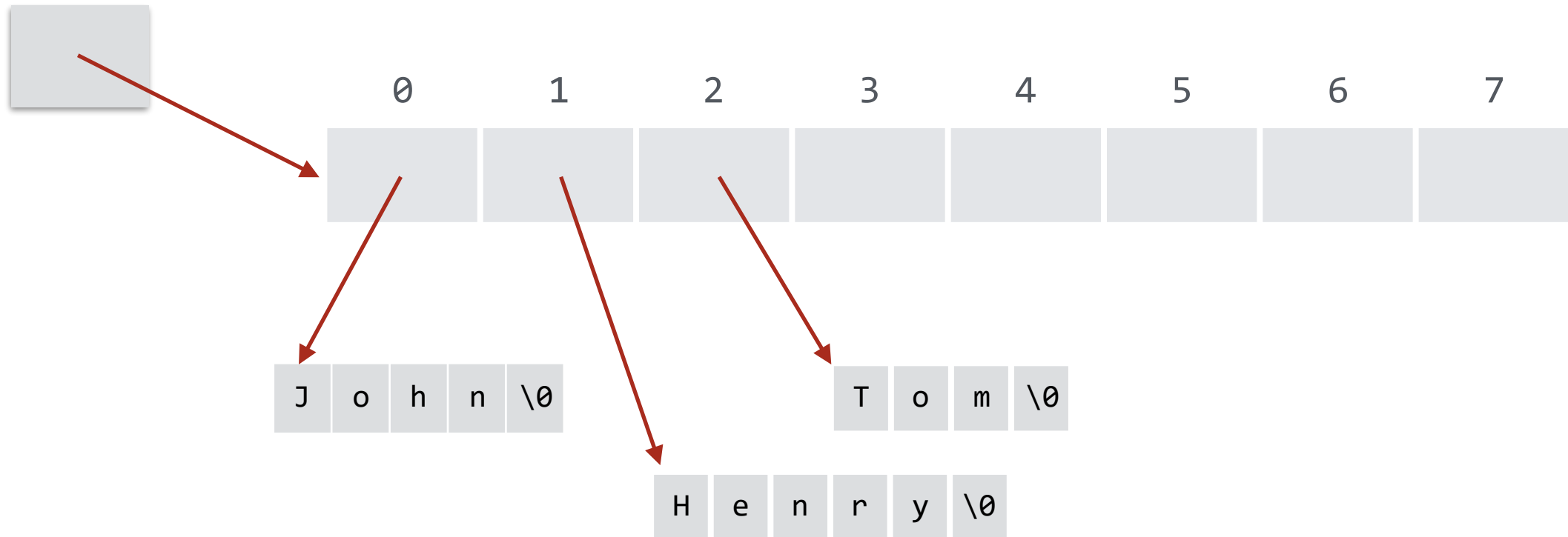
int main() {

    init_directory();       ← 이 함수에서 배열 names와 numbers를 생성한다.
    process_command();      ← 사용자의 명령을 받아 처리하는 부분을 별개의 함수로 만들었다.

    return 0;
}
```

phonebook03.c

names



```
void init_directory() {
```

```
}
```

할당할 메모리의 byte수를 지정한다.
직접 숫자로 지정하는 것 보다 이렇게
sizeof 연산자를 사용하는 것이 바람직하다.

phonebook03.c

배열 str의 크기이다.
즉 limit보다 더 긴 line의 경우에는 뒷부분이 잘린다.

```
int read_line(char str[], int limit)
{
```

```
    int ch, i = 0;
```

```
    while ((ch = getchar()) != '\n') ← 줄바꿈 문자가 나올 때까지 읽는다.
```

```
        if (i < limit-1) ← 배열의 용량을 초과하지 않을 때만 저장한다.
```

```
            str[i++] = ch;
```

```
    str[i] = '\0'; ← 마지막에 null character ('\0')를 추가해준다.
```

```
    return i;
```

```
}
```

실제로 읽은 문자수를 반환한다.

**line단위의 입력은
gets, fgets, getline 등의 함수들을 이용하여
할 수도 있다.**

```
int read_line(char str[], int limit)
{
    int ch, i = 0;

    while (i < limit-1 && (ch = getchar()) != '\n')
        str[i++] = ch;

    str[i] = '\0';

    return i;
}
```

앞 페이지의 코드와 유사해보이지만
이렇게 하면 문제가 있다. 어떤 문제일까?

문자열 tokenizing

- 구분 문자(delimiter)를 이용하여 하나의 긴 문자열을 작은 문자열들로 자르는 일을 문자열 tokenizing이라고 부른다. 잘라진 작은 문자열들을 보통 token이라고 부른다.
- C 언어에서는 주로 strtok 함수를 이용한다.

strtok을 이용한 문자열 찢르기(tokenizing)

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

```
int main(void) {
```

```
    char str[] = "now # is the time # to start preparing ### for the exam#";
    char delim[] = "#";
    char *token;
```

```
    token = strtok( str, delim );
```

← 첫번째 호출

```
    while ( token != NULL ) {
        printf( "next token is: %s:%d\n", token, strlen(token));
        token = strtok( NULL, delim );
```

← 이어진 호출들

```
    return 0;
```

```
}
```

```
next token is: now :4
next token is:  is the time :13
next token is:  to start preparing :20
next token is:  for the exam:13
```

strtok을 이용한 문자열 찌르기(tokenizing)

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

int main(void) {
    char str[] = " study hard,    or  sleep.  ";
    char delim[] = " ";
    char *token;

    token = strtok( str, delim );

    while ( token != NULL ) {
        printf( "next token is: %s:%d\n", token, strlen(token));
        token = strtok( NULL, delim );
    }

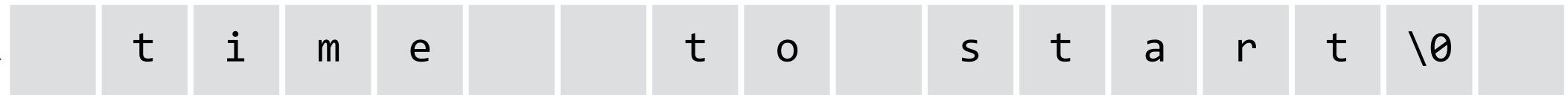
    return 0;
}
```

```
next token is: study:5
next token is: hard,:5
next token is: or:2
next token is: sleep.:6
```

How strtok works?

