1-d Arrays
Array

- Many applications require multiple data items that have common characteristics
  - In mathematics, we often express such groups of data items in indexed form:
    - $x_1, x_2, x_3, \ldots, x_n$

- Array is a data structure which can represent a collection of data items which have the same data type (float/int/char/…).
Example: Printing Numbers in Reverse

3 numbers

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

4 numbers

```c
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
", a);
```
The Problem

- Suppose we have 10 numbers to handle
- Or 20
- Or 100
- Where do we store the numbers? Use 100 variables??
- How to tackle this problem?
- Solution:
  - Use arrays
Printing in Reverse Using Arrays

```c
void 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]);
    for (i = n - 1; i >= 0; --i)
        printf("%d  ", A[i]);
    printf("\n");
}
```
Using Arrays

- All the data items constituting the group share the same name
  ```
  int x[10];
  ```
- Individual elements are accessed by specifying the index

```
x[0]  x[1]  x[2]
```

X is a 10-element one dimensional array
A first example

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

“data refers to a block of 10 integer variables, data[0], data[1], …, data[9]"
The result

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

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

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

- **Type** specifies the type of element that will be contained in the array (int, float, char, etc.).
- **Size** is an integer constant which indicates the maximum number of elements that can be stored inside the array.

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

- **Marks** is an array that can store a maximum of 5 integers.
Examples:

```c
int x[10];
char line[80];
float points[150];
char name[35];
```

If we are not sure of the exact size of the array, we can define an array of a large size

```c
int 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
- In C, the index of an array starts from zero
- Example:
  - An array is defined as `int x[10];`
  - The first element of the array x can be accessed as `x[0]`, fourth element as `x[3]`, tenth element as `x[9]`, etc.
Contd.

- The array index must evaluate to an integer between 0 and n-1 where n is the maximum number of elements possible in the array
  
  \[ a[x+2] = 25; \]
  
  \[ b[3*x-y] = a[10-x] + 5; \]

- Remember that each array element is a variable in itself, and can be used anywhere a variable can be used (in expressions, assignments, conditions,...)
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

- $x$: starting address of the array in memory
- $k$: number of bytes allocated per array element

- $a[i] \rightarrow$ is allocated memory location at address $x + i*k$
void main()
{
    int i;
    int data[10];
    for(i=0; i<10; i++)
        printf("&Data[%d] = %u\n", i, &data[i]);
}

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
Initialization of Arrays

- General form:

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

- Examples:

  ```
  int   marks[5] = {72, 83, 65, 80, 76};
  char  name[4] = {'A', 'm', 'i', 't'};
  ```

- The size may be omitted. In such cases the compiler automatically allocates enough space for all initialized elements.

  ```
  int   flag[ ] = {1, 1, 1, 0};
  char  name[ ] = {'A', 'm', 'i', 't'};
  ```
How to read the elements of an array?

- By reading them one element at a time

```c
for (j=0; j<25; j++)
    scanf ("%f", &a[j]);
```
- The ampersand (&) is necessary
- The elements can be entered all in one line or in different lines
A Warning

- In C, while accessing array elements, array bounds are not checked

Example:

```c
int marks[5];
:
:
:
marks[8] = 75;
```

- The above assignment would not necessarily cause an error
- Rather, it may result in unpredictable program results
Reading into an array

```c
void 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);
}
```

Output

```
4
2.5
3.5
4.5
5
Total = 15.500000
Avg = 3.875000
```
How to print the elements of an array?

- By printing them one element at a time
  
  ```c
  for (j=0; j<25; j++)
      printf ("\n %f", a[j]);
  ```
  
  The elements are printed one per line
  ```c
  printf ("\n");
  ```
  
  ```c
  for (j=0; j<25; j++)
      printf (" %f", a[j]);
  ```
  
  The elements are printed all in one line (starting with a new line)
How to copy the elements of one array to another?

- By copying individual elements
  
  ```c
  for (j=0; j<25; j++)
    a[j] = b[j];
  ```

- The element assignments will follow the rules of assignment expressions

- Destination array must have sufficient size
Example 1: Find the minimum of a set of 10 numbers

