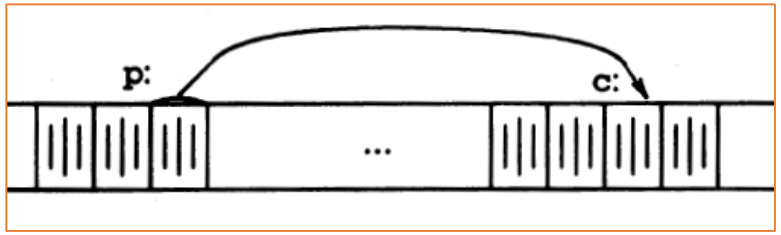
Pointers and Arrays

백 윤 철

Contents

Pointers and Addresses

- A pointer is a variable that contains the address of a variable.



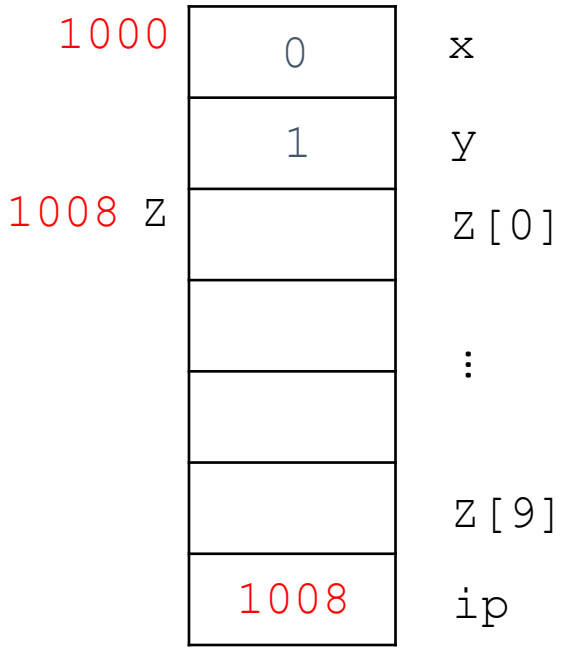
(If c is a char and p is a pointer that points to it.)

- The unary operator & gives the address of an object.

```
p = &c;
```

- The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.
- The unary operator * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to.

Pointers and Addresses



- how to declare a pointer and how to use & and *.

```
int x = 1, y = 2, z[10];
int *ip;          /* ip is a pointer to int */

ip = &x;          /* ip now points to x */
y = *ip;          /* y is now 1 */
*ip = 0;          /* x is now 0 */
ip = &z[0];       /* ip now points to z[0] */
```

- every pointer points to a specific data type.

(There is one exception: a "pointer to void" is used to hold any type of pointer but cannot be dereferenced itself.)

- `*ip += 1, ++*ip and (*ip)++` is possible, parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left

Pointers and Function arguments

- Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function

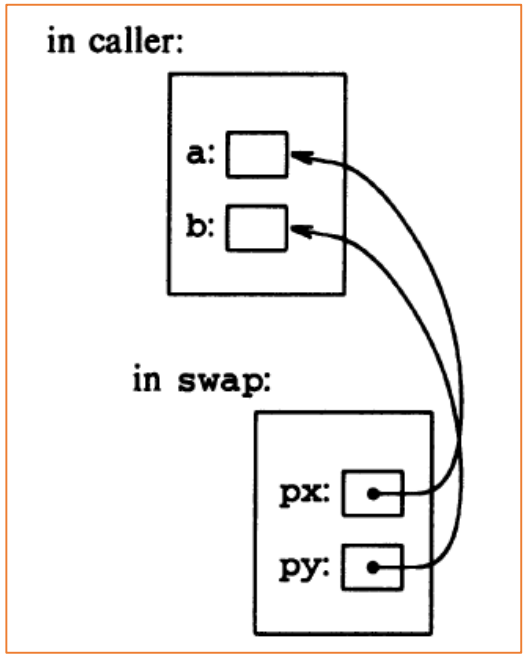
```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
swap(a, b);
```

- Pointer arguments enable a function to access and change objects in the function that called it.

```
void swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