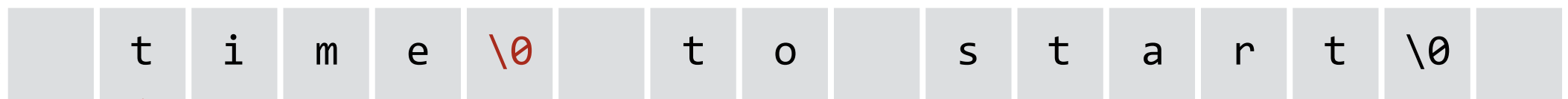
```
char delim[] = " ";
```

str



```
token = strtok( str, delim );
```

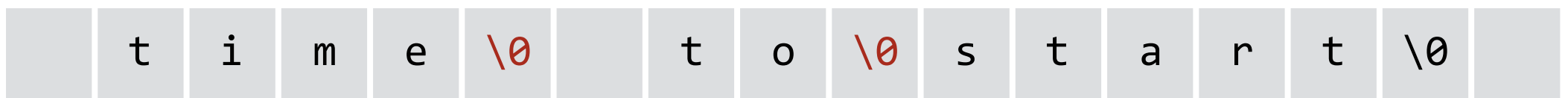
str



token

```
token = strtok( NULL, delim );
```

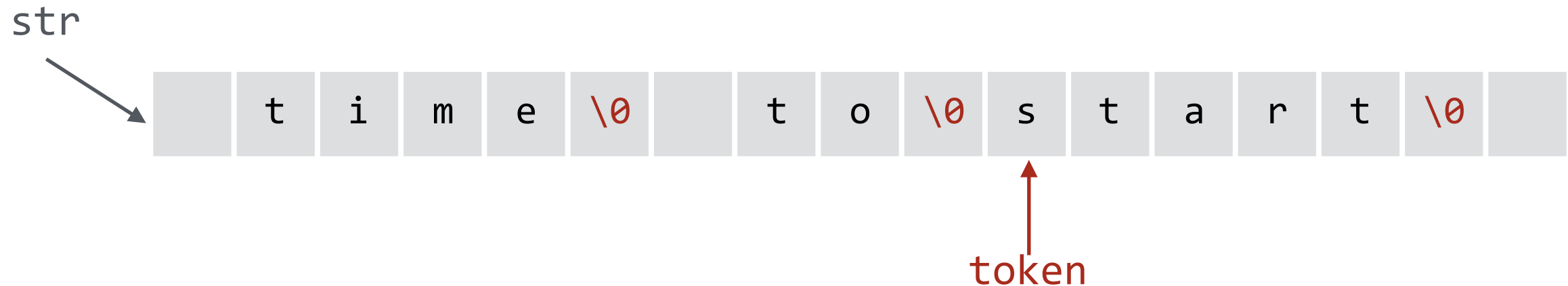
str



token

How strtok works?

```
token = strtok( NULL, delim );
```



- strtok은 원본 문자열을 **변화**시킨다 (‘\0’를 삽입한다.)
 - 따라서 만약 원본 문자열이 보존되어야 한다면 복사한 후 strtok을 해야한다.
- strtok은 새로운 배열을 **생성하지 않는다**.
 - 즉 strdup와는 다르다.

phonebook03.c

```
void process_command() {
    char command_line[BUFFER_SIZE];
    char *command, *argument1, *argument2;

    while (1) {
        printf("$ ");

        if (read_line(command_line, BUFFER_SIZE) <= 0)
            continue;

        [redacted]

        if (strcmp(command, "read") == 0) {
            [redacted]

            if (argument1 == NULL) {
                printf("File name required.\n");
                continue;
            }

            [redacted]
        }
    }
}
```

← 한 라인을 통채로 읽어오기 위한 버퍼

← 명령줄을 통채로 읽는다.

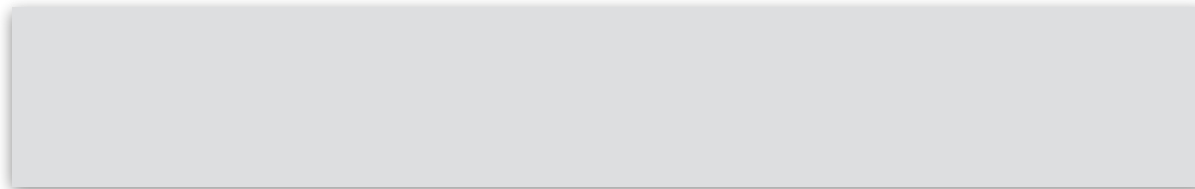
← 첫 번째 토큰은 명령어이다.

← read명령에서 두번째 토큰은 파일명이다.

← 파일명을 인자로 주면서 load를 호출한다.

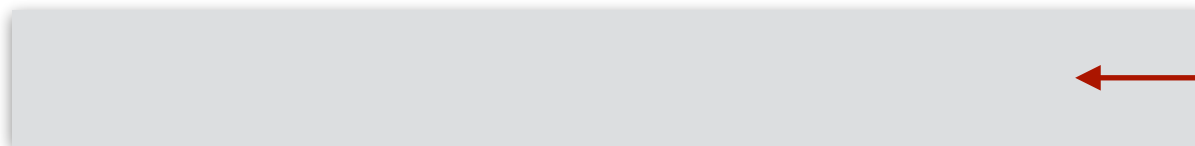
phonebook03.c

```
else if (strcmp(command, "add") == 0) {
```



명령어에 이어지는 2개의 토큰은
각각 이름과 전화번호이다.

```
    if (argument1 == NULL || argument2 == NULL) {  
        printf("Invalid arguments.\n");  
        continue;  
    }
```



이름과 전화번호를 인자로 주면서 add를 호출한다.

```
    printf("%s was added successfully.\n", argument1);  
}
```

phonebook03.c

```
else if (strcmp(command, "find") == 0) {
    argument1 = strtok(NULL, delim);
    if (argument1 == NULL) {
        printf("Invalid arguments.\n");
        continue;
    }
    find(argument1);
}
else if (strcmp(command, "status")==0)
    status();
else if (strcmp(command, "delete")==0) {
    argument1 = strtok(NULL, delim);
    if (argument1 == NULL) {
        printf("Invalid arguments.\n");
        continue;
    }
    remove(argument1);
}
```


phonebook03.c

```
else if (strcmp(command, "save")==0) {  
  
    argument1 = strtok(NULL, delim);  
    argument2 = strtok(NULL, delim);  
  
    if (argument1 == NULL || strcmp("as", argument1) != 0  
        || argument2 == NULL) {  
        printf("Invalid command format.\n");  
        continue;  
    }  
  
    save(argument2);  
}  
else if (strcmp(command, "exit")==0)  
    break;  
}  
}
```

phonebook03.c

```
void load(char *fileName) {
    char buf1[BUFFER_SIZE];
    char buf2[BUFFER_SIZE];

    FILE *fp = fopen(fileName, "r");
    if (fp==NULL) {
        printf("Open failed.\n");
        return;
    }
    while ((fscanf(fp, "%s", buf1)!=EOF)) {
        fscanf(fp, "%s", buf2);
        add(buf1, buf2);
    }
    fclose(fp);
}
```

phonebook03.c

```
void save(char *fileName) {
    int i;
    FILE *fp = fopen(fileName, "w");
    if (fp==NULL) {
        printf("Open failed.\n");
        return;
    }

    for (i=0; i<n; i++) {
        fprintf(fp, "%s %s\n", names[i], numbers[i]);
    }
    fclose(fp);
}
```

phonebook03.c

```
void remove(char *name) {  
  
    int i = search(name); /* returns -1 if not exists */  
    if (i == -1) {  
        printf("No person named '%s' exists.\n", name);  
        return;  
    }  
  
    int j = i;  
    for (; j < n-1; j++) {  
        names[j] = names[j+1];  
        numbers[j] = numbers[j+1];  
    }  
    n--;  
    printf("'%'s' was deleted successfully. \n", name);  
}
```

phonebook03.c

```
void status() {
    int i;
    for (i=0; i<n; i++)
        printf("%s %s\n", names[i], numbers[i]);
    printf("Total %d persons.\n", n);
}

void find(char *name) {
    int index = search(name);
    if (index==-1)
        printf("No person named '%s' exists.\n", name);
    else
        printf("%s\n", numbers[index]);
}

int search(char *name) {
    int i;
    for (i=0; i<n; i++) {
        if (strcmp(name, names[i])==0) {
            return i;
        }
    }
    return -1;
}
```

phonebook03.c

```
void add(char * name, char * number) {
```

배열이 꽉찬 경우
재할당한다.

```
    int i=n-1;
    while (i>=0 && strcmp(names[i], name) > 0) {
        names[i+1] = names[i];
        numbers[i+1] = numbers[i];
        i--;
    }
```

strdup이 반드시 필요한
이유는 ?

```
    n++;
}
```

phonebook03.c

```
void reallocate()  
{  
    int i;
```

← 먼저 크기가 2배인
배열들을 할당한다.

