Pointers and Arrays

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

Introduction

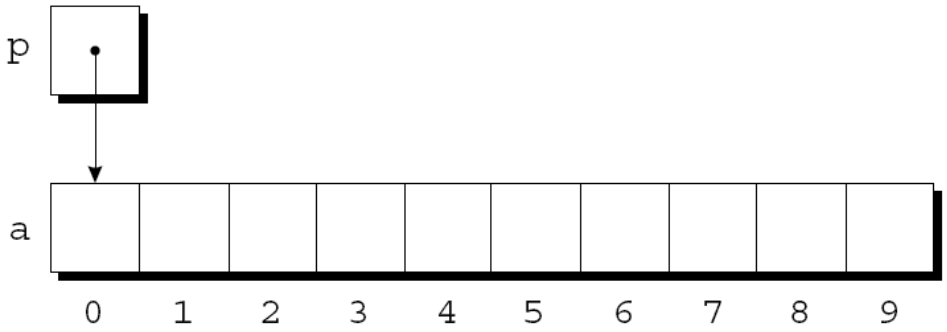
- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- The relationship between pointers and arrays in C is a close one.
- **Understanding this relationship is critical for mastering C**.

Pointer Arithmetic

- Pointers can point to array elements:

```
int a[10], *p;
p = &a[0];
```

- A graphical representation:

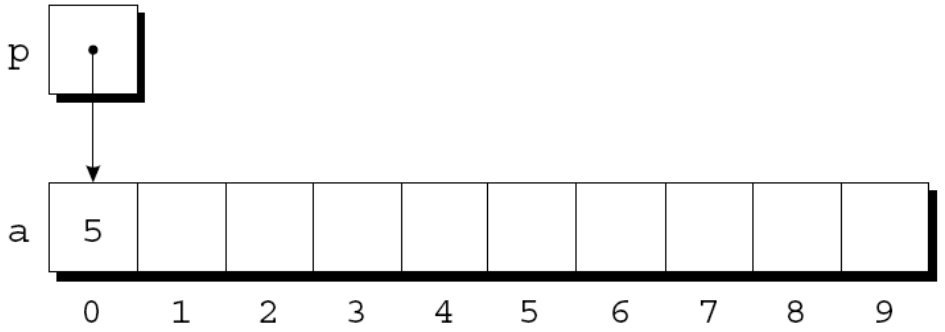


Pointer Arithmetic

- We can now access **`a[0]`** through **`p`**; for example, we can store the value 5 in **`a[0]`** by writing

```
*p = 5;
```

- An updated picture:



Pointer Arithmetic

- If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing ***pointer arithmetic*** (or ***address arithmetic***) on `p`.
- C supports three (and only three) forms of pointer arithmetic:
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another

Adding an Integer to a Pointer

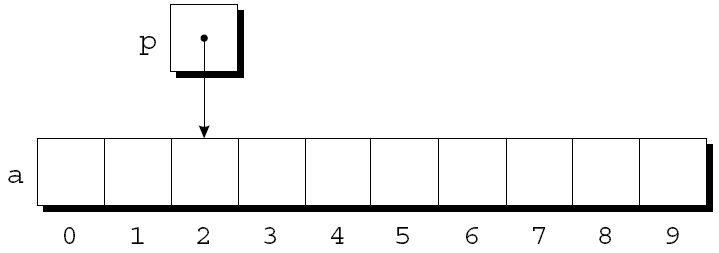
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
- Assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

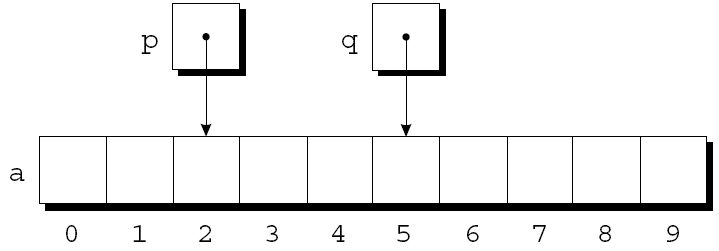
Adding an Integer to a Pointer

- Example of pointer addition:

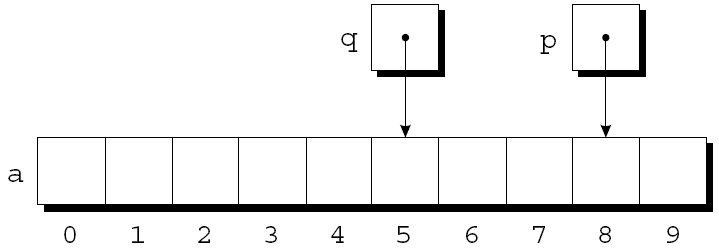
```
p = &a[2];
```




```
q = p + 3;
```




```
p += 6;
```

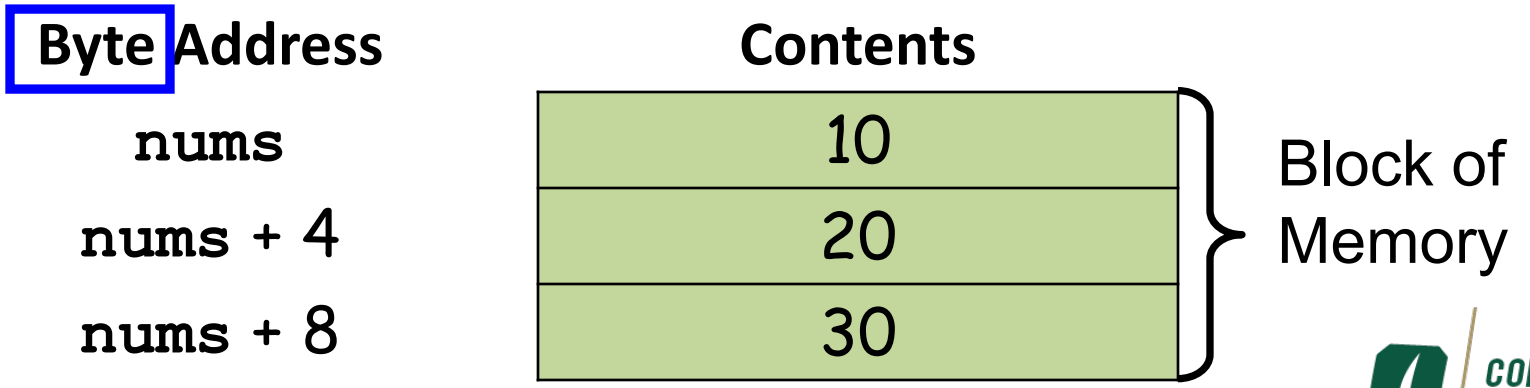


Arrays and Pointers

- An array variable declaration is really two things:
  1. **allocation** (and initialization) of a block **of memory** large enough to store the array
  2. binding of a **symbolic name** to the **address** of the **start** of the array

Example:

```
int nums[3] = { 10, 20, 30 };
```



Ways to Denote Array Addresses

- Address of first element of the array
  - `nums` (or `nums+0`), or
  - `&nums[0]`
- Address of second element
  - `nums+1`
  - `&nums[1]`
- etc.

Address of operator not needed since array name is really a pointer to (address of) the first element.

What happened to the "address of" operator?

The array knows its element size, so you want to only move one element and it doesn't tie your program to a specific type.

+4 would refer to bytes, which is not the unit we want to use

Why "+1" and not "+4"?

Arrays as Function Arguments

- Reminder: an **array** is passed by reference, as an address of (**pointer to**) the first element
- The following are **equivalent**

```
int len, slen ( char s[] );
char str[20] = "a string";
len = slen(str);
…
int slen(char str[])
{
   int len = 0;
   while (str[len] != '\0')
      len++;
   return len;
}
```

With arrays

```
int len, slen ( char *s );
char str[20] = "a string";
len = slen(str);
…
int slen(char *str)
{
   char *strend = str;
   while (*strend != '\0')
      strend++;
   return (strend - str);
}
```

With pointers

Arrays **are** Pointers

- Example: adding together elements of an array
- Version 0, with array **indexing**:

```
int i, nums[3] = {10, 20, 30};
int sum = 0;
for (i = 0; i < 3; i++)
    sum += nums[i];
```

Arrays **are** Pointers(cont'd)

Same example, using **pointers** (version 1)

pointer to int

increment pointer to next element in array **(pointer arithmetic)**

```
int *ap, nums[3] = {10, 20, 30};

int sum = 0;
for (ap = &(nums[0]); ap < &(nums[3]); ap++)
    sum += *ap;
```

add next element to sum

initialize pointer to starting address of array

loop until you exceed the bounds of the array

(see `array_summation.c` in *Code Samples and Demonstrations* in *Canvas*)

Arrays **are** Pointers (cont'd)

Using **pointers** in normal way (version 2)

```
for (ap = nums; ap < (nums+3); ap++)
    sum += *ap;
```

initialize pointer to starting address of array

loop until you exceed the bounds of the array - **more pointer arithmetic**

But **don't** try to do this

```
for ( ap = (nums+3); nums < ap; nums++)
    sum += *nums;
```

(see `array_summation2.c` in *Code samples and Demonstrations* in *Canvas*).

