Arrays and Pointers in C

010.133
Digital Computer Concept and Practice
Spring 2013

Lecture 11

Arrays

- An array is a data structure containing a certain number of elements, all of which have the same type
- To declare an array, specify the type and number of the elements
- For example, `int a[10];` declares `a` to be an array of 10 integers

Arrays (contd.)

- To access an element of an array, write the array name followed by a subscript
- In C, subscripts always start with 0
- For example, the elements of array **`a`** are **`a[0]`**, **`a[1]`**, … , **`a[9]`**

Arrays (contd.)

- An array is initialized by listing the values
- If the number of values is less than that of the array elements, the remaining elements are given value 0

```
int a[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
int a[10] = {0};
float a[3] = { 0.1, 0.2, 0.0 };
int a[3] = { 3, 4 };
int a[] = { 2, -4, 1 };
int a[4] = { 2, -4, 1 };
char s[] = "abcd";
char s[] = { 'a','b','c','d','\0' }
```

Finding Maximum

- The program below reads ten values into an array **ab** and then finds the maximum among the values

```
#include <stdio.h>
int main(void)
{
    int n, i, max;
    int ab[10];

    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter n numbers: ");
    for (i=0; i<n; i++)
        scanf("%d", &ab[i]);
    max = ab[0];
    for (i=1; i<n; i++)
        if (ab[i] > max) max = ab[i];
    printf("max %d\n", max);
    return 0;
}
```

Bubble Sort

- Sorts array **`ab`** of n elements in non-decreasing order
- The first iteration of the inner for loop brings the maximum element to the last position, the second iteration brings the second maximum to the second-to-last position, etc.
- The three assignments inside the if statement exchange the values of **`ab[j]`** and **`ab[j+1]`**

Bubble Sort (contd.)

```
for (i=1; i<n; i++)
  for (j=0; j<n-i; j++)
    if (ab[j] > ab[j+1]) {
      temp = ab[j];
      ab[j] = ab[j+1];
      ab[j+1] = temp;
    }
```

Insertion Sort

```
void insertionSort( int a[], int n )
{
    int i, j, val;
    for( i = 1; i < n; i++){
        val = a[i];
        j = i - 1;
        while ( ( j >= 0 ) && ( a[j] > val ) ) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = val;
        printIntArray( a, n );
    }
}
```

Insertion Sort (contd.)

```
#include <stdio.h>
#define N 10

void insertionSort( int *, int );
void printIntArray( int *, int );

int main( void )
{
    int a[N] = { 23, -3, 5, 9, 11,
                 33, 87, -7, -24, 50 };
    printIntArray( a, N );
    insertionSort( a, N );
    return 0;
}
```

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

    for( i = 0; i < n; i++)
        printf("%4d ", a[i]);

    printf("\n");
}
```

Insertion Sort (contd.)

23	-3	5	9	11	33	87	-7	-24	50
-3	23	5	9	11	33	87	-7	-24	50
-3	5	23	9	11	33	87	-7	-24	50
-3	5	9	23	11	33	87	-7	-24	50
-3	5	9	11	23	33	87	-7	-24	50
-3	5	9	11	23	33	87	-7	-24	50
-3	5	9	11	23	33	87	-7	-24	50
-7	-3	5	9	11	23	33	87	-24	50
-24	-7	-3	5	9	11	23	33	87	50
-24	-7	-3	5	9	11	23	33	50	87

Search

- Suppose that n elements are stored in an array **`ab`**
- Given a new element x, we want to find if x is in array **`ab`**
  - x is one of the elements stored in **`ab`**
- An easy solution for search is to scan the elements in array **`ab`** one by one and check if it is equal to x
  - Linear search

Linear Search

```
#include <stdio.h>
int main(void)
{
    int n, i, x;
    int ab[100];

    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter n numbers: ");
    for (i=0; i<n; i++)
        scanf("%d", &ab[i]);
    printf("Enter x: ");
    scanf("%d", &x);
    for (i=0; i<n; i++)
        if (x == ab[i]) {
            printf("%d\n", i);
            return 0;
        }
    printf("%d\n", -1);
    return -1;
}
```

Binary Search

- If the elements in array **`ab`** are stored in non-decreasing order after sorting, we can solve the search problem more efficiently than linear search
- We first compare x with the element in the middle (i.e., median)
  - If x is equal to the median, we have found it
  - If x is smaller than the median, we are sure that x is not in the upper part of array **`ab`**, so we look for x in the lower part
  - Otherwise, x is larger than the median, we look for x in the upper part
- Binary search is faster than linear search

Binary Search (contd.)