← 원본 배열 names와 numbers의 값을 새로운
배열에 모두 복사한다.

← 원본 배열 names와 numbers는 더 이상 필요없다. 하지만 두 배열은
init_directory() 함수에서 동적메모리할당으로 만들어진 배열이므로 그냥 두
면 없어지지 않고 계속 존재한다. 이런 메모리를 garbage라고 부른다.
garbage는 free 함수를 이용하여 반환한다.

← names와 numbers가 새로운 배열을 가리키도록 한다 .
(배열의 이름은 포인터 변수이다.)

전화번호부.v4

더 많은 항목, 구조체(structure)

실행 예

이름이 하나 이상의 단어로 구성될 수 있다.

단어 사이에 여러 개의 공백이 있을 경우 한 칸의 공백으로 저장된다.

```
$ add Hong Gil-Dong
```

```
Phone: 01023456789
```

```
Email: ← 모든 항목을 입력할 필요는 없다.
```

```
Group: Friend
```

```
John was added successfully.
```

```
$ find Hong Gil-Dong
```

```
Hong Gil-Dong:
```

```
Phone: 23874628
```

```
Email:
```

```
Group: Friend
```

```
$ save as directory.txt
```

```
$ exit
```

물론 파일로 저장하고(save as) 읽는(read) 기능을 지원해야 한다.

각 사람에 대해서 이름, 전화번호,
이메일 주소, 그리고 그룹을
지정할 수 있다.
단 이름을 제외한 다른 항목들은
비워둘 수도 있다.

파일 형식

존재하지 않는 항목의 경우 하나의 공백문자로 표시한다.

'#' 문자를 필드들 간의 구분자로 사용한다.

모든 라인은 반드시 구분자로 끝난다. 이렇게 하는 이유는?

directory.txt

```
David K.#0517778888# #Friend#  
Hong Gil-Dong# #henry@gmail.com# #  
John Doe# # # #  
Sean#01067356574#sean@naver.com#colleague#
```

한 줄에 한 명씩 저장한다.

- 항상 같이 붙어다녀야 하는 데이터를 별개의 변수들에 분산해서 저장하는 것은 바람직하지 않다.
- 어떤 한 사람의 이름, 전화번호, 이메일 주소 등이 그런 예이다.
- C 언어에서는 이런 경우 구조체(structure)를 사용한다.

자료구조: 구조체


```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define CAPACITY 100
#define BUFFER_LENGTH 100
```

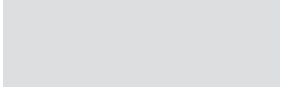
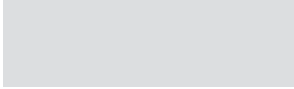
구조체 struct person을 정의하면서
동시에 그것을 Person으로 renaming했다.
이런 식으로 사용할 경우 structure tag인 person을 생략해도 된다.

Person 타입의 배열 directory를 선언한다.

```
int n = 0; /* number of people in phone directory */
```

파일로부터 라인 단위로 읽기

```
int read_line(  char str[], int n )
{
    int ch, i = 0;

    while ((ch =  ) != '\n' &&  )
        if (i < n)
            str[i++] = ch;

    str[i] = '\0';
    return i;
}
```

read_line을 수정하여 키보드만이 아니라 파일로부터도
읽을 수 있도록 하였다.

phonebook04.c

```
int main() {
    char command_line[BUFFER_LENGTH];
    char *command, *argument;
    char name_str[BUFFER_LENGTH];

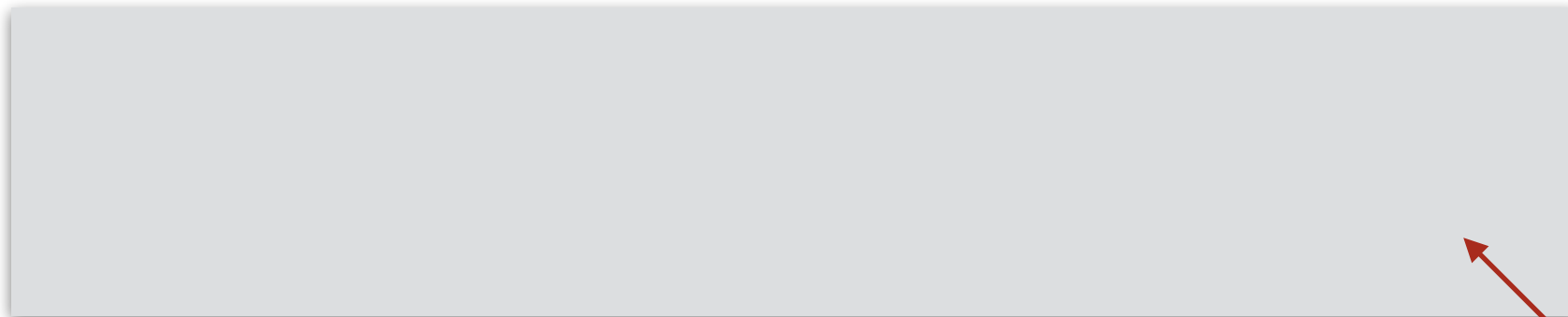
    while (1) {
        printf("$ ");
        if (read_line(stdin, command_line, BUFFER_LENGTH) <= 0)
            continue;

        command = strtok(command_line, " ");

        if (strcmp(command, "read") == 0) {
            argument = strtok(NULL, " ");
            if (argument == NULL) {
                printf("Invalid arguments.\n");
                continue;
            }
            load(argument);
        }
    }
}
```

phonebook04.c

```
else if (strcmp(command, "add") == 0) {
```



```
    handle_add(name_str);
```

```
}
```

```
else if (strcmp(command, "find") == 0) {
```



```
    find(name_str);
```

```
}
```

compose_name은 나머지 토큰들을
merge하여 이름을 구성한다.

phonebook04.c

```
else if (strcmp(command, "status")==0) {
    status();
}
else if (strcmp(command, "delete")==0) {
    if (compose_name(name_str, BUFFER_LENGTH) <= 0) {
        printf("Invalid arguments.\n");
        continue;
    }
    remove(name_str);
}
```


phonebook04.c

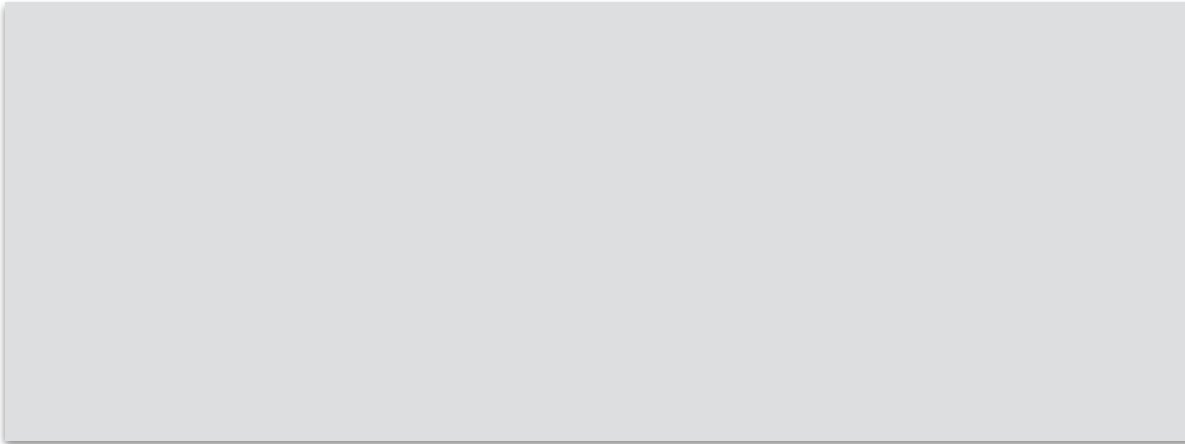
```
else if (strcmp(command, "save")==0) {
    argument = strtok(NULL, " ");
    if (strcmp(argument, "as") != 0) {
        printf("Invalid arguments.\n");
        continue;
    }
    argument = strtok(NULL, " ");
    if (argument == NULL) {
        printf("Invalid arguments.\n");
        continue;
    }
    save(argument);
}
else if (strcmp(command, "exit")==0)
    break;
}
return 0;
}
```

phonebook04.c

```
int compose_name(char str[], int limit) {
    char * ptr;
    int length = 0;

    ptr = strtok(NULL, " ");
    if (ptr == NULL)
        return 0;

    strcpy(str, ptr);
    length += strlen(ptr);

    while ((ptr = strtok(NULL, " ")) != NULL) {
        
    }
    return length;
}
```