```
swap(&a, &b);
```



Pointers and Function arguments

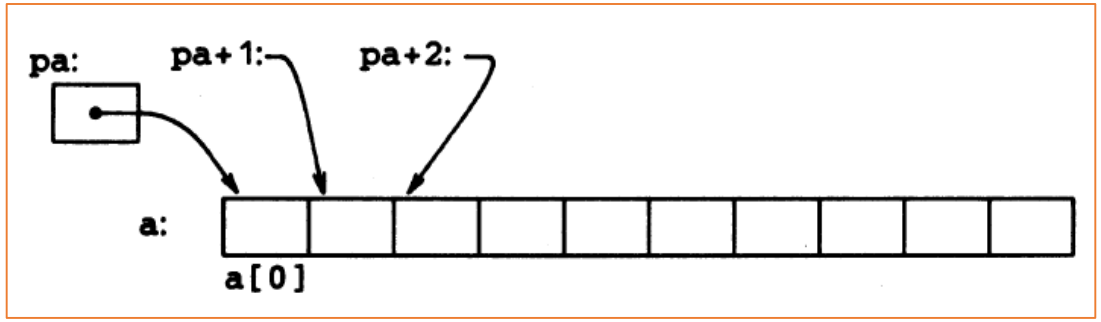
- `getint()` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function.
- This is the scheme used by scanf as well.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))   /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);    /* it's not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Pointers and Arrays

- In C, there is a strong relationship between pointers and arrays
- Any operation that can be achieved by array subscripting can also be done with pointers.

```
int a[10];
int *pa;
pa = &a[0]; // pa = a;
```



Pointers and Arrays

- `a[i]` is same to `*(a+i)`
- `&a[i]` and `a+i` are also identical.
- `pa[i]` is identical to `*(pa+i)` .
- In short, an array-and-index expression is equivalent to one written as a pointer and offset.
- If pa is pointer variable, `pa=a` and `pa++` are legal.
- But an array name is not a variable; constructions like `a=pa` and `a++` are illegal.

Pointers and Arrays

- `strlen()`

```
/* strlen:  return length of string s */
int strlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

  - all followings work

```
strlen("hello, world");  /* string constant */
strlen(array);           /* char array[100]; */
strlen(ptr);             /* char *ptr; */
```

- As formal parameters in a function definition, `char s[]` and `char *s` are equivalent; we prefer the latter because it says more explicitly that the parameter is a pointer.

Pointers and Arrays

- It is possible to pass part of an array to a function
  - passing a pointer to the beginning of the subarray.
  - `f(&a[2])` and `f(a+2)` both pass to the function `f` the address of the subarray that starts at `a[2]`.
- If one is sure that the elements exist, it is also possible to index backwards in an array
  - `p[-1], p[-2],` and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`.
- Of course, it is illegal to refer to objects which are not within the array bounds.

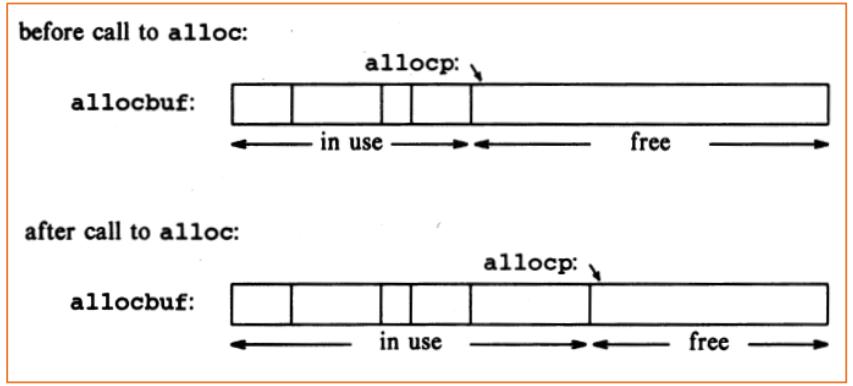
Address Arithmetic

- If `p` is a pointer to some element of an array, then `p++` increments p to point to the next element
- `p+=i` increments it to point `i` elements beyond where it currently does.
- Storage allocator
  - `alloc(n)` returns a pointer p to n-consecutive character positions, which can be used by the caller of `alloc` for storing characters.
  - `afree(p)` releases the storage so it can be re-used later

Address Arithmetic

- `alloc()`

```
#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE];  /* storage for alloc */
static char *allocp = allocbuf;   /* next free position */

char *alloc(int n)  /* return pointer to n characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* it fits */
        allocp += n;
        return allocp - n; /* old p */
    } else       /* not enough room */
        return 0;
}
```



