BI
BIOINTELLIGENCE

Chapter 6. Arrays, Pointers, Strings

Byoung-Tak Zhang
TA: Hanock Kwak

Biointelligence Laboratory
School of Computer Science and Engineering
Seoul National Univertisy

http://bi.snu.ac.kr

Arrays

- Array
  - a simple variable with an index, or subscript, added
  - The brackets [] are used for array subscripts.
    - **int grade[3];**

```
#define N 100
...
int a[N];

// read subsequent N integers from input
for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
```

Arrays

- Array Initialization
  - Array may be of storage class automatic, external, or static, but NOT register.
  - Arrays can be initialized using an array initializer.

```
float f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
int a[100] = {1}; // = {1, 0, 0, 0, …}
int a[] = {2, 3, 5, -7};  // int a[4] = {2, 3, 5, -7};
```

  - If there is **fewer initializers** for an array than the number specified, **the missing elements will be zero** for external, static, and automatic variables.
  - **external** or **static** array
    - If not initialized explicitly, then initialized to zero by default

Arrays

- Array Subscripting
  - The expression, **a[i]**
    - refers to **(i+1)-th** element of the array **a** (zero-based ordering)
    - If **i** has a value outside the range from **0** to **(size of array) - 1**, then **Run-Time Error**
- The operators, **()** in function call and **[ ]** in array subscripting have
  - the highest precedence
  - left to right associativity

Pointers

- Pointers
  - used to access memory and manipulate addresses
  - If **v** is a variable, then **&v** is the location, or address, in memory space.
    - **&**: unary address operator, right-to-left associativity
- Pointer variables

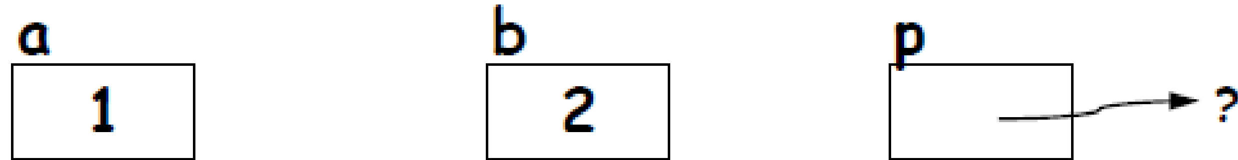
```
int * p; /*declares p to be of type pointer to int*/
p = 0;
p = NULL; /*equivalent to p = 0; */
p = &i;  /* address of i */
p = (int*) 1776; /* an absolute addr. in memory */
```

Pointers

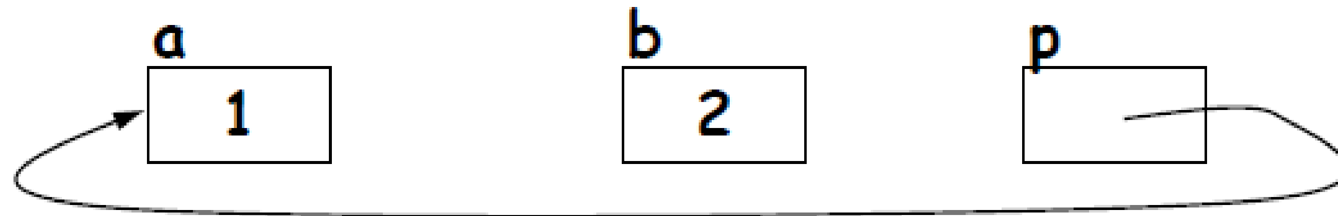
- Pointers
  - If **p** is a pointer, then ***p** is the value of the variable of which **p** is the address.
  - *: unary "**indirection**" or "**dereferencing**" operator, right-to-left associativity
  - The direct value of **p** is a memory location.
  - ***p** is the indirect value of **p**-namely, the value at the memory location stored in **p**.