**command_line의 남아있는 토큰들을
모두 합쳐 이름을 나타내는 문자열을 구성한다.
토큰과 토큰 사이에 하나의 공백문자를 삽입한다.**

phonebook04.c

```
void load(char *fileName) {
    char buffer[BUFFER_LENGTH];
    char * name, *number, *email, *group;

    FILE *fp = fopen(fileName, "r");
    if (fp==NULL) {
        printf("Open failed.\n");
        return;
    }

    while (1) {
        if (read_line(fp, buffer, BUFFER_LENGTH)<=0)
            break;
        name = strtok(buffer, "#");
        number = strtok(NULL, "#");
        email = strtok(NULL, "#");
        group = strtok(NULL, "#");
        add(name, number, email, group);
    }
    fclose(fp);
}
```

phonebook04.c

```
void save(char *fileName) {
    int i;
    FILE *fp = fopen(fileName, "w");
    if (fp==NULL) {
        printf("Open failed.\n");
        return;
    }

    for (i=0; i<n; i++) {
        fprintf(fp, "%s#", directory[i].name);
        fprintf(fp, "%s#", directory[i].number);
        fprintf(fp, "%s#", directory[i].email);
        fprintf(fp, "%s#\n", directory[i].group);
    }
    fclose(fp);
}
```

phonebook04.c

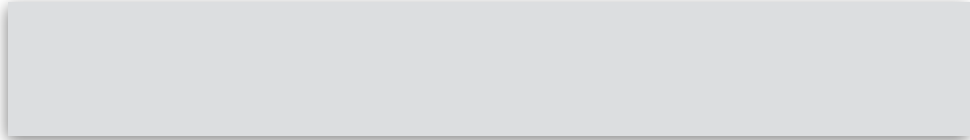
```
int search(char *name) {
    int i;
    for (i=0; i<n; i++) {
        if (strcmp(name, directory[i].name)==0) {
            return i;
        }
    }
    return -1;
}

void print_person(Person p)
{
    printf("%s:\n", p.name);
    printf("    Phone: %s\n", p.number);
    printf("    Email: %s\n", p.email);
    printf("    Group: %s\n", p.group);
}
```

phonebook04.c

```
void remove(char *name) {
    int i = search(name); /* returns -1 if not exists */
    if (i == -1) {
        printf("No person named '%s' exists.\n", name);
        return;
    }

    release_person(i);

    int j = i;
    for (; j < n-1; j++) {
        
    }
    n--;
    printf("'%'s' was deleted successfully. \n", name);
}
```

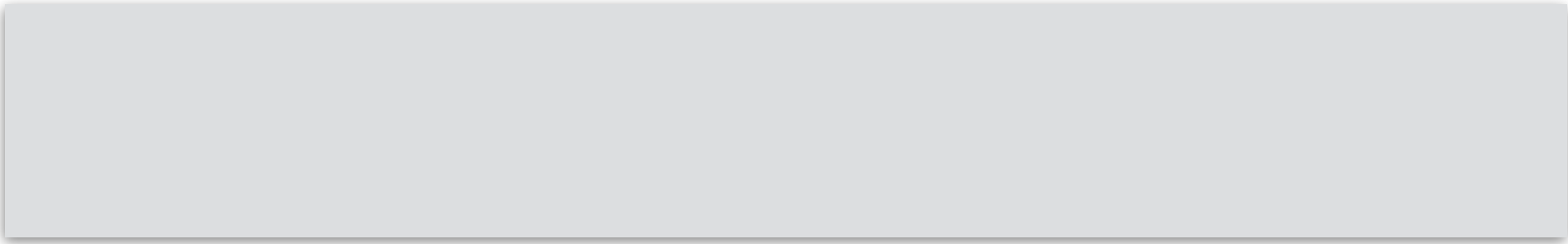
구조체 변수간의 치환연산이 지원되므로
멤버 항목들을 따로따로 치환할 필요가 없다.

phonebook04.c

```
void status() {
    int i;
    for (i=0; i<n; i++)
        print_person(directory[i]);
    printf("Total %d persons.\n", n);
}

void find(char *name) {
    int index = search(name);
    if (index==-1)
        printf("No person named '%s' exists.\n", name);
    else
        print_person(directory[index]);
}
```

phonebook04.c

```
void handle_add(char * name) {  
    char number[BUFFER_LENGTH], email[BUFFER_LENGTH], group[BUFFER_LENGTH];  
    char empty[] = " ";  
  
    printf("  Phone: ");  
    read_line(stdin, number, BUFFER_LENGTH);  
  
    printf("  Email: ");  
    read_line(stdin, email, BUFFER_LENGTH);  
  
    printf("  Group: ");  
    read_line(stdin, group, BUFFER_LENGTH);  
  
      
}
```

존재하지 않는 항목들을 하나의 공백문자로 구성된 문자열로 대체한다.

phonebook04.c

```
void add(char *name, char *number, char *email, char *group) {  
    int i=n-1;  
    while (i>=0 && strcmp(directory[i].name, name) > 0) {  
        directory[i+1] = directory[i];  
        i--;  
    }
```



```
        n++;  
    }
```



모든 항목들은 strdup로 복제하여 저장한다.

