Definition: Arrays

- A collection of elements of the same type stored contiguously in memory under one name
  - can be of any data type, e.g., integer, long integer, float, double, character etc.
  - even collection of arrays!
  - Arrays of structure, union, pointer etc. are also allowed

Advantages:
- For ease of access to any element of an array
- Passing a group of elements to a function
Array Representation

- A sample one-dimensional integer array

**Conceptual Picture**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

- A collection of integer type elements
- Each element is associated with a location index
- In C, array index starts from zero

**Actual Picture**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>1002</td>
<td>5</td>
</tr>
<tr>
<td>1004</td>
<td>1</td>
</tr>
<tr>
<td>1006</td>
<td>7</td>
</tr>
<tr>
<td>1008</td>
<td>3</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
</tr>
</tbody>
</table>
Arrays: Declaration & Initialization

■ Declaration: \texttt{int A[6];}
  - An array of 6 integers
  - \texttt{A[0], A[1], A[2], ..., A[6]}

■ If array is declared within a function it contains garbage, if not initialized

■ If array is globally declared it contains zeros

■ Initialization:
  \texttt{int A[6] = \{2,5,1,7,3,10\};}
  - First index is 0, and Last index is array size - 1

■ Accessing array element at index \texttt{i}: \texttt{A[i]}
Arrays: Characteristics

- The storage class of arrays may be automatic, external, or static, but not register.
- If external or static arrays are not initialized, they are by default initialized to zero.
- If an array is declared without a size and is initialized to a series of values, it implicitly given the size of the number of initializers.

```c
int A[] = {2, 5, 1, 7, 3, 10};
```

size of array A is 6 here.
Arrays: Characteristics (cont.)

- Character arrays:
  ```
  char c[] = {'a', 'b', 'c', '\0'};
  ```

- Alternatively:
  ```
  char c[] = "abc";
  ```

- These two representations are equivalent

- **string** is a sequence of **characters** that is treated as a single data item and terminated by **null character** '\0'. **C** does not support **strings** as a data type. A **string** is actually one-dimensional **array** of **characters** in **C**.
Array Usage: Example

- Sum all the elements of an array

```c
#include <stdio.h>

int main(void) {
    int a[10] = {1,2,3,4,5,6,7,8,9,10};

    int i, sum = 0;

    for(i = 0; i < 10; i++)
    {
        sum += a[i];
    }

    printf("%d\n", sum);

    return 0;
}
```
Errors in array usage

1. If $i$ has a value outside the range $[0, \text{size}-1]$, no compiler error. Run-time error will occur when $A[i]$ is accessed.
   - Overrunning the bounds of an array is a common programming error
   - The effect of the error is system-dependent
   - Often the value of some unrelated variable will be returned

2. If local array is used before initialization garbage value will be processed
2-dimensional array

- A 2D 3-by-3 integer array
  - 2D square array
  - not always necessary to have equal number of columns and rows

- Declaration: `int A[3][3];`

- Initialization: `int A[3][3] = {{2, 5, 1}, {7, 3, 10}, {0, 1, 6}};`

- Applications:
  - Matrix representation, e.g., graph adjacency matrix
2D Array for Graph Adjacency Matrix

```c
int A[6][6] = 
    {{0,1,1,0,0,0},
    {1,0,0,1,0,0},
    {1,0,0,1,1,0},
    {0,1,1,0,0,1},
    {0,0,1,0,0,1},
    {0,0,0,1,1,0}};
```

Undirected unweighted plain graph

Shebuti Rayana (CS, Stony Brook University)
2D Arrays in Memory

- In the computer memory, all elements are stored linearly using contiguous addresses.
- In order to store a two-dimensional matrix, two dimensional address space must be mapped to one-dimensional address space.
- In the computer's memory matrices are stored in either Row-major order or Column-major order form.
### 2D Arrays in Memory (cont.)