```c
void 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 ("\n Minimum is %d", min);
}
```
Alternate Version 1

```c
const int size = 10;

void 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 ("\n Minimum is %d", min);
}
```

Change only one line to change the problem size
Alternate Version 2

```c
#define size 10

void 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 ("n Minimum is %d");
}
```

Change only one line to change the problem size

Used #define macro
#define macro

- `#define X Y`
- Preprocessor directive
- Compiler will first replace all occurrences of string X with string Y in the program, then compile the program
- Similar effect as read-only variables (`const`), but no storage allocated
- We prefer you use `const` instead of `#define`
Alternate Version 3

void 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 ("\n Minimum is %d", min);
}
Example 2: Computing cgpa

```c
const int nsub = 6;

void 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 ("\n Grade point average: is %.2lf", gpa);
}
```

Handling two arrays at the same time
Example: Binary Search

- Searching for an element $k$ in a sorted array $A$ with $n$ elements

- Idea:
  - Choose the middle element $A[n/2]$
  - If $k == A[n/2]$, we are done
  - If $k < A[n/2]$, search for $k$ between $A[0]$ and $A[n/2 - 1]$
  - If $k > A[n/2]$, search for $k$ between $A[n/2 + 1]$ and $A[n-1]$
  - Repeat until either $k$ is found, or no more elements to search

- Requires less number of comparisons than linear search in the worst case ($\log_2 n$ instead of $n$)
```c
void main() {
    int A[100], n, k, i, mid, low, high;
    scanf("%d %d", &n, &k);
    for (i=0; i<n; ++i) scanf("%d", &A[i]);
    low = 0; high = n - 1; mid = low + (high - low)/2;
    while (high >= low) {
        printf("low = %d, high = %d, mid = %d, A[%d] = %d\n", low, high, mid, mid, A[mid]);
        if (A[mid] == k) {
            printf("%d is found\n", k);
            break;
        }
        if (k < A[mid]) high = mid - 1;
        else low = mid + 1;
        mid = low + (high - low)/2;
    }
    if (high < low) printf("%d is not found\n", k);
}
```
Output

8  21
9  11  14  17  19  20  23  27
low = 0, high = 7, mid = 3, A[3] = 17
21 is not found

8  14
9  11  14  17  19  20  23  27
low = 0, high = 7, mid = 3, A[3] = 17
low = 0, high = 2, mid = 1, A[1] = 11
low = 2, high = 2, mid = 2, A[2] = 14
14 is found
Example: Selection Sort

- Sort the elements of an array $A$ with $n$ elements in ascending order

- Basic Idea:
  - Find the min of the $n$ elements, swap it with $A[0]$ (so min is at $A[0]$ now)
  - Now find the min of the remaining $n-1$ elements, swap it with $A[1]$ (so $2^{nd}$ min is at $A[1]$ now)
  - Continue until no more elements left
void main() {
    int A[100], n, i, j, k, min, pos, temp;
    scanf("%d", &n);
    for (i=0; i<n; ++i) scanf("%d", &A[i]);
    for (i = 0; i < n - 1; ++i) {
        min = A[i]; pos = i;
        for (j = i + 1; j < n; ++j) {
            if (A[j] < min) {
                min = A[j];
                pos = j;
            }
        }
        temp = A[i];
        A[i] = A[pos];
        A[pos] = temp;
    }
    for (k=0; k<n; ++k) printf("%d  ", A[k]);
    printf("\n");
}
### Output

<table>
<thead>
<tr>
<th></th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Things you cannot do

- You cannot
  - use = to assign one array variable to another
    ```
    a = b; /* a and b are arrays */
    ```
  - use == to directly compare array variables
    ```
    if (a == b) ...........
    ```
  - directly scanf or printf arrays
    ```
    printf (“……..”, a);
    ```
Character Arrays and Strings