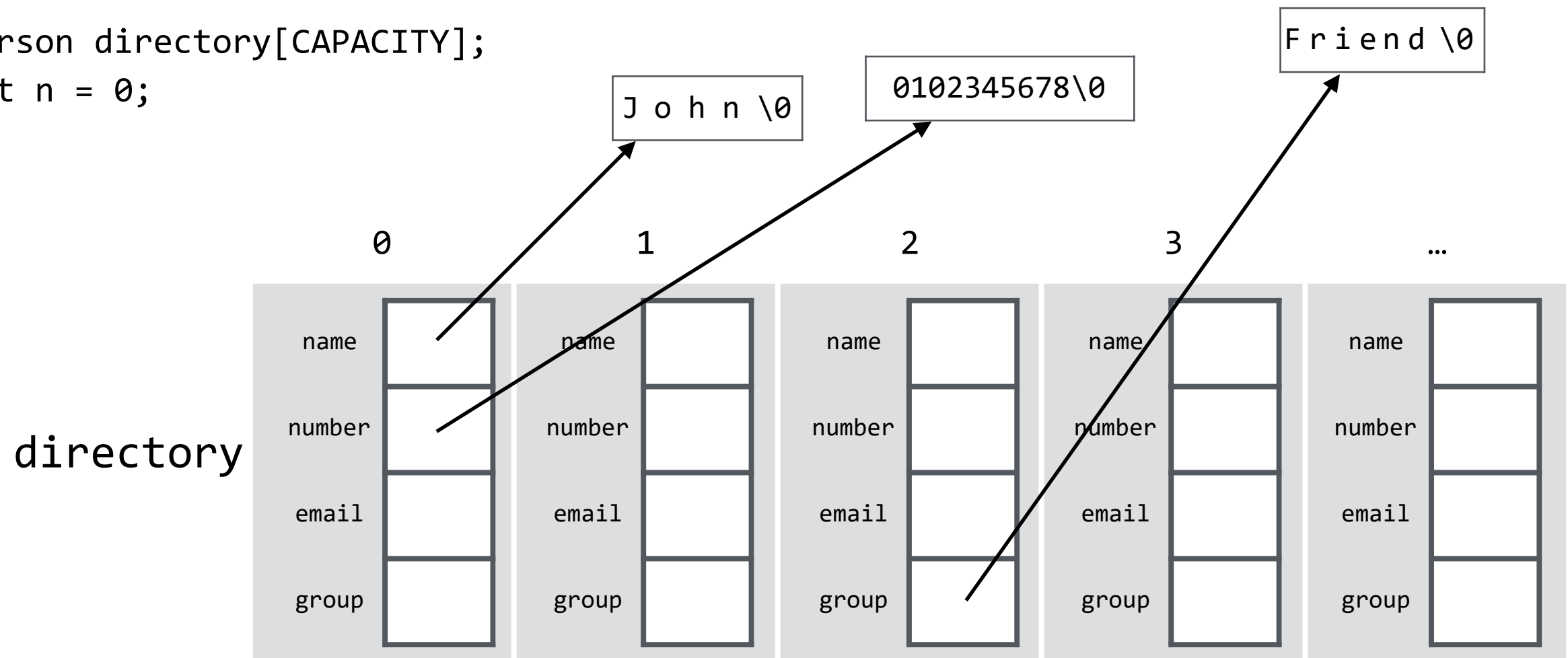
전화번호부.v5

구조체에 대한 포인터, 동적 메모리 할당

v4.0에서는

```
typedef struct person {  
    char *name;  
    char *number;  
    char *email;  
    char *group;  
} Person;
```

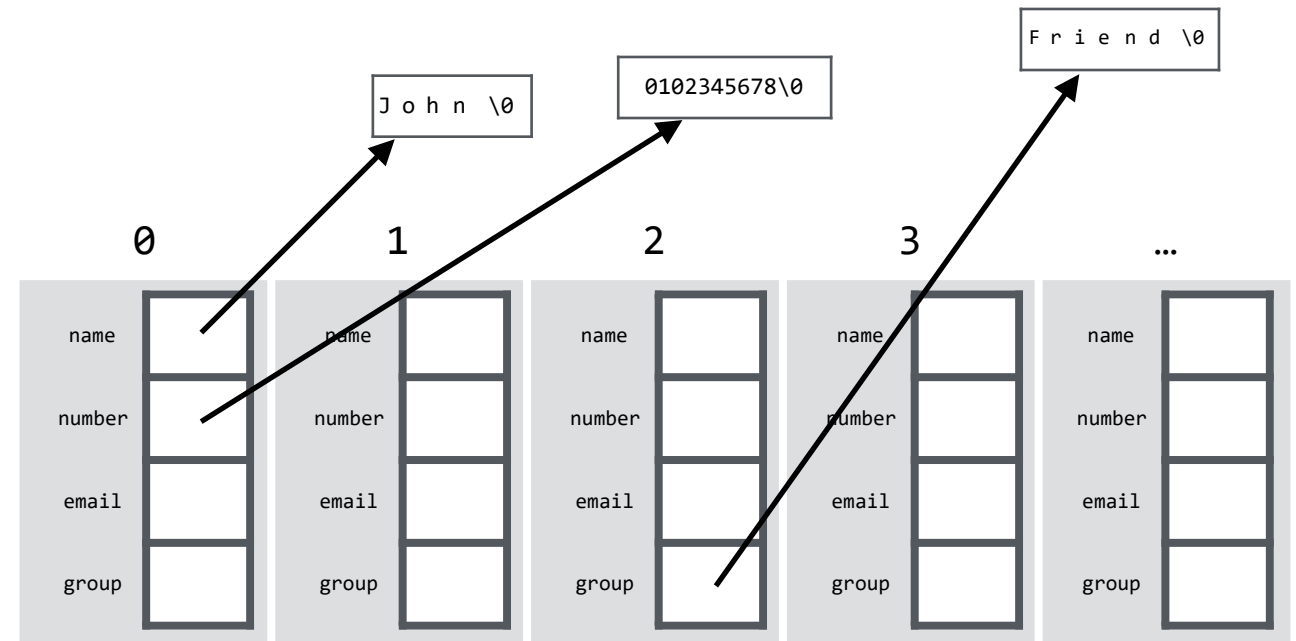
```
Person directory[CAPACITY];  
int n = 0;
```



가령 print_person() 함수에서

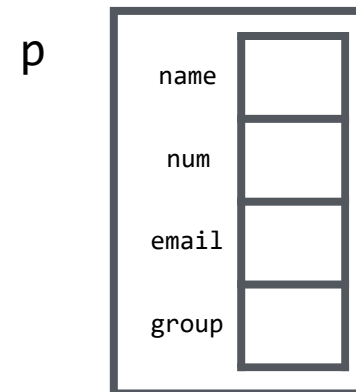
```
void status() {
    int i;
    for (i=0; i<n; i++)
        print_person(directory[i]);
    printf("Total %d persons.\n", n);
}
```

directory



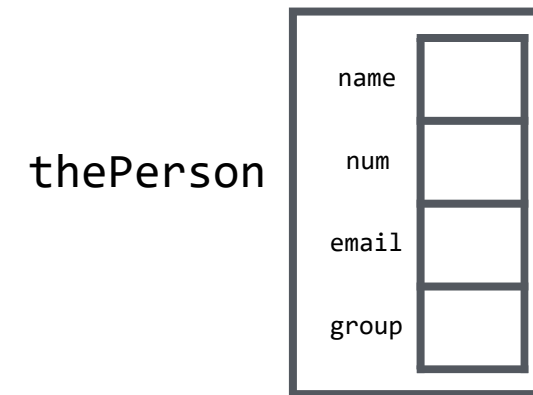
호출시에 모든 멤버들이 복사됨

```
void print_person(Person p)
{
    printf("%s:\n", p.name);
    printf("    Phone: %s\n", p.number);
    printf("    Email: %s\n", p.email);
    printf("    Group: %s\n", p.group);
}
```

