

Pointers and Arrays

– Recall the pointer syntax:

- `char *cptr;`
 - declares a pointer to a char
 - allocates space to store a pointer (to a char)
- `char c = 'a';`
- `cptr = &c;`
 - `cptr` gets the value of the address of `c`
 - the value stored at the memory location referred to by `cptr` is the address of the memory location referred to by `c`;
- `*cptr = 'b';` – dereference `cptr`
 - the address stored at `cptr` identifies the memory location where 'b' will be stored.

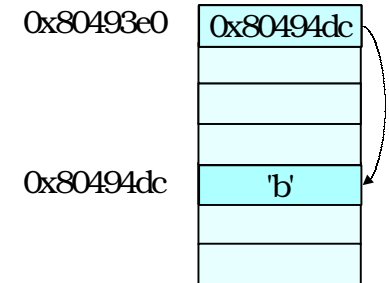
1

Pointers and Arrays

- `char *cptr;`
- `char c = 'a';`
- `cptr = &c;`
- `*cptr = 'b';`

Symbol Table

<code>cptr</code>	<code>0x80493e0</code>
<code>c</code>	<code>0x80494dc</code>



2

Arrays vs. Pointers

- An array name in expression context decays into a pointer to the zero'th element.
- E.g.

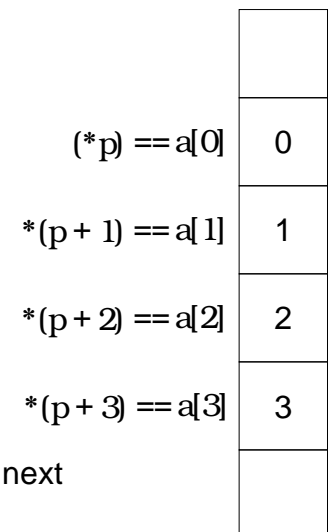

```
int a[3] = {1, 3, 5};
int *p = a;  p = &a[0];
p[0] = 10;
printf("%d %d\n", a[0], *p);
```

3

Example

```
int a[4] = {0, 1, 2, 3};
int *p = a;
int i = 0;

for(i = 0; i < 4; i++) {
    printf("%d\n", *(p + i));
}
```



Why does adding 1 to `p` move it to the next spot for an int, when an int is 4 bytes?

4

Pointer Arithmetic

- Pointer arithmetic respects the type of the pointer.
- E.g.,

```
int i[2] = {1, 2};    char c[2] = {'a', 'z'};
int *ip;             char *cp;
ip = i;              cp = c;
*(ip + 1) += 2;      *(cp + 1) = 'b';
(really adds 4 to ip) (really adds 1 to cp)
```
- C knows the size of what is being pointed at from the *type* of the pointer.

5

Pointer Arithmetic

- The array access operator [] is really only a shorthand for pointer arithmetic + dereference
- These are equivalent in C:
 $a[i] == *(a + i)$
- C translates the first form into the second.
 - *pointers* and *arrays* are nearly the same in C!

6

Passing Arrays as Parameters

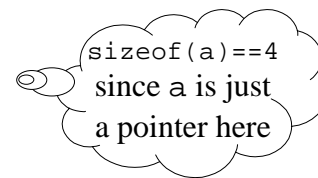
```
int main()
{
    int i[3] = {10, 9, 8};
    printf("sum is %d\n", sum(i)); /*??*/
    return 0;
}
int sum( What goes here? ) {
}
```

- What is being passed to the function is the name of the array which decays to a pointer to the first element – a pointer of type int.

7

Passing Arrays as Parameters

```
int sum( int *a ) {
    int i, s = 0;
    for(i = 0; i < ??; i++)
        s += a[i]; /* this is legal */
    return s;
}
```



- How do you know how big the array is?
- Remember that arrays are not objects, so knowing where the zero'th element of an array is does not tell you how big it is.
- Pass in the size of the array as another parameter.

8

Array Parameters

```
int sum(int *a, int size)
```

- Also legal is:

```
int sum(int a[], int size)
```

- Many advise against using this form.
 - You really are passing a pointer-to-int not an array.
 - You still don't know how big the array is.
 - Outside of a formal parameter declaration `int a[];` is illegal
- ⇒ `int a;` and `int a[10];` are completely different things

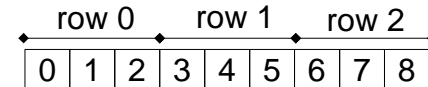
9

Summary

- The name of an array can also be used as a pointer to the zero'th element of the array.
- This is useful when passing arrays as parameters.
- Use array notation rather than pointer arithmetic whenever you have an array.

Multi-dimensional arrays

- Remember that memory is a sequence of bytes.



```
int a[3][3] = { {0, 1, 2},  
               {3, 4, 5},  
               {6, 7, 8}};
```

- Arrays in C are stored in row-major order
- row-major access formula

$x[i][j] == *(x + i * n + j)$
where n is the row size of x

But use array notation!