Arrays, Pointers, and Strings

One-dimensional Arrays (1/3)

int grade0, grade1, grade2, grade3, grade4, grade5, grade6;

- Array
  - a simple variable with an index, or subscript
  - The brackets [] are used for array subscripts.

int grade[6];

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

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

✓ The indexing of array elements always starts from 0.

One-dimensional Arrays (2/3)

- 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] = {0};          /* initializes all elements of a to zero*/
int     a[] = {2, 3, 5, -7}; ⇔  int   a[4] = {2, 3, 5, -7};
```

  - If a list of initializers is shorter than the number of array elements, the missing elements are initialized to zero .
  - external or static array
    - If not initialized explicitly, then all elements are initialized to zero by default
  - automatic array
    - Is not necessarily initialized.

One-dimensional Arrays (3/3)

- Array Subscripting

  a[*expr***]**

  - **a[i]**
    - refers to (**i+1**)th element of the array **a**
    - If **i** has a value outside the range from 0 to N-1, then Run-Time Error (system dependent)
- **()** in function call and **[]** in array subscripting have
  - the highest precedence
  - left to right associativity

Example: Sorting

Input data: 4 -6 81 52 -23 15 9 13

```
loop
{   a minimum of a[i] ~ a[N-1] ⟶ a[i]
    increment i }
```

i=0: -23

i=1: -23 -6

i=2: -23 -6 4

⋮

i=N-1: -23 -6 4 9 13 15 52 81

Example: Bubble Sort Program

4	-6	81	52	-23	15	9	13
-23	4	-6	81	52	9	15	13
-23	-6	4	9	81	52	13	15

⋮

```
for (i=0; i<N-1; i++)
    for  (j=N-1; j>i; j--)
      if (a[j] <a[j-1])
          swap a[j],  a[j-1]
```

```
#include <stdio.h>
#define   N     8
void main(void)
{
    int  a[N]= {4, -6, 81, 52,-23, 5, 9, 15};
    int   i, j, tmp;
    printf("<Before Sorting>\n");
    for (j=0; j<N; j++)   printf("%d ", a[j]);
    for (i=0; i<N-1; i++)
         for (j=N-1; j>i; --j)
              if (a[j-1]>a[j]) {
                   tmp=a[j-1];
                   a[j-1]=a[j];
                   a[j]=tmp;
              }
    printf("\n<After Sorting>\n");
    for (j=0; j<N; j+)  printf("%d ", a[j]);
}
```

Example: Selection Sort Algorithm

```
#include <stdio.h>
void main(void)
{
    int  a[]= {4, -6, 81, 52,-23, 5, 9, 15};
    int  n = 8, i, j, tmp, min_ix;
    printf("<Before Sorting>\n");
    for (j=0; j<n; j++)   printf("%d ", a[j]);
    for (i=0; i<n-1; i++) {
        min_ix=i;
         for (j=i+1; j<n; j++)
             if (a[min_ix]>a[j]) min_ix=j;
         tmp=a[min_ix];
         a[min_ix]=a[i];
         a[i]=tmp;
    }
    printf("\n<After Sorting>\n");
    for (j=0; j<n; j++)  printf("%d  ", a[j]);
}
```

4	-6	81	52	-23	15	9	13
-23	-6	81	52	4	15	9	13
-23	-6	81	52	4	15	9	13
-23	-6	4	52	81	15	9	13

```
for (i=0; i<n-1; i++)
{
    min_ix=i;
    for  (j=i+1; j<n; j++)
      if (a[j] <a[min_ix])
          min_ix=j;

     swap a[min_ix], a[i]
}
```

Pointers (1/8)

- Pointers
  - If **v** is a variable, then **&v** is the address (location) in memory space.
    - **&**: unary address operator, right-to-left associativity
  - Pointer variable:
    - A variable which takes addresses as values
    - can be declared in program
      - When we want to declare p as a point variable, we should declare *p like a simple variable

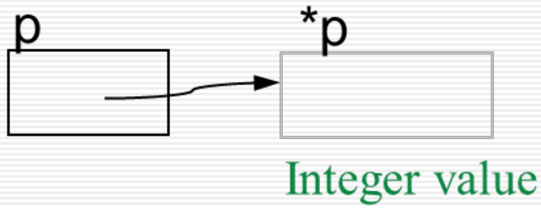
```
int   *p;
```

Pointers (2/8)

- 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 of the memory space of which address is stored in **p**.

