

CSE 230
Intermediate Programming
in C and C++
Arrays and Pointers

Fall 2017

Stony Brook University

Instructor: Shebuti Rayana

<http://www3.cs.stonybrook.edu/~cse230/>

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

2	5	1	7	3	10
[0]	[1]	[2]	[3]	[4]	[5]

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

Actual Picture

Memory Address	content
1000	2
1002	5
1004	1
1006	7
1008	3
1010	10

Arrays: Declaration & Initialization

- **Declaration:** `int A[6];`

- An array of 6 integers

- `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:**

- `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 `i`: `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.
`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' };
                ↑
```

Null character, represents end of string

- 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

```
#include <stdio.h>

int main(void)
{

    int n = 10;
    int A[n] = {1,2,3,4,5,6,7,8,9,10};
    int i, sum = 0;

    for(i = 0; i < n; i++)
    {
        sum += A[i];
    }

    printf("%d", sum);
    return 0;
}
```

Errors in array usage

1. If i has a value outside the range $[0, \text{size}-1]$, a 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

rows [0] [1] [2] ← columns

[0]	2	5	1
[1]	7	3	10
[2]	0	1	6

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

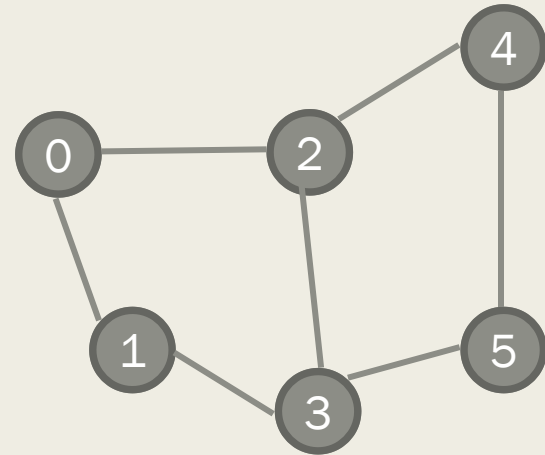
- Initialization: `int A[3][3] = {{2, 5, 1}, {7, 3, 10}, {0, 1, 6}};`

- Applications:

- Matrix representation, e.g, graph adjacency matrix

2D Array for Graph Adjacency Matrix

```
■ 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

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

	[0]	[1]	[2]
[0]	2	5	1
[1]	7	3	10
[2]	0	1	6

Actual Picture

Address	Content	Index
1000	2	(0, 0)
1002	5	(0, 1)
1004	1	(0, 2)
1006	7	(1, 0)
1008	3	(1, 1)
1010	10	(1, 2)
1012	0	(2, 0)
1014	1	(2, 1)
1016	6	(2, 2)

Row Major Order

Example is given for row major order only

2D Array Usage: Example

- Matrix multiplication code for matrix a and b

```
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`

```
p = 0;
```

```
p = Null; // same as p = 0
```

```
p = &i; // pointing to i
```

```
p = (int *)1776; /* absolute address  
*/
```


Pointers: Characteristics

- If p is a pointer then $*p$ is the value of the variable of which p is the address.
- Direct value of p is an address of a memory location, and $*p$ is indirect value of p , which is the value stored in that memory location.
- In a certain sense $*$ is the inverse operator of $\&$

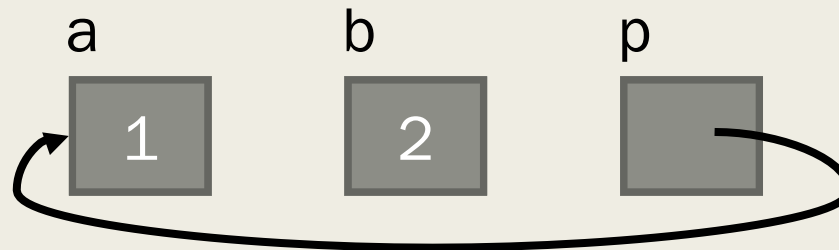
Pointers: Example

■ `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

```
#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

- 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

Declaration and Initialization

```
int i=3, j=5, *p=&i, *q=&j, *r;  
double x;
```

Expression	Equivalent Expression	Value
<code>p == &i</code>	<code>p == (&i)</code>	1
<code>**&p</code>	<code>* (* (&p))</code>	3
<code>r = &x</code>	<code>r = (&x)</code>	illegal
<code>7* *p/ *q+7</code>	<code>((7* (*p)) / (*q)) + 7</code>	11
<code>*(r=&j) *= *p</code>	<code>(* (r = (&j))) *= (*p)</code>	15

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.

Call by value

```
1  #include <stdio.h>
2
3  void swap(int i, int j)
4  {
5      int temp;
6      temp = i;
7      i = j;
8      j = temp;
9  }
10
11 int main()
12 {
13     int i = 5;
14     int j = 10;
15     swap(i,j);
16     printf("i = %d\n",i);
17     printf("j = %d\n",j);
18 }
```

Output:
i = 5
j = 10

```
1  #include <stdio.h>
2
3  void swap(int *i, int *j)
4  {
5      int temp;
6      temp = *i;
7      *i = *j;
8      *j = temp;
9  }
10
11 int main()
12 {
13     int i = 5;
14     int j = 10;
15     swap(&i,&j);
16     printf("i = %d\n",i);
17     printf("j = %d\n",j);
18 }
```

Output:
i = 10
j = 5

Call by reference

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:

`a = p` `++a` `a += 2` `&a`

- Suppose `a` is an array and `i` is an `int`,
 - `a[i]` is equivalent to `*(a+i)`

- Equivalent expressions:

```
#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:

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

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

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