```c
char C[8] = { 'a', 'b', 'h', 'i', 'j', 'i', 't', '\0' };  
```

- C[0] gets the value 'a', C[1] the value 'b', and so on. The last (7th) location receives the null character ‘\0’
- Null-terminated (last character is ‘\0’) character arrays are also called strings
- Strings can be initialized in an alternative way. The last declaration is equivalent to:

```c
char C[8] = "abhijit";  
```

- The trailing null character is missing here. C automatically puts it at the end if you define it like this
- Note also that for individual characters, C uses single quotes, whereas for strings, it uses double quotes
Reading strings: %s format

```c
void main()
{
    char name[25];
    scanf("%s", name);
    printf("Name = %s \n", name);
}
```

%s reads a string into a character array given the array name or start address. It ends the string with ‘\0’
void main()
{
#define SIZE 25
int i, count=0;
char name[SIZE];
scanf("%s", name);
printf("Name = %s \n", name);
for (i=0; name[i]!="\0"; i++)
    if (name[i] == 'a') count++;
printf("Total a's = %d \n", count);
}
void main()
{
    const int SIZE = 25;
    int i, flag, count=0;
    char name[SIZE];
    scanf("%s", name);    /* Read Name */
    for (i=0; name[i]!='\0'; i++); /* Find Length of String */
    printf("Total length = %d\n",i);
    count=i;  flag = 0;
    /* Loop below checks for palindrome by comparison*/
    for(i=0; i<count; i++) if (name[i]!=name[count-i-1]) flag = 1;
    if (flag ==0) printf("%s is a Palindrome\n", name);
    else printf("%s is NOT a Palindrome\n", name);
}
Some Exercises

1. Write a C program that reads an integer n and stores the first n Fibonacci numbers in an array.
2. Write a C program that reads an integer n and uses an array to efficiently find out the first n prime numbers.
3. Read in an integer n, read in n integers and print the integer with the highest frequency.
4. Read in an integer n, read in n numbers and find out the mean, median and mode.
5. Read in two names and compare them and print them in lexicographic (dictionary) order.
6. Read in an integer n, read in n names and print the last name when compared in lexicographic order.
2-d Arrays
Two Dimensional Arrays

- We have seen that an array variable can store a list of values
- Many applications require us to store a table of values

<table>
<thead>
<tr>
<th></th>
<th>Subject 1</th>
<th>Subject 2</th>
<th>Subject 3</th>
<th>Subject 4</th>
<th>Subject 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student 1</td>
<td>75</td>
<td>82</td>
<td>90</td>
<td>65</td>
<td>76</td>
</tr>
<tr>
<td>Student 2</td>
<td>68</td>
<td>75</td>
<td>80</td>
<td>70</td>
<td>72</td>
</tr>
<tr>
<td>Student 3</td>
<td>88</td>
<td>74</td>
<td>85</td>
<td>76</td>
<td>80</td>
</tr>
<tr>
<td>Student 4</td>
<td>50</td>
<td>65</td>
<td>68</td>
<td>40</td>
<td>70</td>
</tr>
</tbody>
</table>
The table contains a total of 20 values, five in each line

- The table can be regarded as a matrix consisting of four rows and five columns

- C allows us to define such tables of items by using two-dimensional arrays
Declaring 2-D Arrays

- General form:
  ```
  type   array_name [row_size][column_size];
  ```

- Examples:
  ```
  int  marks[4][5];
  float sales[12][25];
  double matrix[100][100];
  ```
Initializing 2-d arrays

- \texttt{int a[2][3] = \{1,2,3,4,5,6\};}
- \texttt{int a[2][3] = {{1,2,3}, \{4,5,6\}};
- \texttt{int a[][][3] = {{1,2,3}, \{4,5,6\}};

All of the above will give the 2x3 array

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{array}
\]
Accessing Elements of a 2-d Array

- Similar to that for 1-d array, but use two indices
  - First indicates row, second indicates column
  - Both the indices should be expressions which evaluate to integer values (within range of the sizes mentioned in the array declaration)

- Examples:
  
  ```
  x[m][n] = 0;
  c[i][k] += a[i][j] * b[j][k];
  a = sqrt (a[j*3][k]);
  ```
Example

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

A two-dimensional array of 15 elements
Can be looked upon as a table of 3 rows and 5 columns

<table>
<thead>
<tr>
<th></th>
<th>col0</th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>col4</th>
</tr>
</thead>
<tbody>
<tr>
<td>row0</td>
<td>a[0][0]</td>
<td>a[0][1]</td>
<td>a[0][2]</td>
<td>a[0][3]</td>
<td>a[0][4]</td>
</tr>
<tr>
<td>row1</td>
<td>a[1][0]</td>
<td>a[1][1]</td>
<td>a[1][2]</td>
<td>a[1][3]</td>
<td>a[1][4]</td>
</tr>
</tbody>
</table>
How is a 2-d array is stored in memory?

- Starting from a given memory location, the elements are stored **row-wise** in consecutive memory locations (**row-major** order)
  - x: starting address of the array in memory
  - c: number of columns
  - k: number of bytes allocated per array element

- $a[i][j]$ is allocated memory location at address $x + (i \times c + j) \times k$