```
int *p;
```



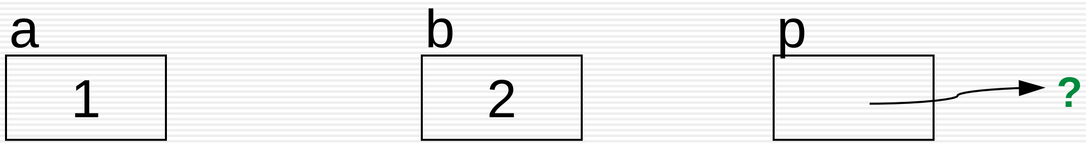
Pointers (3/8)

- a legal value of pointer variable
  - a special address 0
  - Positive integers being interpreted as machine addresses

```
p = 0;
p = NULL;                /*   equivalent to p = 0;   */
p = &i;
p = (int *) 1776;        /*  an absolute addr. in memory  */
```

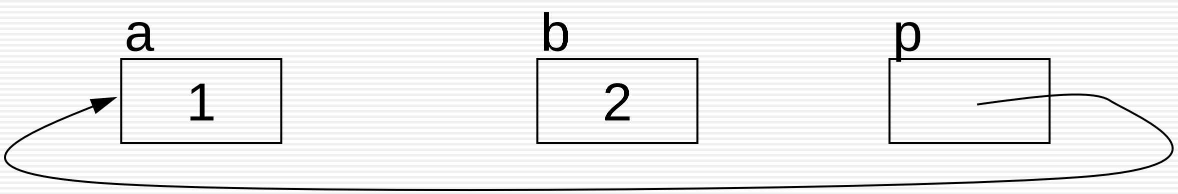
Pointers (4/8)

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



Since a value of p has not been assigned, we do not know yet what it points to

p = &a; “**p** is assigned the address of **a**”



b = *p; “**b** is assigned the value of storage pointed to by p”

b= *p; ⇔ b = a;

Pointers (5/8)

```
/*   printing an address    */
#include <stdio.h>
int main(void)
{
    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: effffb24
```

Pointers (6/8)

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 (7/8)

- 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 (8/8)

- 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 */

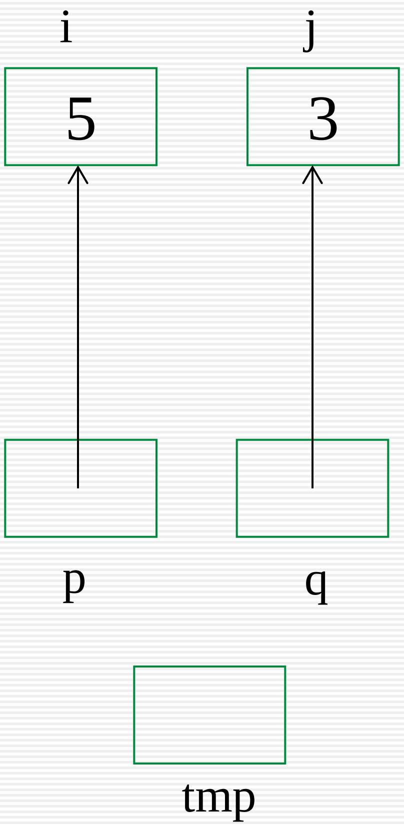
Call-by-Reference (1/2)

- "Call-by-value": parameter passing in C
  - The values of variables in the calling environment are unchanged.
- "Call-by-reference" mechanism
  - for changing the values of variables in the calling environment

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 (2/2)

```
#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 (1/3)

- An array name
  - by itself is an address
- Arrays and Pointers
  - can be subscripted.
  - A pointer variable can take different address as values
  - An array name is an FIXED address.