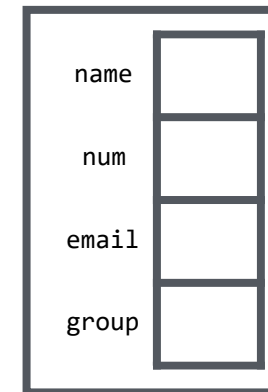


return 문의 경우

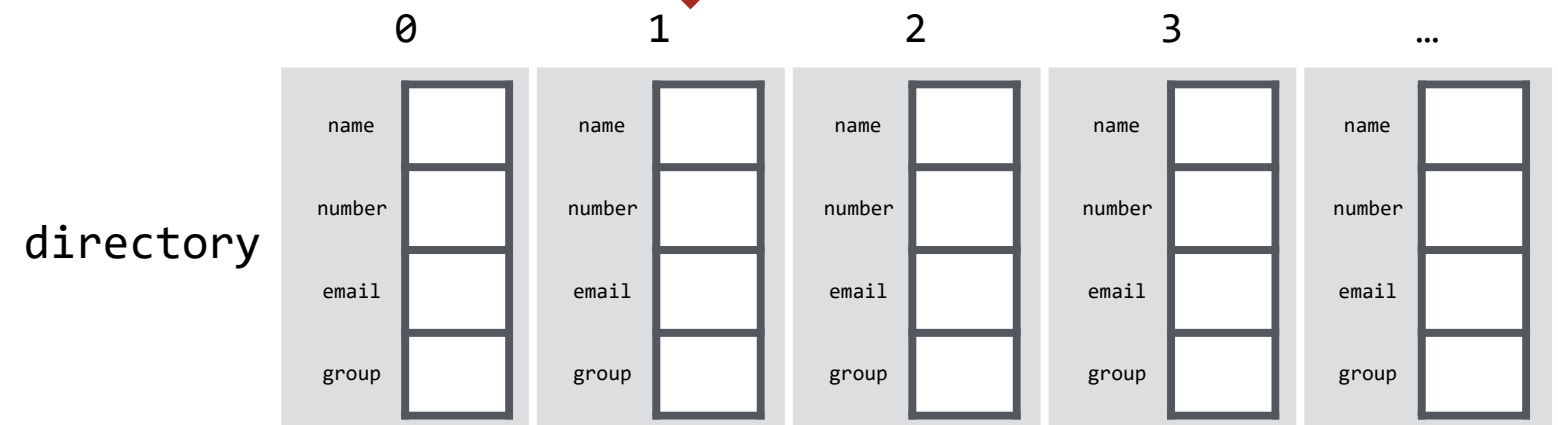
```
void some_function() {  
    ...  
    Person thePerson = get_person("John");  
    ...  
}
```



이름 없는 임시 객체

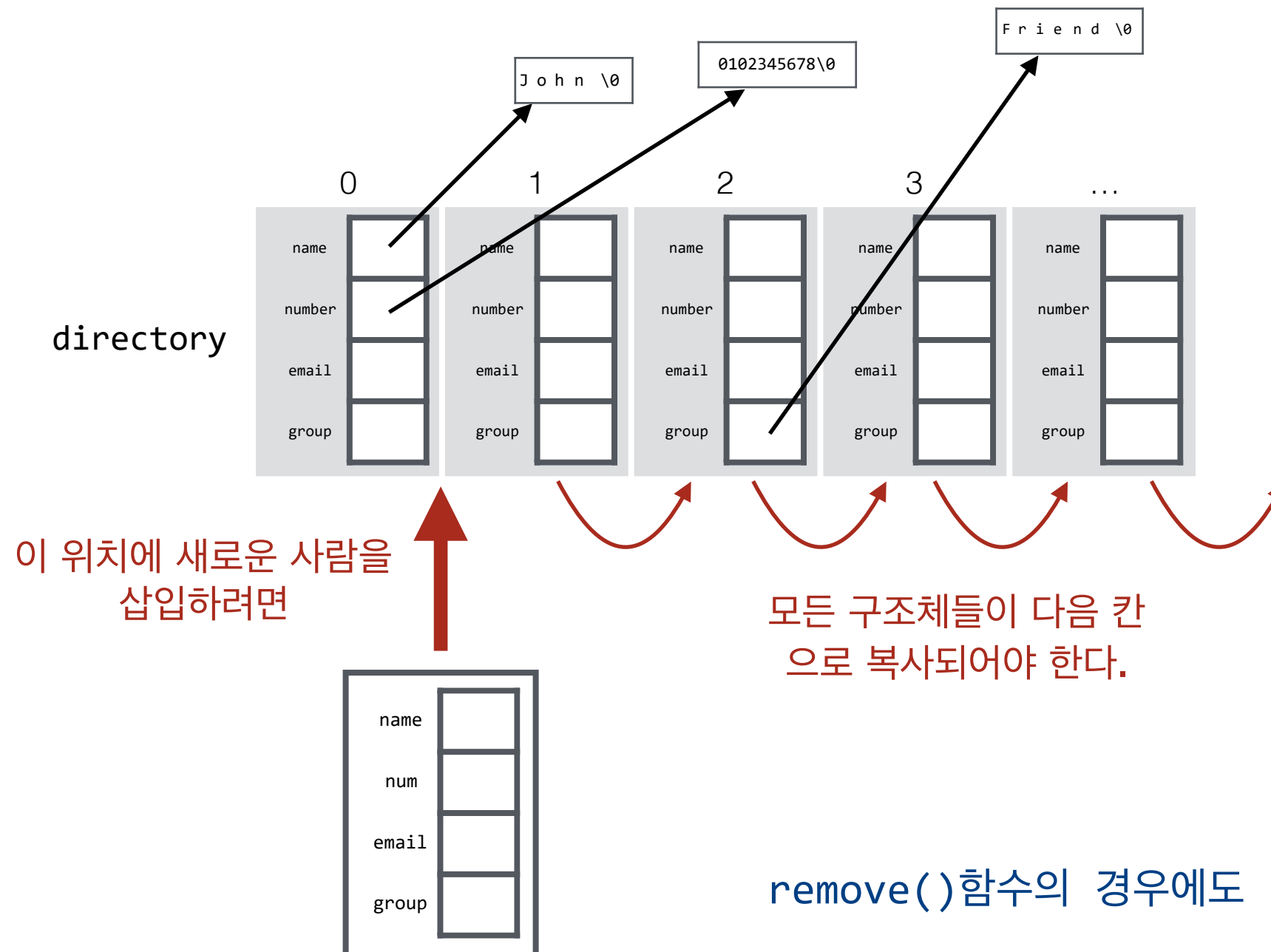


```
Person get_person(char *name)  
{  
    ...  
    return directory[i];  
}
```