#### Conceptual Picture

<table>
<thead>
<tr>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Actual Picture

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>1002</td>
<td>5</td>
<td>(0, 1)</td>
</tr>
<tr>
<td>1004</td>
<td>1</td>
<td>(0, 2)</td>
</tr>
<tr>
<td>1006</td>
<td>7</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>1008</td>
<td>3</td>
<td>(1, 1)</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>1012</td>
<td>0</td>
<td>(2, 0)</td>
</tr>
<tr>
<td>1014</td>
<td>1</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>1016</td>
<td>6</td>
<td>(2, 2)</td>
</tr>
</tbody>
</table>

**Row Major Order**

Example is given for row major order only.
2D Array Usage: Example

- Matrix multiplication code for matrix $a$ and $b$

```c
int i, j, k;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        double sum = 0;
        for (k = 0; k < n; k++) {
            sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}
```
Pointers
Introduction

- A variable in a program is stored in a certain number of bytes at a particular memory location or address.
- **Pointers** are used to access memory and manipulate address.
- If v is a variable, then \&v gives its memory address
  - Address operator \& is an unary operator
**Pointers: Declaration**

- **Example Declaration:** `int *p;`
  - `p` is a pointer to integer
  - The indirection or dereferencing operator `*` is unary

- Its range of values include a special address `0` and a set of positive integers that represent machine addresses.

- **Example assignment to pointer `p`**
  ```c
  p = 0;
p = Null; // same as p = 0
p = &i; // pointing to i
p = (int *)1776; /* absolute address */
  ```

Shebuti Rayana (CS, Stony Brook University)
Pointers: Characteristics

- If \( p \) is a pointer then \( \star p \) is the value of the variable of which \( p \) is the address.

- Direct value of \( p \) is an address of a memory location, and \( \star p \) is indirect value of \( p \), which is the value stored in that memory location.

- In a certain sense \( \star \) is the inverse operator of \&
Pointers: Example

- `int a = 1, b = 2, *p;`
- Think of the pointer as an arrow, but it is not yet assigned a value. So, we do not know what it points to.
- **Next line:** `p = &a`
- `b = *p;  b = ?`
Pointers: Example Code

```c
#include <stdio.h>

int main(void)
{
    int i = 7, *p = &i;

    printf("Value of i: ", *p, "Location of i: ", p);
    return 0;
}
```

Value of i: 7
Location of i: effffb24

- A pointer can be initialized in a declaration.
  - The variable `p` is of type `int` and its initial value is `&i`.
  - The declaration of `i` must occur before we take its address.
Pointers: Declaration and Initialization

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>p == &amp;i</td>
<td>p == (&amp;i)</td>
<td>1</td>
</tr>
<tr>
<td>**&amp;p</td>
<td><em>(</em>(&amp;p))</td>
<td>3</td>
</tr>
<tr>
<td>r = &amp;x</td>
<td>r = (&amp;x)</td>
<td>illegal</td>
</tr>
<tr>
<td>7* *p/ *q+7</td>
<td>((7*(p))/(*q)) +7</td>
<td>11</td>
</tr>
<tr>
<td>*(r=&amp;j) *= *p</td>
<td>(*(r = (&amp;j))) *= (*p)</td>
<td>15</td>
</tr>
</tbody>
</table>
Constructs not to be pointed at

■ Do not point at constants.
- &3 /* illegal */

■ Do not point at ordinary expressions.
- &(k + 99) /* illegal */

■ Do not point at register variables.
- register v;
- &v /* illegal */

■ Address operator can be applied to variables and array elements.
- If a is an array, expressions such as &a[0] and &a[i+j+3] make sense.
Call-by-reference

- "call-by-reference" is a way of passing addresses (references) of variables to a function that then allows the body of the function to make changes to the values of variables in the calling environment.

```c
#include <stdio.h>

void swap(int i, int j)
{
    int temp;
    temp = i;
    i = j;
    j = temp;
}

int main()
{
    int i = 5;
    int j = 10;
    swap(i, j);
    printf("i = %d\n", i);
    printf("j = %d\n", j);
}
```

Output:

```
i = 5
j = 10
```

```c
#include <stdio.h>

void swap(int *i, int *j)
{
    int temp;
    temp = *i;
    *i = *j;
    *j = temp;
}

int main()
{
    int i = 5;
    int j = 10;
    swap(&i, &j);
    printf("i = %d\n", i);
    printf("j = %d\n", j);
}
```

Output:

```
i = 10
j = 5
```
Relationship between Arrays and Pointers

- A pointer variable can take different addresses as values. In contrast, an array name is an address, or pointer, that is fixed. So following are illegal:

\[
\begin{align*}
a &= p \\
++a \\
a &+= 2
\end{align*}
\]

- Suppose \(a\) is an array and \(i\) is an \(\text{int}\),
  - \(a[i]\) is equivalent to \(*(a+i)\)

- Equivalent expressions:

```c
#define N 100
int a[N], i, *p, sum = 0;
p = a       equivalent to \(p = &a[0]\)
p = a + 1  equivalent to \(p = &a[1]\)
```
Relationship between Arrays and Pointers

- Following 3 `for` loops are equivalent:

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

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

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