Arrays and Pointers (2/3)

#define N 100

int a[N], i, *p, sum = 0;

a[i] ⇔ ***(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)** ⇔ **p[i]**

- **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

p = a; ⇔ **p = &a[0];**

p = a + 1; ⇔ **p = &a[1];**

Arrays and Pointers (3/3)

```
#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,
  the following expressions are illegal.
  - a = p
  - ++a
  - a += 2
  - &a

Pointer Arithmetic and Element Size

- Pointer arithmetic
  - If the variable **p** is a pointer to a particular type,

    p + 1   p + i   ++p   p += i

```
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 "call-by-value"

```
double sum(double a[], int n)      ⇔      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;
}
```

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]

Example: Bubble Sort (1/2)

```
#include  <stdio.h>

void  swap(int *, int *);

void  bubble(int d[], int n)          ➡  void bubble(int *d, int n)
{
      int   i, j;

      for (i=0; i<n-1; ++i)
         for (j=n-1; j>i; --j)
               if (d[j-1]>d[j])
                     swap(&d[j-1], &d[j]);   ➡  swap(d+j-1, d+j);
}
```

Example: Bubble Sort (1/2)

```
void main(void)
{
        int  a[]= {4, -6, 81, 52,-23, 5, 9, 15};
        int  n = 8, i;
        printf("<Before Sorting>\n");
        for (i=0; j<n; i++)   printf("%d ", a[i]);
        bubble(a, 8);
        printf("\n<After Sorting>\n");
        for (i=0; i<n; i++)  printf("%d  ", a[i]);
}
void swap(int *p, int *q)
{
        int    tmp;
        tmp = *p;  *p = *q;  *q = tmp;
}
```

Dynamic Memory Allocation (1/2)

- **calloc(), 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 */
    ……
    a = calloc(n, sizeof(int));          /* get space for a */
    if  (a!=NULL)
    …..
}
```

Dynamic Memory Allocation (2/2)

- **calloc(), malloc()**

  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.

Example: Merge Sort (1/5)

mergesort.h

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

void  merge(int a[], int b[], int c[], int m, int n);
void  mergesort(int key[], int n);
void  wrt(int key[], int sz);
```

Example: Merge Sort (2/5)

a: 0 1 3 4 4 8 55 67 b: -5 -1 1 2 4 7 9 37

m = 8 merge n = 8

c: -5 -1 0 1 1 2 3 4 4 4 7 8 9 37 55 67

merge.c

```
#include  "mergesort.h"
void merge(int a[], int b[], int c[], int m, int n)
{
    int i=0, j=0, k=0;
    while (i<m && j<n)
        if (a[i]<b[j]) c[k++]=a[i++];
        else  c[k++]=b[j++];
    while (i<m)  c[k++]=a[i++];
    while (j<n) c[k++]=b[j++]
}
```

Example: Merge Sort (3/5)

k=1	key	4	3	1	67	55	8	0	4	-5	37	7	4	2	9	1	-1
	w																
k=2		3	4	1	67	8	55	0	4	-5	37	4	7	2	9	-1	1
k=4		1	3	4	67	0	4	8	55	-5	4	7	37	-1	1	2	9
k=8		0	1	3	4	4	8	55	67	-5	-1	1	2	4	7	9	37
		-5	-1	0	1	1	2	3	4	4	4	7	8	9	37	55	67

```
for (k=1; k<n; k*=2)
{
        for (j=0; j<n-k; j+=2*k)
            merge(key+j, key+j+k, w+j, k, k);

        for (j=0; j<n; ++j)   key[j]=w[j];
}
```

Example: Merge Sort (4/5)

mergesort.c

```
#include   "mergesort.h"
void mergesort(int key[], int n)
{
    int  j, k, m, *w;

    for (m=1; m<n; m*=2) ;
    if (n<m) exit(1);
    w=calloc(n, sizeof(int));
    assert(w!=NULL);
    for (k=1; k<n; k*=2) {
        for (j=0; j<n-k; j+=2*k)
            merge(key+j, key+j+k, w+j, k, k);
        for (j=0; j<n; ++j)  key[j]=w[j];
    }
    free(w);
}
```

Example: Merge Sort (5/5)

main.c

```
#include  "mergesort.h"
void main(void)
{
    int  sz;
    int  key[]={4, 3, 1,67,55, 8,0,4,-5,37,7,4,2,9,1,-1};
    sz=sizsof(key)/sizeof(int);
    printf("Before mergesort:\n);
    wrt(key,sz);
    mergesort(key,sz);
    printf("After mergesort:\n);
    wrt(key,sz);
 }
```

wrt.c

```
#include  "mergesort.h"
void wrt(int key[], int  sz)
{
    ...  /* print  */
}
```