Pointers

```
int  a = 1, b = 2, *p;
```



`p = &a;` "p is assigned the address of a"



`b = *p;` "b is assigned the value pointed to by p"

`b = *p;` ⇔ `b = a;`

Pointers

```
/* printing an address, or location */
#include <stdio.h>
int main()
{
    int i= 7, *p = &i;
    printf("%s%d\n%s%p\n", " Value of i: ", *p,
    "Location of i: ", p);
    return 0;
}

Value of i: 7
Location of i: 0x7fffb894d07c
```

Pointers

Declarations and Initializations

```
int     i = 3, j = 5, *p = &i, *q = &j, *r;
double  x;
```

Expression	Equivalent expression	Value
p == & i	p == (& i)	1
* * & p	* (* (& p))	3
r = & x	r = (& x)	/*illegal*/
7 * * p / * q + 7	((7 * (* p)) / (* q)) + 7	11
* (r = & j) *= *p	(* (r = (& j))) *= (* p)	15

Pointers

- Conversions during assignment between different pointer types are allowed
  - when one of the type is a pointer to **void**
  - when the right side is the constant **0**

Declarations and Initializations	
int *p; **float *q;** **void *v;**	
Legal assignments	Illegal assignments
p = 0;	p = 1;
p = (int *) 1;	v = 1;
p = v = q;	p = q;
p = (int *) q;	

Pointers

- Keep in mind the following prohibitions!
  - Do not point at constants.
    - **&3** **/* illegal */**
  - Do not point at ordinary expression.
    - **&(k+99)** **/* illegal */**
  - Do not point at register variable.
    - **register v;**
    - **&v** **/* illegal */**

Call-by-Reference

- “Call-by-value” mechanism
- “Call-by-reference” mechanism
  - for changing the values of variables in the calling environment
  - Pointers must be used in parameter list in the function definition.
    - 1. Declaring a function parameter to be a pointer
    - 2. Using the dereferenced pointer in the function body
    - 3. Passing an address as an argument when calling the function

Call-by-Reference

```
#include <stdio.h>
void swap(int *, int *);

int main(void)
{
    int i = 3, j = 5;
    swap(&i, &j);
    printf("%d %d\n", i, j); /* 5 3 is printed */
    return 0;
}

void swap(int *p, int *q)
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

Arrays and Pointers

- An array name
  - by itself is an address, or pointer value of a first element of the array
- Arrays and Pointers
  - can be subscripted.
  - A pointer variable can take different address as values
  - An array name is an FIXED address, or pointer.

Arrays and Pointers

```
#define N 100
int a[N], i, *p, sum = 0;
a[i];  // same as *(a+i) : the value of the i-th element of the array, a
```

- **a + i**
  - A pointer arithmetic
  - has as its value the **i**-th offset from the base address of the array, **a**
  - points to the **i**-th element of the array (counting from 0)

```
p[i];          // same as *(p+i)
p = a;         // p = &a[0];
p = a + 1; // p = &a[1];
```

- **p + i**
  - is the **i**-th offset from the value of p.
  - The actual address produced by such an offset depends on the type that **p** points to.

Arrays and Pointers

```
#define N 100
int a[N], i, *p, sum = 0;
```

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

```
for (i = 0; i < N; ++i)
    sum += *(a+i);
```

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

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

- Note that because **a** is a constant pointer, expressions such as
  `a = p;    ++a;    a += 2;    &a;`
  are illegal.

Pointer Arithmetic

- Pointer arithmetic

```
double a[2], *p, *q;
p = a;     /* points to base of array */
q = p + 1;      /* equivalent to q = &a[1] */
printf("%d\n", q-p);      /* 1 is printed */
printf("%d\n", (int) q - (int) p);     /* 8 is printed */
```

- **q - p**
  - yields the **int** value representing the number of array elements between **p** and **q**

Arrays as Function Arguments

- In a function definition, a formal parameter that is declared as an array is actually a pointer.
  - When an array is passed as an argument to a function, the base address of the array is passed by "call-by-value"

```
double sum(double a[], int n)
{ /* n is the size of a[] */
    int i;
    double sum = 0.0;
    for ( i = 0; i < n; ++i)
        sum += a[i];
    return sum;
}
```

```
double sum(double *a, int n)
{
….
```

Arrays as Function Arguments

Various ways that **sum()** might be called	
Invocation	What gets computed and returned
sum(v, 100)	v[0] + v[1] + ... + v[99]
sum(v, 88)	v[0] + v[1] + ... + v[87]
sum(&v[7], k-7)	v[7] + v[8] + ... + v[k-1]
sum(v+7, 2*k)	v[7] + v[8] + ... + v[2*k + 6]

Dynamic Memory Allocation

- **calloc()** and **malloc()**
  - declared in **stdlib.h**
  - **calloc()** : contiguous allocation
  - **malloc()** : memory allocation

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

int main(void)
{
    int *a;  /* to be used as an array */
    int n;  /* the size of the array */
    …       /* get n from somewhere */
    a = calloc(n, sizeof(int));  /* get space for a */
    …      /* use a as an array */
}
```