```
Row 0    Row 1    Row 2
a[0][0] a[0][1] a[0][2] a[0][3] a[1][0] a[1][1] a[1][2] a[1][3] a[2][0] a[2][1] a[2][2] a[2][3]
```
int main()
{
    int a[3][5];
    int i, j;

    for (i=0; i<3; i++)
    {
        for (j=0; j<5; j++) printf("%u\n", &a[i][j]);
        printf("\n");
    }

    return 0;
}
More on Array Addresses

```c
int main()
{
    int a[3][5];
    printf("a = %u\n", a);
    printf("&a[0][0] = %u\n", &a[0][0]);
    printf("&a[2][3] = %u\n", &a[2][3]);
    printf("a[2]+3 = %u\n", a[2]+3);
    printf("*(a+2)+3 = %u\n", *(a+2)+3);
    printf("*(a+2) = %u\n", *(a+2));
    printf("a[2] = %u\n", a[2]);
    printf("&a[2][0] = %u\n", &a[2][0]);
    printf("(a+2) = %u\n", (a+2));
    printf("&a[2] = %u\n", &a[2]);
    return 0;
}
```

Output:

```
a = 3221224480
&a[0][0] = 3221224480
&a[2][3] = 3221224532
a[2]+3 = 3221224532
*(a+2)+3 = 3221224532
*(a+2) = 3221224520
a[2] = 3221224520
&a[2][0] = 3221224520
(a+2) = 3221224520
&a[2] = 3221224520
```
How to read the elements of a 2-d array?

- By reading them one element at a time
  
  ```
  for (i=0; i<nrow; i++)
    for (j=0; j<ncol; j++)
      scanf ("%f", &a[i][j]);
  ```

- The ampersand (&) is necessary
- The elements can be entered all in one line or in different lines
How to print the elements of a 2-d array?

- By printing them one element at a time
  
  ```c
  for (i=0; i<nrow; i++)
    for (j=0; j<ncol; j++)
      printf ("\n %f", a[i][j]);
  
  □ The elements are printed one per line
  ```

  ```c
  for (i=0; i<nrow; i++)
    for (j=0; j<ncol; j++)
      printf ("%f", a[i][j]);
  
  □ The elements are all printed on the same line
  ```
for (i=0; i<nrow; i++)
{
    printf ("
");
    for (j=0; j<ncol; j++)
    {
        printf ("%f ", a[i][j]);
    }
}

The elements are printed nicely in matrix form
Example: Matrix Addition

```c
int main()
{
    int a[100][100], b[100][100],
        c[100][100], p, q, m, n;

    scanf ("%d %d", &m, &n);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &a[p][q]);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            scanf ("%d", &b[p][q]);

    for (p=0; p<m; p++)
        for (q=0; q<n; q++)
            c[p][q] = a[p][q] + b[p][q];

    for (p=0; p<m; p++)
    {
        printf ("n");
        for (q=0; q<n; q++)
            printf ("%d   ", c[p][q]);
    }

    return 0;
}
```