Address Arithmetic

- afree()
  - Too simple but it can work like stack

```
void afree(char *p)  /* free storage pointed to by p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```

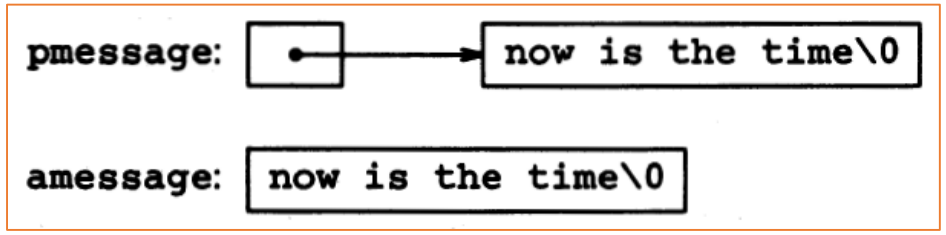
Address Arithmetic

- If `p` and `q` point to members of the same array, then comparison (e.g. `p < q`) works.
- a pointer and an integer may be added or subtracted
  - `p + n` means the address of the n-th object beyond the one p currently points to
  - `n` is scaled according to the size of the objects p points to
  - If an `int` is four bytes, for example, the `int` will be scaled by four
  - Pointer subtraction is also valid: if `p` and `q` point to elements of the same array, and `p<q`
- Illegal pointer operation
  - add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them
  - assign a pointer of one type to a pointer of another type without a cast.

Character Pointers and Functions

```
char amessage[] = "now is the time";    /* an array */
char *pmessage = "now is the time";     /* a pointer */
```

- amessage 배열
  - an array, just big enough to hold the sequence of characters and '\0' .
  - Individual characters within the array may be changed
  - amessage will always refer to the same storage.
- pmessage 포인터
  - a pointer, initialized to point to a string constant
  - the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



Character Pointers and Functions

- `strcpy(s,t)`, which copies the string t to the string s.

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

```
/* strcpy:  copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Character Pointers and Functions

- `strcpy(s,t)` more versions

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

```
/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Character Pointers and Functions

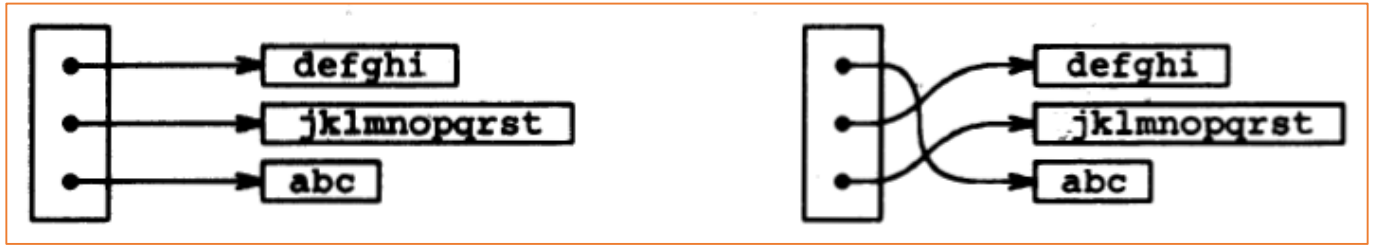
- `strcmp(s,t)`
  - Compares the character strings `s` and `t`
  - returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`.
  - The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Pointer Arrays; Pointers to Pointers

- Since pointers are variables themselves, they can be stored in arrays just as other variables can.
- writing a program that will sort a set of text lines into alphabetic order
  - lines of text of different lengths can't be compared or moved in a single operation.
  - each text line can be accessed by a pointer to its first character
  - When two out-of-order lines have to be exchanged, the pointers in the pointer array are exchanged, not the text lines themselves.



Pointer Arrays; Pointers to Pointers

- `main()`

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

#define MAXLINES 5000         /* max #lines to be sorted */

char *lineptr[MAXLINES];      /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* sort input lines */
main()
{
    int nlines;     /* number of input lines read */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}
```

Pointer Arrays; Pointers to Pointers

- `readlines()`
- `writelines()`

```
#define MAXLEN 1000   /* max length of any input line */
int getline(char *, int);
char *alloc(int);

/* readlines:  read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        else {
            line[len-1] = '\0'; /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Pointer Arrays; Pointers to Pointers

- `qsort()`
- `swap()`

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)   /* do nothing if array contains */
        return;          /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Similarly, the swap routine needs only trivial changes:

```
/* swap:  interchange v[i] and v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Multi-dimensional Arrays

- C provides rectangular multi-dimensional arrays.
- they are much less used than arrays of pointers.
- Date Conversion
  - `day_of_year()`
    - converts the month and day into the day of the year
  - `month_day()`
    - converts the day of the year into the month and day
  - for leap years and non-leap years, it's easier to separate them into two rows of a two-dimensional array
  - for `leap`, is either zero (false) or one (true), so it can be used as a subscript of the array `daytab`.
  - We made `daytab` array of `char` to illustrate a legitimate use of `char` for storing small non-character integers.