Strings

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

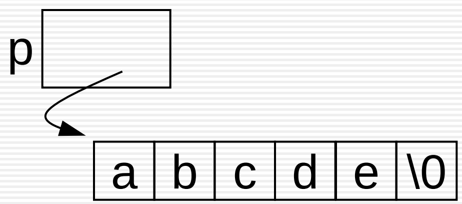
char *p = “abc”;

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

Strings

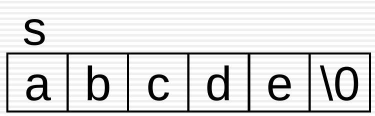
char *p = “abcde”;

- allocates space in memory for **p**
- puts the string constant **“abcde”** in memory somewhere else,
- initializes **p** with the base addr. of the string constant



char s[] = “abcde”; ⇔ **char s[]={‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘\0’};**

- allocates 6 bytes of memory for the array **s**.



Example: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;
}
```

String-Handling Functions in Standard Library

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

char *strcat(char *s1, const char *s2);

- A copy of s2 is appended to the end of s1

char *strcpy(char *s1, const char *s2);

- copies the string s2 to the string s1 including ‘\0’

size_t strlen(const char *s);

- the number of characters before **‘\0’**
- The type **size_t** is an integral unsigned type

```
size_t strlen(const char *s)
{
    size_t n;
    for(n = 0; *s != ‘\0’; ++s)   ++n;
    return n;
}
```

String-Handling Functions in Standard Library

```
char *strcpy(char *s1, const char *s2)
{
    char *p = s1;
    while (*p++ = *s2++) ;
    return s1;
}
```

- ✓ *p++ ⇔ *(p++)
  : p itself is being incremented.
- ✓ (*p)++
  : would increment what p is pointing to.

```
char *strcat(char *s1, const char *s2)
{
    char *p = s1;
    while (*p)   ++p;      /* go to the end */
    while (*p++ = *s2++) ;   /* copy */
    return s1;
}
```

- ✓ while (*p) ⇔ while (*p != '\0')

Note that it is the programmer's responsibility to allocate sufficient space for the strings that are passed as arguments to these functions

String-Handling Functions in Standard Library

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 (1/4)

- C language allows multi-dimensional arrays, including arrays of arrays.
- Two-dimensional array: using two bracket pairs, [ ][ ]

```
int a[100];            /*   a one-dimensional array  */
int  b[2][7];          /*   a two-dimensional array   */
double  c[5][3][2];    /*   a three-dimensional array */
```

- *k*-dimensional array
  - allocates space for $s_1 \times s_2 \times \ldots \times s_k$ elements, where s_i represents the size of ith dimension.
  - Starting at the base address of the array, all the elements are stored contiguously in memory.

Multidimensional Arrays (2/4)

- Two-dimensional array

  int a[3][5];

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
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[0][0] + 5*i + j)

  - a[i]: the address of *i*th low
  - The base address of the array is **&a[0][0]**.
    - Starting at the base address of the array, compiler allocate for 15 integers.

Multidimensional Arrays (3/4)

- 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   sum(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 (4/4)

- 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}, {4,5}};  ⇔  int a[][3] = {{1,0,0}, {4,5,0}};

int a[2][3] = {0};
```

Storage Mapping Function

- mapping between pointer values and array indices
  - int a[3][5]
    - a[i][j] : *(&a[0][0] + 5*i + j)
  - int a[7][9][2]
    - a[i][j][k] : *(&a[0][0][0] + 9*2*i + 2*j + k)

- all sizes except the FIRST must be specified so that the compiler can determine the correct storage mapping function

Use of **typedef**

```
#define    N     3
typedef double scalar;
typedef scalar vector[N];
typedef scalar matrix[N][N];
```

```
void add(vector x, vector y, vector z)
{  /* x = y + z */
     int     i;
     for (i = 0; i < N; ++i)
          x[i] = y[i] + z[i];
}
scalar dot_product(vector x, vector y)
{
     int     i;
     scalar sum = 0.0;
     for (i = 0; i < N; ++i)
          sum += x[i] * y[i];
     return sum;
}
void multiply(matrix a, matrix b, matrix c)
{  /* a = b * c */
     int    i, j, k;
     for (i = 0; i < N; ++i) {
         for (j = 0; j < N; ++j) {
               a[i][j] = 0.0;
               for (k = 0; k < N; ++k)
                    a[i][j] += b[i][k] * c[k][j];
          }
     }
}
```

Arrays of Pointers (1/5)

- Array elements can be of any type, including a pointer type.
- An array of pointers
  - An example program : lexicographically sorting words in a file

Input

A is for apple or alphabet pie

which all get a slice of, come taste it and try.