add()함수에서

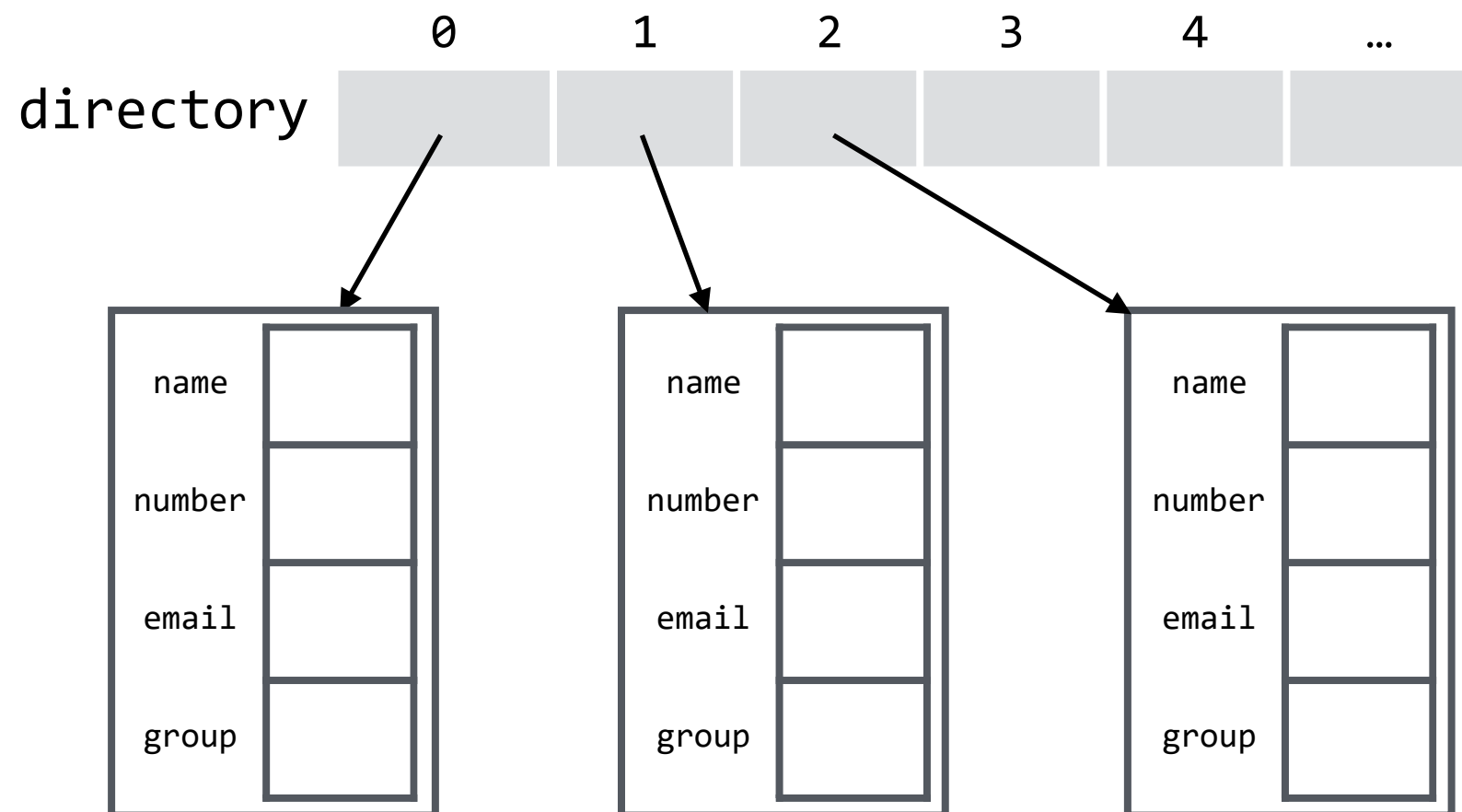
```
int i=n-1;
while (i>=0 && strcmp(directory[i].name, name) > 0) {
    directory[i+1] = directory[i];
    i--;
}
```



v5.0에서는

```
typedef struct person {  
    char *name;  
    char *number;  
    char *email;  
    char *group;  
} Person;
```

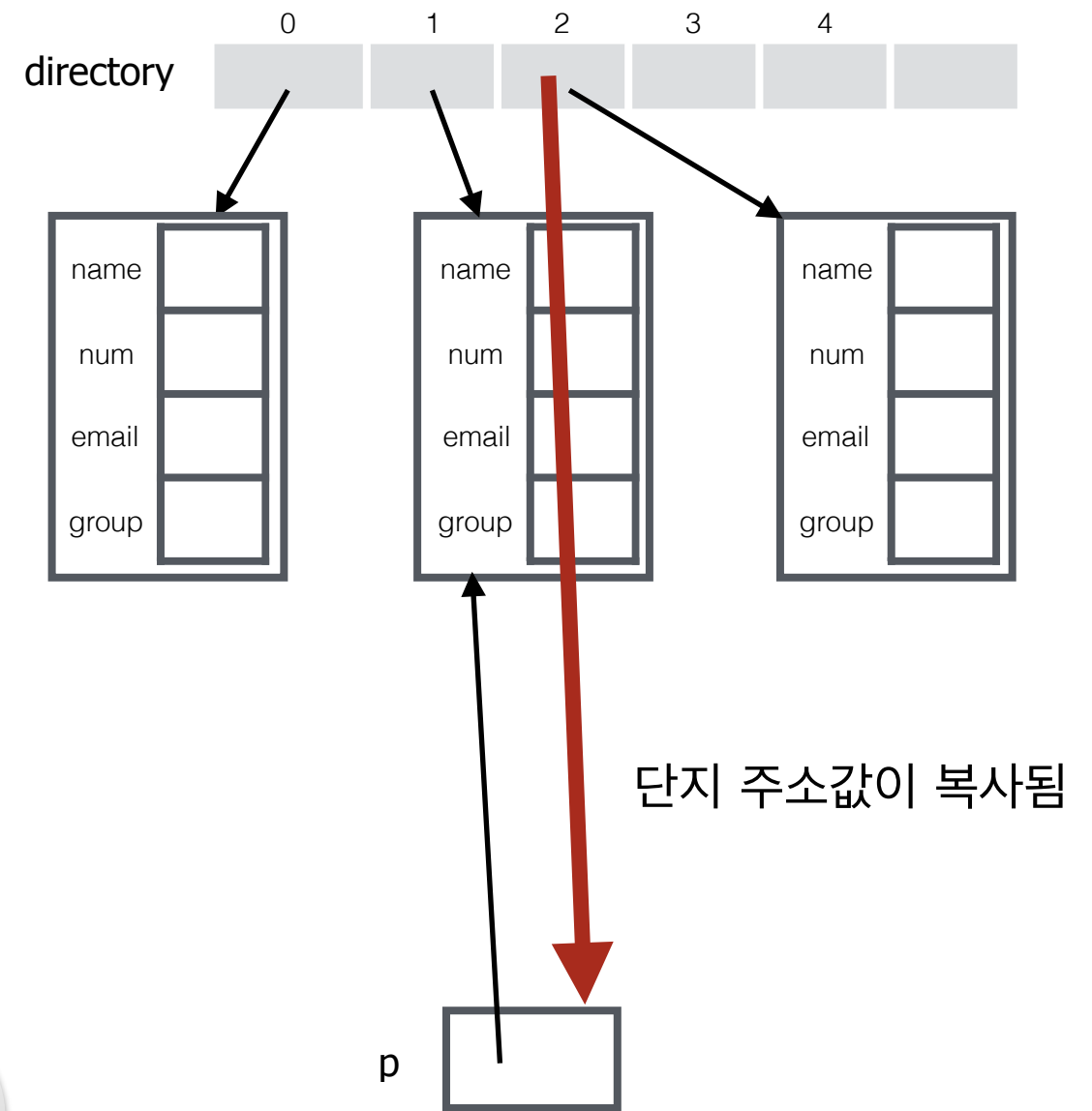
```
Person * directory[CAPACITY];  
int n = 0;
```



가령 print_person() 함수에서

```
void status() {  
    int i;  
    for (i=0; i<n; i++)  
        print_person(directory[i]);  
    printf("Total %d persons.\n", n);  
}
```