Multi-dimensional Arrays

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year:  set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

  /* month_day:  set month, day from day of year */
  void month_day(int year, int yearday, int *pmonth, int *pday)
  {
      int i, leap;

      leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
      for (i = 1; yearday > daytab[leap][i]; i++)
          yearday -= daytab[leap][i];
      *pmonth = i;
      *pday = yearday;
  }
```

2020 4 20

1

20 ➡ 51 ➡ 80 ➡ 111

111 ➡ 80 ➡ 51 ➡ 20

Multi-dimensional Arrays

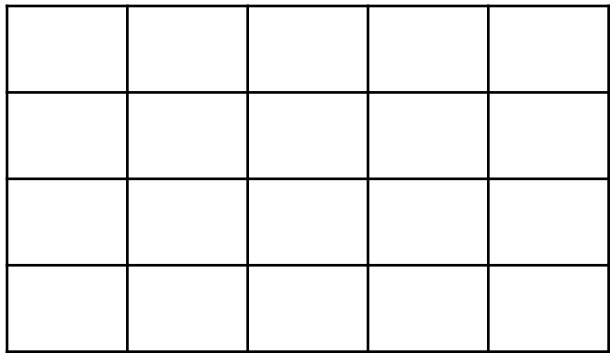
- If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of column

```
f (int daytab[2][13]) {…}
f (int daytab[ ][13]) {…}
f (int (*daytab)[13]) {…}
```

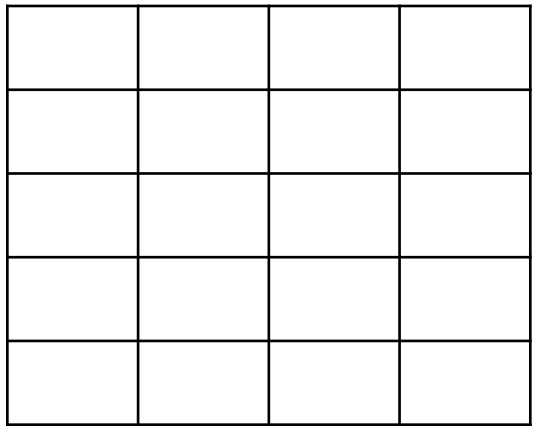
 is same.

- But `int *daytab[13]` is different.

a[4][5]



b[5][4]



Initialization of Pointer Arrays

- a function `month_name(n)`, which returns a pointer to a character string containing the name of the n-th month

```
/* month_name:  return name of n-th month */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

Pointers vs. Multi-dimensional Arrays

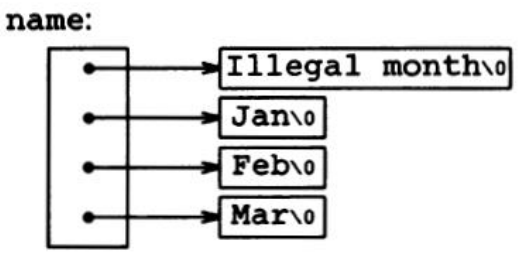
```
int a[10][20];
int *b[10];
```

- `a` is a true two-dimensional array: 200 int-sized locations have been set aside
- conventional rectangular subscript calculation `20xrow+col` is used to find the element `a[row][col]`.
- `b`, however, the definition only allocates 10 pointers and does not initialize them
- The important advantage of the pointer array is that the rows of the array may be of different lengths

Pointers vs. Multi-dimensional Arrays

- Array of pointers

```
char *name[] = { "Illegal month", "Jan, "Feb", "Mar" };
```



- 2-dim array

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```

aname:

Illegal month\0 Jan\0	Feb\0	Mar\0
0 15	30	45

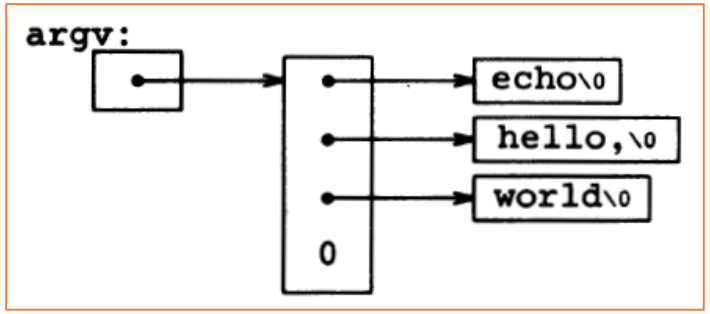
Command-line arguments

- a way to pass command-line arguments or parameters to a program when it begins executing
  - `argc`(argument count) : the number of command-line arguments the program was invoked with.
  - `argv`(argument vector):a pointer to an array of character strings that contain the arguments.
- The program `echo` echoes its command-line arguments on a single line, separated by blanks.

```
        echo hello, world

prints the output

        hello, world
```

Command-line arguments

- layout



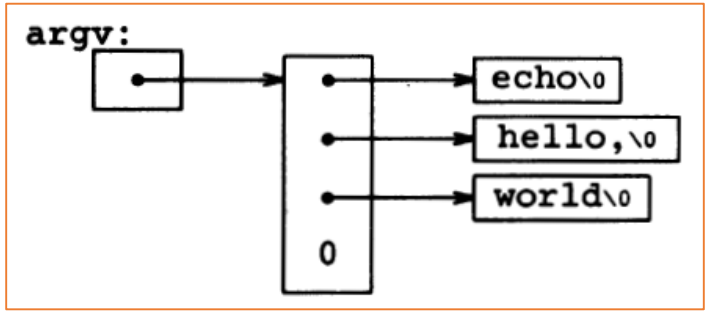
- echo: array version

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Command-line arguments

- echo: pointer version

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```



Command-line arguments

- UNIX grep: the pattern to be matched supplied by first argument on the command line

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

Command-line arguments

- A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter.
  - -x (except) to signal the inversion
  - -n (number) to request line numbering

```
find -x -n pattern
```

  - Optional arguments should be permitted in any order
  - it is convenient for users if option arguments can be combined

```
find -nx pattern
```

Command-line arguments

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

Pointers to Functions

- it is possible to define pointers to function
  - it can be assigned
  - it can be placed in arrays
  - it can be passed to functions
  - it can be returned by functions
- Sort revision
  - add –n option for numerical order sort

Pointers to Functions

- A sort often consists of three parts.
  - a comparison that determines the ordering of any pair of objects
  - an exchange that reverses their order
  - a sorting algorithm that makes comparisons and exchanges until the objects are in order
- Lexicographic comparison of two lines is done by `strcmp()`.
- `numcmp()` compares two lines on the basis of numeric value and returns the same kind of condition indication as `strcmp()` does.

124	<	1234	numcmp()
124	>	1234	strcmp()

Pointers to Functions

- main() of sort

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

#define MAXLINES 5000     /* max #lines to be sorted */
char *lineptr[MAXLINES];  /* pointers to text lines */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
int numcmp(char *, char *);

/* sort input lines */
main(int argc, char *argv[])
{
    int nlines;           /* number of input lines read */
    int numeric = 0;      /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void **) lineptr, 0, nlines-1,
          (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

Pointers to Functions

- qsort()

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
    int i, last;
    void swap(void *v[], int, int);

    if (left >= right)   /* do nothing if array contains */
        return;          /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Pointers to Functions

- `int (*comp)(void *, void *)` : comp is a pointer to a function that has two void * arguments and returns an int
- it is different from `int *comp(void *, void *)` : comp is a function returning a pointer to an int

Pointers to Functions

- numcmp()

```
#include <stdlib.h>

/* numcmp:  compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Complicated Declarations

- C declaration and a word description

```
char **argv
    argv:  pointer to pointer to char
int (*daytab)[13]
    daytab:  pointer to array[13] of int
int *daytab[13]
    daytab:  array[13] of pointer to int
void *comp()
    comp:  function returning pointer to void
void (*comp)()
    comp:  pointer to function returning void
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

정리

- Pointers and Addresses
- Pointers and Function Arguments
- Pointers and Arrays
- Address Arithmetic
- Character Pointers and Functions
- Pointer Arrays; Pointers to Pointers
- Multi-dimensional Arrays
- Initialization of Pointer Arrays
- Pointers vs. Multi-dimensional Arrays
- Command-line Arguments
- Pointers to Functions
- Complicated Declarations