```
low = 0;
high = n-1;
while (low <= high) {
    mid = (low+high) / 2;
    if (x < ab[mid])
        high = mid - 1;
    else if (x > ab[mid])
        low = mid + 1;
    else {
        printf("%d\n", mid);
        return 0;
    }
}
printf("%d\n", -1);
return -1;
```

Two-dimensional Arrays

- Two dimensional arrays can be visualized as a multicolumn table or grid
- `int b[2][5];`
  - Declares a two-dimensional array **`b`** that has 2 rows and 5 columns
- We can initialize a two-dimensional array as follows:
  - `int b[2][5] = {{1,0,0,1,1},{0,0,1,1,1}};`

Exercise 1

- A program that reads a number of elements, stores them in an array, and computes the average and the standard deviation of the elements

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    int n, i;
    double ab[100], avg, sd;

    printf("Enter n: ");
    scanf("%d", &n);
    printf("Enter n numbers: ");
    for (i=0; i<n; i++)
        scanf("%lf", &ab[i]);
    avg = 0;
    for (i=0; i<n; i++)
        avg += ab[i];
    avg /= n;
    sd = 0;
    for (i=0; i<n; i++)
        sd += (ab[i]-avg) * (ab[i]-avg);
    printf("Average: %f\nStandard deviation: %f\n",avg,sqrt(sd/n));
    return 0;
}
```

Exercise 1 (contd.)

- **`avg += ab[i];`** stands for **`avg = avg + ab[i];`**
- **`avg /= n;`** stands for **`avg = avg / n;`**
- In general, **`exp1 op= exp2`** means **`exp1 = exp1 op exp2`** for most binary operators such as +, -, *, /, and %

Math.h

- The library <math.h> contains mathematical functions (x is of type double, and all functions return double)
- sqrt(x) square root of x
- exp(x) exponential function ex
- log(x) natural logarithm ln(x)
- log10(x) log10(x)
- sin(x) sine of x
- cos(x) cosine of x
- tan(x) tangent of x

Exercise 2

- Write a program that multiplies two NxN matrices

Pointers (revisited)

```
int i = 4, j = 6, *p = &i, *q = &j, *r;

if (p == &i) ...;
if (p == (& i)) ...;
... = **&p;
... = *(*(& p));
... = 9 * *p / *q + 8;
... = (((9*(*p)))/(*q)) + 8;
*(r = &i) *= *p;
(* (r = (& j))) *= (* p);
```

Pointers (contd.)

```
int *p;
float *q;
void *v; /* void*: generic pointer type */
p = 0;
p = v = q;
p = (int *) 3;
p = (int *) q;
```

Pointers as Function Arguments

- Suppose we want to make a function that returns the maximum and the minimum of three integers a, b, and c
  - We cannot pass the results with the return mechanism of the function because we need to return two values
  - Use pointers
- The code in the next slide shows such a function
  - We can call it by
    - `maxmin(a, b, c, &max, &min);`

Pointers as Function Arguments (contd.)

```
void maxmin(int a, int b, int c, int *pmax, int *pmin) {
    if (a >= b) {
        if (a >= c) {               /* a is maximum */
            *pmax = a;
            if (b >= c) *pmin = c;
            else *pmin = b;
        } else {                    /* c > a >= b */
            *pmax = c;
            *pmin = b;
        }
    } else {
        if (b >= c) {               /* b is maximum */
            *pmax = b;
            if (a >= c) *pmin = c;
            else *pmin = a;
        } else {                    /* c > b > a */
            *pmax = c;
            *pmin = a;
        }
    }
}
```