- an string array
  - Array with elements of which type is a character pointer, char *

Output

A
a
all
alphabet
and
…
which

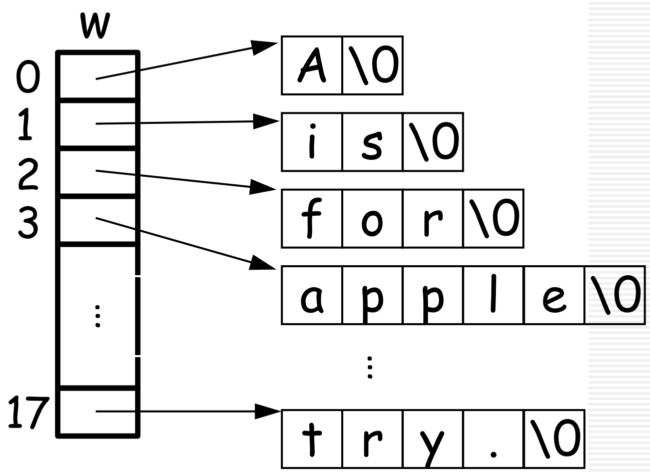
Arrays of Pointers (2/5)

In file 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);
```

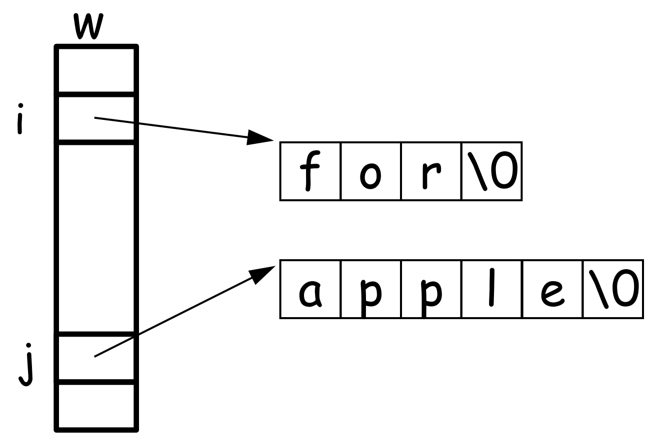
[main.c] /*Sort words lexicographically. */

```
#include "sort.h"
int main(void)
{
        char word[MAXWORD];     /* work space */
        char *w[N];             /* an array of pointers */       ⇔ char *(w[N]);
        int n;                  /* number of words to be sorted */
        int i;

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

        sort_words(w, n);       /* sort the words */
        wrt_words(w, n);        /* write sorted list of words */
        return;
}
```



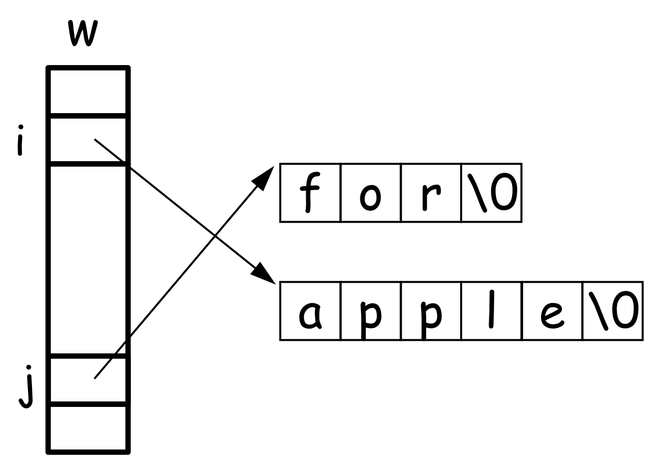
[sort.c]