Dynamic Memory Allocation

- calloc() and malloc()
  - Returned value is NULL if the allocation failed.
  - **ptr = calloc(n, sizeof(int));**
    - The allocated memory is initialized with all bits set to zero.
  - **ptr = malloc(n * sizeof(int));**
    - does not initialize the memory space
- Space having been dynamically allocated MUST be returned to the system upon function exit.
  - **free(ptr);**
    - ptr must be the base address of space previously allocated.

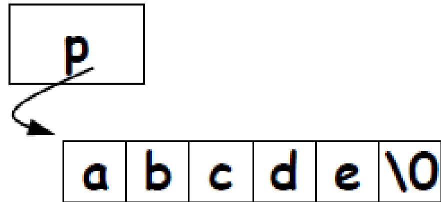
Strings

- Strings
  - one-dimensional array of type char
  - terminated by the end-of-string sentinel **‘\0’**, or null character (**0x00**)
  - The size of a string must include the storage needed for the null character.
    - **“abc”** : a char. array of size 4
  - String constant, like an array name by itself, is treated as a pointer

```
char *p = “abc”;
printf(“%s %s\n”, p, p+1); /*abc bc is printed */
```

Strings

- char *p = “abcde”;
  - allocates space in memory for p
  - puts the string constant **“abcde”** in memory somewhere else,
  - and initializes **p** with the base address of the string constant



- char s[] = “abcde”;   char s[]={‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’};
  - allocates 6 bytes of memory for the array s.

Strings

```
/*Count the number of words in a string */
#include <ctype.h>
int word_cnt(const char *s)
{
    int cnt = 0;
    while (*s != '\0') {
        while ( isspace(*s)) /* skip white space */
            ++s;

        if( *s != '\0') { /* found a word */
            ++cnt;
            while ( !isspace(*s) && *s != '\0' ) /* skip the word */
                ++s;
        }
    }
    return cnt;
}
```

Standard Library for String

- A standard header file, **string.h**

```
// concatenate s to s1
char *strcat(char *s1, const char *s);

// If s1 < s2, returns negative integer
// If s1 = s2, returns zero
// If s1 > s2, returns positive integer
int strcmp(const char *s1, const char *s2);

// copy string of s2 to s1
char *strcpy(char *s1, const char *s2);

// the number of characters before '\0'
size_t strlen(const char *s);
```

Standard Library for String

Declarations and Initializations	
char s1[] = "beautiful big sky country", s2[] = "how now brown cow";	
Expression	Value
strlen(s1)	25
strlen(s2 + 8)	9
strcmp(s1, s2)	negative integer
Expression	What gets printed
printf("%s", s1 + 10);	big sky country
strcpy(s1 + 10, s2 + 8);	
strcat(s1, "s!");	
printf("%s", s1);	beautiful brown cows!

Multidimensional Arrays

- C langauge allows arrays of any type, including arrays of arrays.
- Multi-dimensional array
  - using multiple brackets, [][]...
    - **int a[100]; a one-dimensional array**
    - **int b[2][7]; a two-dimensional array**
    - **int c[5][3][2]; a three-dimensional array**
  - Starting at the base address of the array, all the elements are stored contiguously in memory.

Multidimensional Arrays

- Two-dimensional array

	col 1	col 2	col 3	col 4	col 5
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Expressions equivalent to a[i][j]
*(a[i] + j)
(*(a + i))[j]
(((a + i)) + j)
*(&a[0][0] + 5*i + j)

- The array name ***a*** by itself is equivalent to ***&a[0]***; it is a pointer to an array of 5 ints.
- The base address of the array is ***&a[0][0]***, not ***a***.
  - Starting at the base address of the array, compiler allocate for 15 ints.

