

One-Dimensional Arrays

Random-access lists of elements

CS10003 PROGRAMMING AND DATA STRUCTURES



Array

Many applications require multiple data items with common characteristics.

- In mathematics, we often express such groups of data items in indexed form:

$$x_1, x_2, x_3, \dots, x_n$$

Array is a data structure that can represent a collection of data items with the same data type (**float** / **int** / **char** / ...).

Example: Printing Numbers in Reverse

3 numbers

```
int a, b, c;  
scanf("%d", &a);  
scanf("%d", &b);  
scanf("%d", &c);  
printf("%d  ", c);  
printf("%d  ", b);  
printf("%d  \n", a);
```

4 numbers

```
int a, b, c, d;  
scanf("%d", &a);  
scanf("%d", &b);  
scanf("%d", &c);  
scanf("%d", &d);  
printf("%d  ", d);  
printf("%d  ", c);  
printf("%d  ", b);  
printf("%d  \n", a);
```

The Problem

Suppose we have 1000 numbers to handle.

Where do we store the numbers? Use 1000 variables? Sounds absurd!

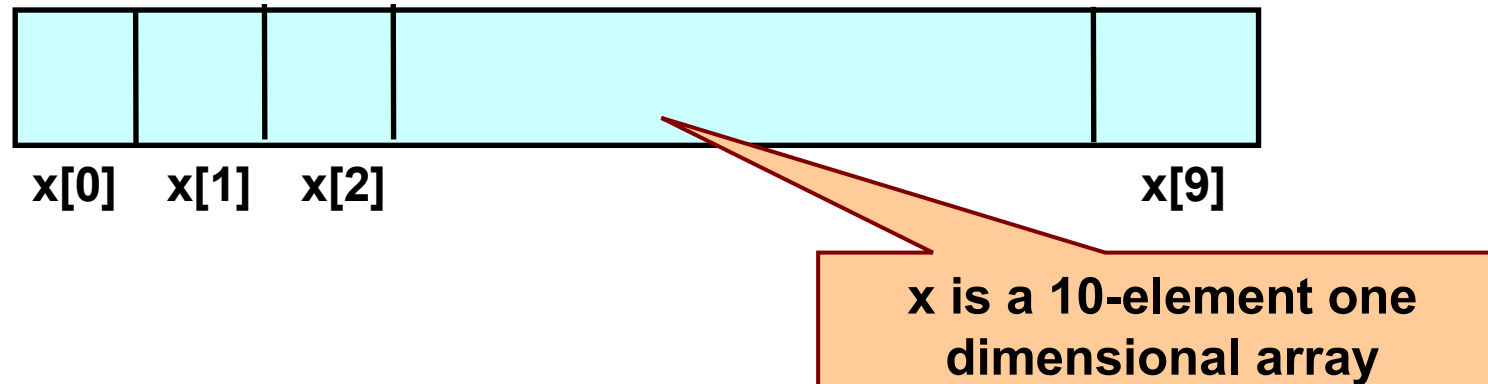
Practical solution: **array**

Using Arrays

All the data items constituting the group share the same name

```
int x[10]; // to store 10 integers
```

Individual elements are accessed by specifying the index



Declaring Arrays

Like variables, the arrays must be declared in a program before they are used.

General syntax:

```
type array-name [size];
```

- **type** specifies the type of elements that will be stored in the array (int, float, char, etc.)
 - **size** is the maximum number of elements that can be stored in the array
- Example: **marks** is an array that can store a maximum of 10 integers:

```
int marks[10];
```

Array Declarations: examples

Examples:

```
int id[10];
```

```
char name[20];
```

```
float marks[50];
```

If we are not sure of the exact size of the array that we will need, we can define an array of a large enough size:

```
float marks[50];
```

though in a particular run we may only be using, say, 10 elements.

Accessing Array Elements

A particular element of the array can be accessed by specifying two things:

- Name of the array
- Index (relative position) of the element in the array

Important to remember: In C, the **index of an array starts from 0, not 1**

Example:

- An array is defined as `int x[10];`
- The first element of the array `x` can be accessed as `x[0]`, i^{th} element as `x[i-1]`, tenth element as `x[9]`.

A First Example

Array size should be a constant

```
int main()
{
    int i;
    int data[10];
    for (i=0; i<10; i++)
        data[i]= i;
    for (i=0; i<10; i++)
        printf("Data[%d] = %d\n", i, data[i]);
    return 0;
}
```

“data” is an array of 10 integer variables:
data[0], data[1], ..., data[9]



```
Data[0] = 0
Data[1] = 1
Data[2] = 2
Data[3] = 3
Data[4] = 4
Data[5] = 5
Data[6] = 6
Data[7] = 7
Data[8] = 8
Data[9] = 9
```

Output

How is an array stored in memory?