Swap Function

```
#include <stdio.h>

void swap(int*, int*);

int main(void)
{
    int x = 4, y = 5;
    swap( &x, &y );
    printf("%d %d\n", x, y);
    return 0;
}

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

```
#include <stdio.h>

void swap(int, int);

int main(void)
{
    int x = 4, y = 5;
    swap( x, y );
    printf("%d %d\n", x, y);
    return 0;
}

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

Pointers and Arrays

- An array name is actually a constant pointer
  - Its value cannot be changed
- When x is an array, **`x[i]`** is the same as **`*(x + i)`**
- When p is a pointer, **`*(p + i)`** is the same as **`p[i]`**

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

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

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

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

Pointer Arithmetic

- If p is a pointer to a particular type, the expression p + 1 gives the address for the storage of the next variable of that type

```
double a[10], *p, *q;
p = a;          /* p points to the first element of a */
q = p + 2;      /* q points to the third element of a */
printf("%d\n", q - p)                    /* q - p is 2 */
printf("%d\n", (int) q - (int) p)        /* 16  */
```

Arrays as Function Arguments

- The base address of the array is passed to the function

```
double sum(double x[], int n)
   /* ≡ double sum(double *x, int n) */
{
    int i;
    double sum = 0.0;
    for (i = 0; i < n; i++)
        sum += x[i];
    return sum;
}

double y[ 100 ];
…
sum(y, 100);
sum(y, 20);
sum(&y[10], 20);
sum(y + 10, 20);
```

Arrays and Strings

- A character array can be initialized with a string constant when it is declared
  - `char str[12] = "programming";`
- The length of str should be the number of characters in the string constant plus 1
  - The last byte contains the null character
- The length of the array in the example above may be omitted
  - `char str[] = "programming";`
  - 12 bytes are assigned to str by the compiler

Arrays and Strings (contd.)

- If there is no room for the null character as in the example below, **`carr`** cannot have a terminating null character and it is not a string
  - **`char carr[11] = "programming";`**
  - It is an array of characters
- The conversion specification for a string in both printf and scanf is %s

String Constants

- String constants are written between double quotes
  - Treated as a pointer
  - The value is the base address of the string
- `char *p = "abc";`
- `printf("%s %s\n", p, p+1);`
- `"abc"[2]`
- `*("abc" + 2)`
- `char s[] = "abc";`
- `char s[] = { 'a', 'b', 'c', '\0' };`

String Pointers

- Consider the following declarations:
  - `char str[] = "programming";`
  - `char *pstr = "programming";`
- The first one declares a string variable **`str`**, i.e., an array of characters
  - We can modify the characters in **`str`**
    - `str[3] = 't';`
- The second one declares a pointer variable **`pstr`**, which points to a string constant
  - We may modify the pointer itself, but may not modify characters in the string constant

Pointer Arrays

- Suppose we want to store an array of strings such as country names
- The best way is to use an array of pointers
  - `char *pcountry[] = {"Korea", "China", "Japan", "U.S.A.", "Russia"};`
- Then **`pcountry[0]`** is a pointer that points to "Korea", **`pcountry[1]`** is a pointer to "China", etc.

Simple Crypto-system

- A cryptosystem consists of an encryption function and a decryption function
  - The encryption function gets a plaintext and a key, and produces a ciphertext
  - The decryption function gets a ciphertext and a key, and produces a plaintext
  - If the decryption key is the same as the encryption key, the decryption function should produce the original plaintext

Simple Crypto-system (contd.)

- The Shift Cipher is one of the simple cryptosystems
  - In fact, it is too simple to be secure, but it was actually used in history
  - Assume that plaintexts and ciphertexts are strings of lowercase alphabet letters
  - The key is an integer k between 0 and 25

Simple Crypto-system (contd.)

- The encryption function shifts each letter of the plaintext to the right by k positions (modulo 26)
- For example, if the key is 3, each letter of the plaintext is changed to a ciphertext letter as shown in the table below

plaintext	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
ciphertext	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c

Simple Crypto-system (contd.)