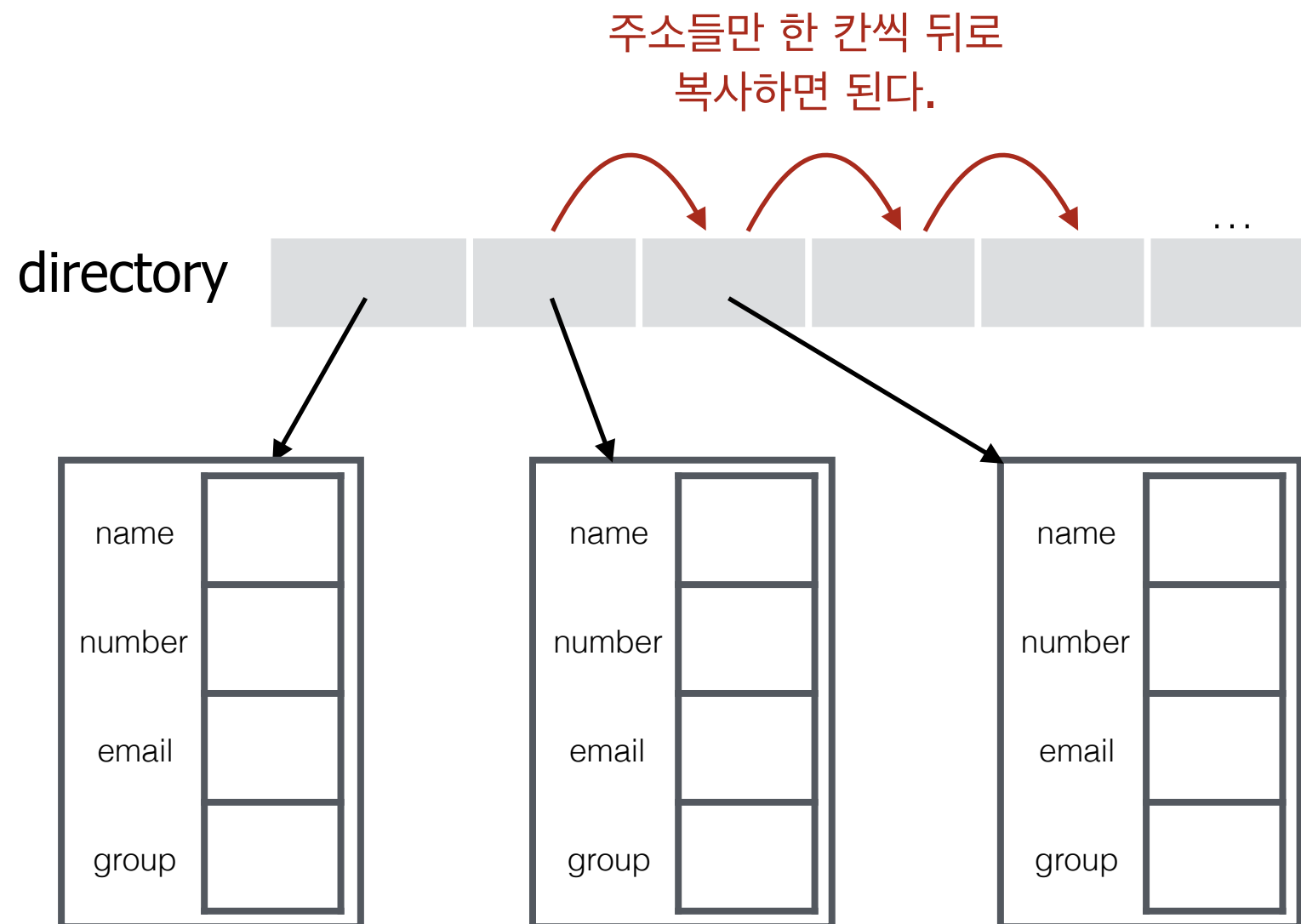
```
void print_person(Person *p)  
{  
    printf("%s:\n", (*p).name);  
    printf("    Phone: %s\n", (*p).number);  
    printf("    Email: %s\n", (*p).email);  
    printf("    Group: %s\n", (*p).group);  
}
```



p가 포인터이므로 이렇게 해야함

add() 함수에서는

```
int i=n-1;  
while (i>=0 && strcmp(directory[i]->name, name) > 0) {  
    directory[i+1] = directory[i];  
    i--;  
}
```



새로운 연산자 ->

실제 v5에 사용될 print_person 함수는 이것과 조금 달라야 한다.

```
void print_person(Person *p)
{
    printf("%s:\n", p->name);
    printf("    Phone: %s\n", p->number);
    printf("    Email: %s\n", p->email);
    printf("    Group: %s\n", p->group);
}
```

->연산자를 사용하면 표현이 간결해짐

자료구조와 init() 함수

```
#define INIT_CAPACITY 100
```

```
typedef struct {  
    char *name;  
    char *number;  
    char *email;  
    char *group;  
} Person;
```

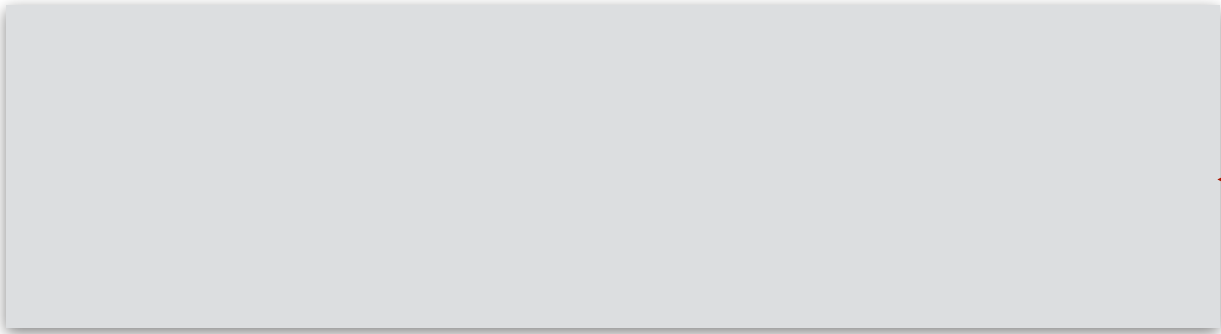
```
Person ** directory;  
int capacity;  
int n;
```

```
void init() {  
    directory = (Person **)malloc(INIT_CAPACITY*sizeof(Person *));  
    capacity = INIT_CAPACITY;  
    n = 0;  
}
```

```
void load(char *fileName) {  
    char buffer[BUFFER_LENGTH];  
    char * name, *number, *email, *group;  
    char *token;  
  
    FILE *fp = fopen(fileName, "r");  
    if (fp==NULL) {  
        printf("Open failed.\n");  
        return;  
    }  
}
```

```
while (1) {  
    if (read_line(fp, buffer, BUFFER_LENGTH) <= 0)  
        break;  
    name = strtok(buffer, "#");  
    token = strtok(NULL, "#");  
    if (strcmp(token, " ") == 0)  
        number = NULL;  
    else  
        number = strdup(token);  
    token = strtok(NULL, "#");  
    if (strcmp(token, " ") == 0)  
        email = NULL;  
    else  
        email = strdup(token);  
    token = strtok(NULL, "#");  
    if (strcmp(token, " ") == 0)  
        group = NULL;  
    else  
        group = strdup(token);  
    add(strdup(name), number, email, group);  
}  
fclose(fp);  
}
```

add

```
void add(char *name, char *number, char *email, char *group) {  
    if (n>=capacity)  
        reallocate();  
  
    int i=n-1;  
    while (i>=0 && strcmp(directory[i]->name, name) > 0) {  
        directory[i+1] = directory[i];  
        i--;  
    }  
  
    directory[i+1] = (Person *)malloc(sizeof(Person));  
  
     ← strdup으로 복제하지 않고 저장한다.  
  
    n++;  
}
```

reallocate

```
void reallocate() {  
    capacity *= 2;  
    Person **tmp = (Person **)malloc(capacity*sizeof(Person *));  
    for (int i=0; i<n; i++)  
        tmp[i] = directory[i];  
    free(directory);  
    directory = tmp;  
}
```

remove

```
void remove(char *name) {  
    int i = search(name); /* returns -1 if not exists */  
    if (i == -1) {  
        printf("No person named '%s' exists.\n", name);  
        return;  
    }
```

```
        printf("'%s' was deleted successfully. \n", name);  
    }
```

```
void release_person(Person *p) {
```

```
}
```


- main함수는 처음에 init()을 호출해주는 것을 제외하면 v4와 동일함
- read_line, compose_name은 v4와 동일함

수정할 함수들

- 변경된 자료구조와 add 함수에 맞게 프로그램의 나머지 부분을 수정하라.
- save, search, print_person, status, find, handle_add
- v4와는 달리 존재하지 않는 항목들은 NULL로 저장되어 있다.