Starting from a given memory location, the successive array elements are allocated space in consecutive memory locations



Array *a*

Let s = starting address of the array in memory

k = number of bytes allocated per element (e.g., 4 for each int, 1 for each char)

Then, the element $a[i]$ is allocated memory location at address $s + i * k$

A Special Operator: AddressOf (&)

Remember that each variable is stored at a memory location with a unique address.

Putting **&** before a variable name gives the starting address of the variable in the memory (where it is stored, not the value).

Can be put before any variable (with no blank in between)

```
int a = 10;  
printf("Value of a = %d, address of a = %d\n", a, &a);
```

Similarly, if we have an array: **int Data[10];**

Memory address of the first element is **&Data[0]**

Memory address of the second element is **&Data[1]**

Memory address of the third element is **&Data[2]**

Example

```
int main()
{
    int i;
    int Data[10];
    for (i=0; i<10; i++)
        printf("&Data[%d] = %u\n", i, &Data[i]);
    return 0;
}
```

Note: memory addresses are being printed as unsigned integers using **%u** in printf.
A better format is **%p** that prints the address in hexadecimal.

Typically, variables are allocated memory locations whose addresses are multiple of 4.

Output

```
&Data[0] = 3221224480
&Data[1] = 3221224484
&Data[2] = 3221224488
&Data[3] = 3221224492
&Data[4] = 3221224496
&Data[5] = 3221224500
&Data[6] = 3221224504
&Data[7] = 3221224508
&Data[8] = 3221224512
&Data[9] = 3221224516
```

How to read the elements of an array?

By reading one element at a time.

Suppose we have declared an array: `float a[25];`

```
for (j=0; j<25; j++)  
    scanf ("%f", &a[j]);
```

Note the ampersand (&) in scanf.

Reading into an array: example

```
int main() {  
    const int MAX_SIZE = 100;  
    int i, size;  
    float marks[MAX_SIZE];  
    float total;  
    scanf("%d", &size);  
    for (i=0, total=0; i<size; i++){  
        scanf("%f", &marks[i]);  
        total = total + marks[i];  
    }  
    printf("Total = %f \n Avg = %f\n",  
          total, total/size);  
    return 0;  
}
```

Input = a list of marks from the user.
Output = total and average.

Output

```
4  
2.5  
3.5  
4.5  
5  
Total = 15.500000  
Avg = 3.875000
```

Printing in Reverse Using Arrays

```
int main(){
    int n, A[100], i;
    printf ("How many numbers to read? ");
    scanf ("%d", &n);
    for (i=0; i < n; ++i)
        scanf ("%d", &A[i]);    // input the i-th array element
    for (i=n-1; i >= 0; --i)    // note: loop counts downward
        printf ("%d  ", A[i]); // output the i-th array element
    printf("\n");
    return 0;
}
```

Indexes into Arrays

The **array index** can be any **expression** that evaluates to an **integer** between **0** and **n-1** where **n** is the maximum number of elements possible in the array.

Examples:

```
a[x+2] = 25;
```

```
b[3*x-y] = a[10-x] + 5;
```

Remember:

Each **array element** is a **variable** in itself, and can be used anywhere a variable can be used (in expressions, assignments, conditions,...)

Initialization of Arrays

General form:

```
type array_name[size] = {comma-separated list of values};
```

Examples:

```
int marks[5] = {72, 83, 65, 80, 76};
```

```
char name[4] = {'A', 'm', 'i', 't'};
```

The **size** may be omitted if all initializers are specified. In such cases, the compiler automatically allocates enough space for all initialized elements:

```
int marks[] = {72, 83, 65, 80, 76};
```

```
char name[] = {'A', 'm', 'i', 't'};
```

A Warning

In C, while accessing array elements, **array bounds** are not checked.

Example:

```
int marks[5];
```

```
...
```

```
marks[0] = 87;
```

```
...
```

```
marks[8] = 75; // Caution: out of the array bounds!
```

- The last assignment may not give any compilation error. But it may result in unpredictable results during execution.

How to copy the elements of one array to another?

By copying individual elements:

```
int a[25], b[50];  
  
for (j=0; j<25; j++)  
    a[j] = b[2*j];
```

The element assignments will follow the rules of assignment expressions.