- If the plaintext is "hello" and the key is 3, the ciphertext becomes "khoor"
- The decryption function shifts each letter of the ciphertext to the left by k positions (modulo 26)

Encryption Function

```
#include <stdio.h>
#define SMAX 100
#define KMOD 26

void EncShift(char ptext[],
              char ctext[],
              int key)
{
    int i=0;

    while (ptext[i] != '\0') {
        ctext[i] = (ptext[i]-'a'+key)
                 % KMOD + 'a';
        i++;
    }
    ctext[i] = '\0';
}
```

```
int main(void)
{
    char ptext[SMAX],
    char ctext[SMAX];
    int key;

    printf("Enter plaintext: ");
    scanf("%s", ptext);
    printf("Enter key: ");
    scanf("%d", &key);
    EncShift(ptext, ctext, key);
    printf("Ciphertext: %s\n", ctext);

    return 0;
}
```

Decryption Function

- Can you write a decryption function?

Swapping with XOR

- S1: *x = a^b, *y = b
- S2: *x = a^b, *y = (a^b)^b = a
- S3: *x = (a^b)^a = b, *y = a

```
void swap(int *x, int *y)
{
     *x = *x ^ *y; /* S1 */
     *y = *x ^ *y; /* S2 */
     *x = *x ^ *y; /* S3 */
}
```

Enumeration Constants

- A data type consisting of a set of named values called elements, members or enumerators of the type
- The enumerator names are usually identifiers
  - Behave as constants
- A means of naming a finite set and a user defined type
  - `enum` keyword
- Integers and enum values can be mixed freely
  - All arithmetic operations on enum values are permitted
- The programmer can choose the values of the enumeration constants explicitly

Enumeration Constants (contd.)

- `enum day { sun, mon, tue, wed, thu, fri, sat };`
- `enum day { sun = 1, mon, tue, wed, thu, fri, sat } p, q, r;`
- `enum day { sun = 7, mon, tue, wed = 2, thu, fri, sat } p, q, r;`

Enumeration Constants (contd.)

- The nextDay function returns the next day of a given day

```
enum day { sun, mon, tue,
           wed, thu, fri, sat };

typedef enum day day;

day nextDay( day d )
{
    day next_day;
    switch( d ) {
        case sun: next_day = mon;
        break;
        case mon: next_day = tue;
        break;
        …
        case sat: next_day = sun;
        break;
    }
    return next_day;
}
```

```
enum day { sun, mon, tue,
           wed, thu, fri, sat };

typedef enum day day;

day nextDay( day d )
{
    assert((int) d >= 0
              && (int) d < 7);

    return ((day)(((int)d+1)%7));
}
```

Signed vs. Unsigned in C Revisited

```
/* Assume 32-bit word */
int x = …;
int y = …;
unsigned ux = x;
unsigned uy = y;
```

- If x < 0 then (x*2) < 0
  - False when x = SignedMin
- ux >= 0 is always true
  - True
- If x & 15 == 15 then (x << 30) < 0
  - True
- ux > -1 is always true
  - False when ux = 0

Signed vs. Unsigned in C Revisited (contd.)

```
/* Assume 32-bit word
*/
int x = …;
int y = …;
unsigned ux = x;
unsigned uy = y;
```

- x * x >= 0 is always true
  - False when x = 30426
- If x > 0 && y > 0 then x + y > 0
  - False when x = SignedMax, y = SignedMax
- If x >= 0 then -x <= 0
  - True
- If x <= 0 then -x >= 0
  - False when x = SignedMin

Functions as Arguments

- Pointers to functions can be passed as arguments

```
double sum_square(double f(double x), int m, int n)
{
    int k;
    double sum = 0.0;
    for (k=m; k <= n; ++k)
        sum += f(k) * f(k);
    return sum;
}
```

```
double sum_square(double f(double), int m, int n)
{
    ...
}
```

```
double sum_square(double (*f)(double), int m, int n)
{   ...
    sum+= (*f)(k) * (*f)(k);
    ...
}
```

(*f)(k)