```
#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]);
}
```

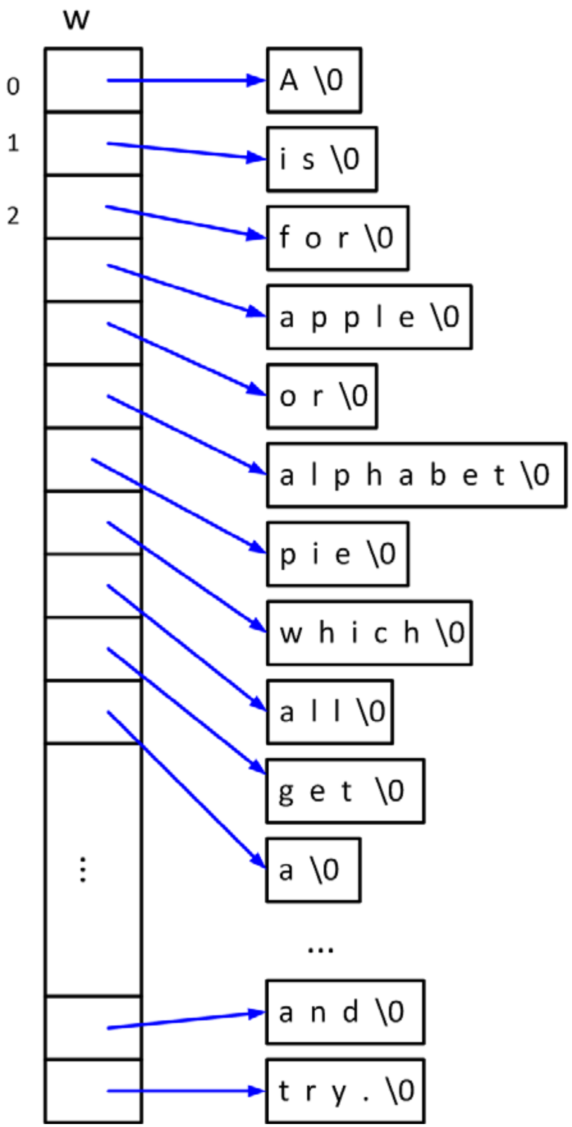


[swap.c]

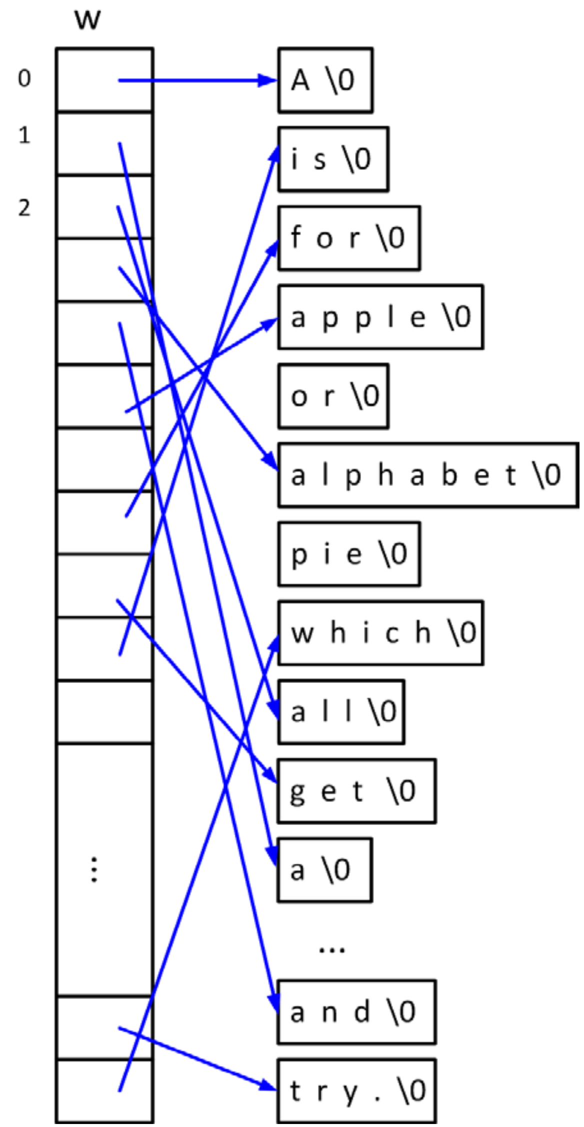
```
#include "sort.h"
void swap(char **p, char **q)
{
        char *tmp;

        tmp = *p;
        *p = *q;
        *q = tmp;
}
```



Before sorting



After sorting



[wrt.c]