Pointer Arithmetic

Question: How **much** is the increment?

Adds **4** to the address

```
int *ap, nums[3] = {10, 20, 30};
int sum = 0;
for (ap = nums; ap <= (nums+2); ap++)
    sum += *ap;
```

Adds **1** to the address

```
char *ap, nums[3] = {10, 20, 30};
char sum = 0;
for (ap = nums; ap <= (nums+2); ap++)
    sum += *ap;
```

Answer: the **size of one element** of the array (e.g., 4 bytes for an **`int`**, 1 byte for a **`char`**, 8 bytes for a **`double`**, …)

(see **`array_iteration.c`** in *Code samples and Demonstrations* in *Canvas*).

…Arithmetic (cont'd)

Array of `ints`

Symbolic Address	Byte Addr	Contents
`nums`	Start of `nums`	10
`nums+1`	Start of `nums + 4`	20
`nums+2`	Start of `nums + 8`	30

Array of `chars`

Symbolic Address	Byte Addr	Contents
`nums`	Start of `nums`	10
`nums+1`	Start of `nums + 1`	20
`nums+2`	Start of `nums + 2`	30

…Arithmetic (cont'd)

Referencing the **ith element** of an array

```
int nums[10] = {…};
…
nums[i-1] = 50;
```

```
int nums[10] = {…};
…
*(nums + i - 1) = 50;
```

Equivalent

`nums` points to the beginning of our array. To get to element `i`, we add `i`, then subtract 1 to move left.

Referencing the **end** of an array

```
int *np, nums[10] = {…};
…
for (np = nums; np < (nums+10); np++ )
    …
```

The end of the array is at the address of what would be the 11th element.

Processing the Rows of a Multidimensional Array

- A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.
- To visit the elements of row `i`, we'd initialize `p` to point to element `0` in row `i` in the array `a`:

```
p = &a[i][0];
```

or we could simply write

```
p = a[i];
```

Processing the Rows of a Multidimensional Array

- For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first element in row `i`.
- To see why this works, recall that `a[i]` is equivalent to `*(a + i)`
- Thus, `&a[i][0]` is the same
- as `&(*(a[i] + 0))`, which is equivalent to `&*a[i]`
- This is the same as `a[i]`, since the `&` and `*` operators cancel.

Processing the Rows of a Multidimensional Array

- A loop that clears row **i** of the array **a**:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

- Since **a[i]** is a pointer to row **i** of the array **a**, we can pass **a[i]** to a function that's expecting a one-dimensional array as its argument.
- In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.

Processing the Columns of a Multidimensional Array

- Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.
- A loop that clears column **`i`** of the array **`a`**:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

Multidimensional Arrays and Pointers

2-D array ≡ 1-D array of 1-D arrays

```
double rain[years][months] =
{ {3.1, 2.6, 4.3, …},
  {2.7, 2.8, 4.1, …},
  …
};
```

```
year = 3, month = 5;
rain[year][month] = 2.4;
```

`rain` is the **address** of the entire array

`rain[3]` is the **address** of the 4th row of the array

`rain[3][5]` is the **value** of the 6th element in the 4th row

`&rain[3][5]` is the **address** of the 6th element in the 4th row

```
double *yp, *mp;
yp = rain[3];
mp = yp + 5;
*mp = 2.4;
```

yp = address of 4th row

mp = address of 6th element in 4th row

…Multidimensional (cont'd)

Equivalent:

```
double *yp, *mp;
yp = rain[3];
mp = yp + 5;
*mp = 2.4;
```

inconsistent?

```
double *mp;
mp = &(rain[3][5]);
*mp = 2.4;
```

`rain` is the **address** of the entire array

`rain[3]` is the **address** of the 4th row of the array

`rain[3][5]` is the **value** of the 6th element in the 4th row

`&(rain[3][5])` is the **address** of the 6th element in the 4th row

The 1st dimension is an address, whereas the 2nd dimension is a value.

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.