Destination array must have sufficient size.

Things you cannot do with arrays

You cannot:

- assign one array variable to another

```
a = b; /* a and b are arrays */
```

Indeed, a or b cannot be an l-value in any assignment.

- use == to compare arrays

```
if (a == b) { ..... } Doesn't make element-by-element comparison
```

- directly scanf or printf arrays (works, but not recommended unless purposefully made)

```
printf(".....", a);
```

```
scanf(".....", a);
```

Example: Find the minimum in an array of 10 numbers

```
int main() {  
    int a[10], i, min;  
  
    for (i=0; i<10; i++)  
        scanf("%d", &a[i]);  
  
    min = a[0];  
    for (i=1; i<10; i++) {  
        if (a[i] < min)  
            min = a[i];  
    }  
    printf("\nMinimum is %d", min);  
    return 0;  
}
```

Alternate Version 1

By #define:
You have to change
only one line of code
to change the problem size

```
#define size 10

int main() {
    int a[size], i, min;

    for (i=0; i<size; i++)
        scanf("%d", &a[i]);

    min = a[0];
    for (i=1; i<size; i++) {
        if (a[i] < min)
            min = a[i];
    }
    printf("Minimum is %d\n", min);
    return 0;
}
```

Alternate Version 2

Define an array of **large size** and use only the required number of elements

```
int main() {
    int  a[100], i, min, n;

    scanf ("%d", &n); //Number of elements
    for  (i=0; i<n; i++)
        scanf ("%d", &a[i]);

    min = a[0];
    for  (i=1; i<n; i++){
        if  (a[i] < min)
            min = a[i];
    }
    printf ("Minimum is %d\n", min);
    return 0;
}
```

Example: Computing Grade Point Average

Handling two
arrays
at the same time

cred[j] stores credit of subject **j**

grade_pt[j] stores grade point
obtained by a student in subject **j**

```
const int nsub = 6;

int main()
{
    int  grade_pt[nsub], cred[nsub], i, gp_sum=0, cred_sum=0;
    double gpa;

    for (i=0; i<nsub; i++)
        scanf ("%d %d", &grade_pt[i], &cred[i]);

    for (i=0; i<nsub; i++)
    {
        gp_sum += grade_pt[i] * cred[i];
        cred_sum += cred[i];
    }

    gpa = ((float) gp_sum) / cred_sum;
    printf ("Grade point average is %f\n", gpa);
    return 0;
}
```


Example: Find largest contiguous sequence of equal numbers

```
#include<stdio.h>
int main()
{
    int i, n, A[20], k, maxbegin, maxcount, ssbegin, count;
    scanf ("%d", &n);
    for (i=0; i<n; i++)    scanf ("%d", &A[i]);
    printf ("A = ");
    for (i=0; i<n; i++)    printf ("%d, ", A[i]); printf("\n");
    maxbegin = 0; maxcount = 1;
    ssbegin = 0; count = 1; k = 1;
    while (k < n) {
        if (A[k] == A[k-1]) {
            count++;
            if (count > maxcount) {
                maxbegin = ssbegin;
                maxcount = count;
            }
        } else {
            ssbegin = k; count = 1;
        }
        k++;
    }
    printf ("Sequence starting from A[%d] of Length = %d, Value = %d \n",
            maxbegin, maxcount, A[maxbegin]);
}
```

```
10
1 2 2 2 3 2 2 2 2 7
A = 1, 2, 2, 2, 3, 2, 2, 2, 2, 7,
Sequence starting from A[5] of Length = 4, Value = 2
```

Practice Problems

1. Read in an integer n ($n < 25$). Read n integers in an array A . Then do the following (write separate programs for each, only the reading part is common).
 - a) Find the sum of the absolute values of the integers.
 - b) Copy the positive and negative integers in the array into two additional arrays B and C respectively. Print A , B , and C .
 - c) Exchange the values of every pair of values from the start (so exchange $A[0]$ and $A[1]$, $A[2]$ and $A[3]$ and so on). If the number of elements is odd, the last value should stay the same.

2. Read in two integers n and m ($n, m < 50$). Read n integers in an array A . Read m integers in an array B . Then do the following (write separate programs for each part, only the reading part is common).
 - a) Find if there are any two elements x, y in A and an element z in B , such that $x + y = z$
 - b) Copy in another array C all elements that are in both A and B (intersection)
 - c) Copy in another array C all elements that are in either A and B (union)
 - d) Copy in another array C all elements that are in A but not in B (difference)