Multidimensional Arrays

- Formal Parameter Declarations
  - When a multidimensional array is a formal parameter in a function definition, all sizes except the first must be specified
    - so that the compiler can determine the correct mapping.

```
int a[3][5];

int sum(int a[][5])  // int a[][5] or  int a[3][5]  or  int (*a)[5]
{
    int i, j, sum = 0;
    for(i=0; i<3; ++i)
        for(j=0; j<3; ++j)
            sum += a[i][j];
    return sum;
}
```

Multidimensional Arrays

- Initialization
  - The indexing is by rows.
  - All sizes except the first must be given explicitly

```
int a[2][3]= {1,2,3,4,5,6};
int a[2][3]= {{1,2,3}, {4,5,6}};
int a[][3]= {{1,2,3}, {4,5,6}};
int a[][3]= {{1,0,0,}, {4,5,0}}; // int a[][3]= {{1}, {4,5}};
int a[2][3]= {0};
```

Arrays of Pointers

- Array elements can be of any type, including a pointer type.
- Ex) An array with elements of type **char ***
  - an array of strings

```
char * str_arr[4] = { “apple”, “banana”, “candy”, “duck” };
int i;
for (i = 0; i < 4; i++)
    printf("%s\n", str_arr[i]);
```

The Sorting Program

```
[input]
11
which all gets a slice of, come taste it and try.
```

```
[output]
a
all
and
come
gets
it
of,
slice
taste
try.
which
```

The Sorting Program

```
[main.c]

#include "sort.h"

int main(void)
{
    char word[MAXWORD];
    char *w[N];
    int n, i;
    scanf("%d",&n);
```

```
    for (i=0; i < n; ++i)
    {
        scanf("%s", word);
        w[i] = calloc(strlen(word) + 1,
        sizeof(char));
        strcpy(w[i], word);
    }

    sort_words(w, n); /* sort */
    wrt_words(w, n); /* write sorted words */

    for (i=0; i < n; ++i)
        free(w[i]);
    return 0;
}
```

The Sorting Program

```
[sort.h]

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

#define MAXWORD 50 /* max word size */
#define N 300 /* array size of w[] */

void sort_words(char *w[], int n);
void swap(char **p, char **q);
void wrt_words(char *w[], int n);
```

The Sorting Program

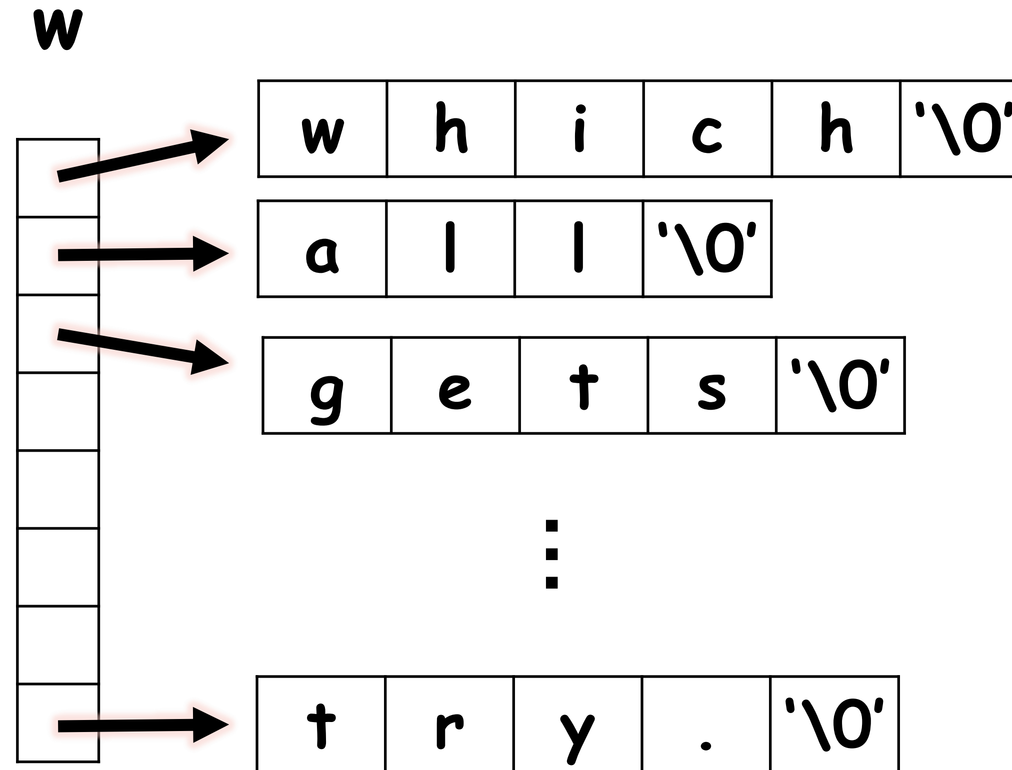
[sort.c]

```
#include "sort.h"

void sort_words(char *w[], int n)
{
    int i,j;
    for (i=0; i<n; ++i)
        for (j=i+1; j<n; ++j)
            if (strcmp(w[i], w[j]) > 0)
                swap(&w[i], &w[j]);
}
```

```
void swap(char **p, char **q)
{
    char *tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}

void wrt_words(char *w[], int n)
{
    int i;
    for (i=0; i<n; ++i)
        printf("%s\n", w[i]);
}
```