```
#include "sort.h"
void wrt_words(char *w[], int n)
{
        int i;

        for (i=0; i<n; ++i)
        printf("%s₩n", w[i]);
}
```

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

Ragged Arrays (1/3)

- Ragged array
  - an array of pointers whose elements are used to point to arrays of varying sizes

```
#include <stdio.h>
int main(void)
{
        char a[2][15]={“abc:”, “a is for apple”};
        char *p[2]= {“abc:”, “a is for apple”};
        printf(“%c%c%c %s %s\n”, a[0][0], a[0][1], a[0][2], a[0], a[1]);
        printf(“%c%c%c %s %s\n”, p[0][0], p[0][1], p[0][2], p[0], p[1]);
}
```

- (output)

abc abc: a is for apple

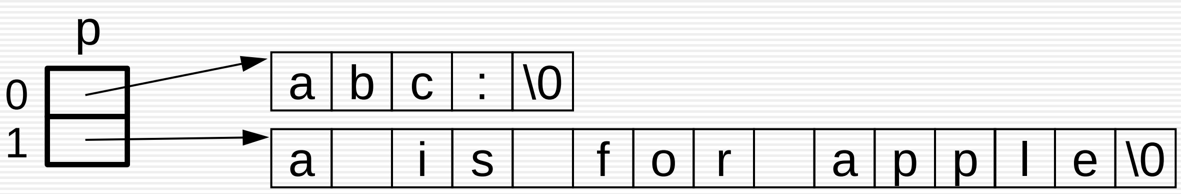
abc abc: a is for apple

Ragged Arrays (2/3)

- **char a[2][15] = {“abc:”, “a is for apple”};**
  - Space for 30 **chars** is allocated
  - Each of **a[0]** and **a[1]** is an array of 15 **chars**
  - **a[0]** and **a[1]** are strings
  - **char a[2][15] = {{‘a’, ‘b’, ‘c’, ‘:’, ‘\0’}, {‘a’, ‘ ’, ‘i’, ‘s’, … ‘\0’}};**
    - The array a[0] is initialized to {‘a’, ‘b’, ‘c’, ‘:’, ‘\0’} and ten null characters, i.e., ‘\0’ (decimal zero)
  - Compiler generates a storage mapping function for accessing array element a[i][j].

Ragged Arrays (3/3)

- **char *p[2]   = {“abc:”, “a is for apple”};**
  - one-dimensional array of pointers to **char**
  - It causes space for two pointers to be allocated.
  - **p[0]** is initialized to point at **“abc:”**, a string constant of 5 characters, thus there is no way to modify “abc:” (e.g. p[0][3]=‘d’ is not allowed)
  - **p[1]** is initialized to point at **“a is for apple”**, a string constant of 15 characters including the null characters \0 at the end of the string.
  - **p** does its work in less space than **a**
  - Compiler does not generate a storage mapping function for accessing array elements ⇒ faster working than **a**



Function as Arguments (1/2)

- We calculate $\sum_{k=m}^{n} f^2(k)$ for a variety of functions
  - $f(k) = \sin(k)$
  - $f(k) = x^2 - 7x + 5$

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

double sum_square(double f(double), int m, int n)
{
	int k;
	double sum=0.;

	for (k=m; k<=n; ++k)
		sum+=f(k)*f(k);
	return sum;
}
```

Function as Arguments (2/2)

```
double polynomial(double x)
 {
      return (x*x-7*x+5);
  }

int main(void)
{
    printf(“%s%e\n”,“Function  x*x-7*x+5:”,
                 sum_square(polynomial, 1, 100);
    printf(“%s%e\n”,“Function  sin(x):”, sum_square(sin, 2, 13);
    return 0;
 }
```

Function Pointers

- f   a pointer to a function
  *f   the function itself
  (*f)(k)   the call to the function

```
double sum-square( double f(double), int m, int n)
/* ⇔  double sum-square( double (*f)(double), int m, int n) */
{
     int k;
     double sum=0.;

     for (k=m; k<=n; ++k)
          sum+=f(k)*f(k);           /* ⇔  sum+=(*f)(k) * (*f)(k);  */

     return sum;
}
```