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

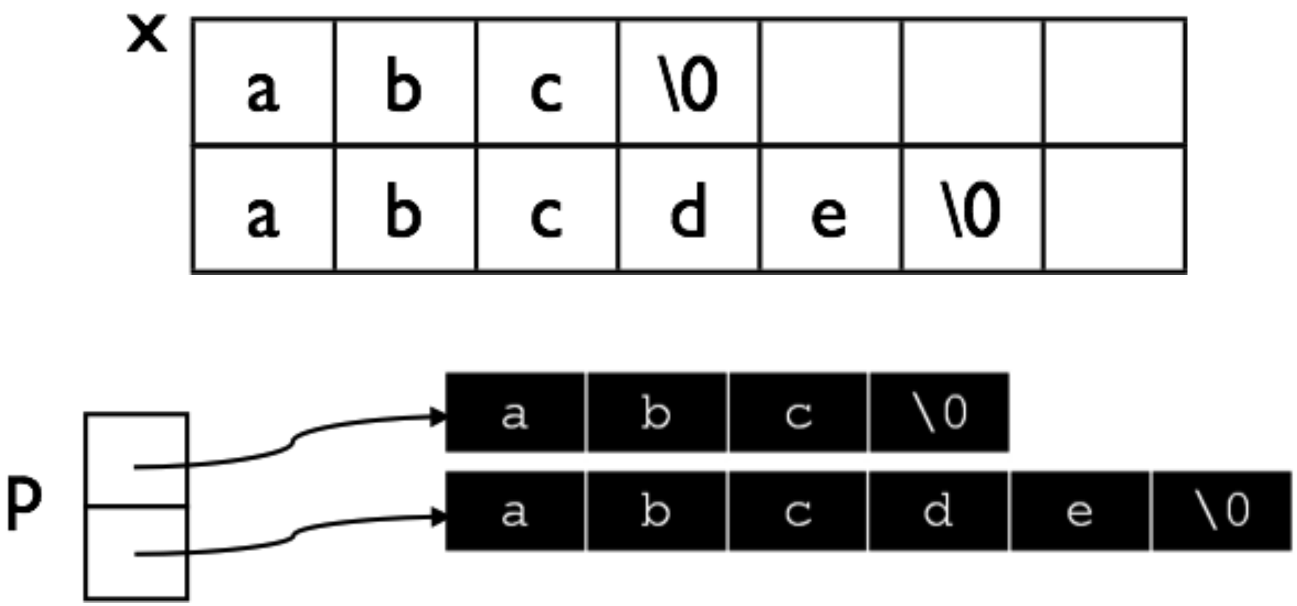
- All the following function prototypes are the same
  - `double sum_square(double f(double x), int m, int n);`
  - `double sum_square(double f(double), int m, int n);`
  - `double sum_square(double f(double), int, int);`
  - `double sum_square(double (*f)(double), int, int);`
  - `double sum_square(double (*)(double), int, int);`
  - `double sum_square(double g(double y), int a, int b);`

(*f)(k) (contd.)

```
#include <math.h>
#include <stdio.h>
double f(double x);
double sum_square(double f(double x), int m, int n);
int main(void){
    printf("%.7f\n%.7f\n", sum_square(f, 1, 10000),
           sum_square(sin, 2, 13));
    return 0;
}
double f(double x){
    return 1.0/x;
}
```

Ragged Arrays

- An array of pointers whose elements are used to point to arrays of varying sizes.
- `char x[2][7] = {"abc", "abcde" };` (normal arrays)
- `char *p[2] = {"abc", "abcde" };` (ragged arrays)



Command-line Arguments

- To communicate with the OS
- **`argc`**: the number of command line arguments
- **`argv`**: an array of strings
  - The strings are the words that make up the command line
  - **`argv[0]`** contains the name of the command itself

Command-line Arguments (contd.)

```
#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-line Arguments (contd.)

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

int main(int argc, char* argv[])
{
    int a, b;

    if (argc < 3) {
        printf("usage: %s <operand1> <operand2>\n", argv[0]);
        return 1;
    }

    a = atoi(argv[1]);
    b = atoi(argv[2]);

    printf(%d times %d is %d.\n", a, b, a*b);

    return 0;
}
```