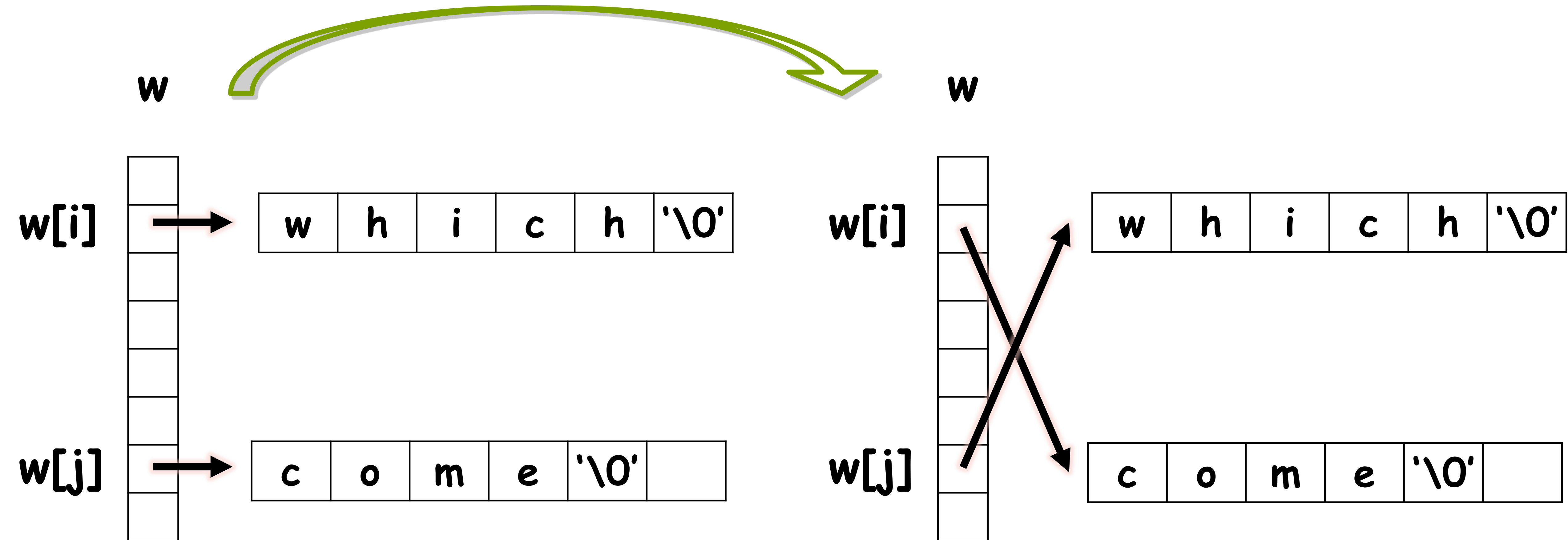
The Sorting Program

- char *w[N];



The Sorting Program

- swap(&w[i], &w[j]);



Arguments to main()

- Two arguments, **argc** and **argv**, can be used with **main().**

```
/* Echoing the command line arguments.*/
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    printf("argc = %d\n", argc);
    for (i=0; i< argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

[Command]

```
my_echo a is for apple
```

[Output]

```
argc = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```