

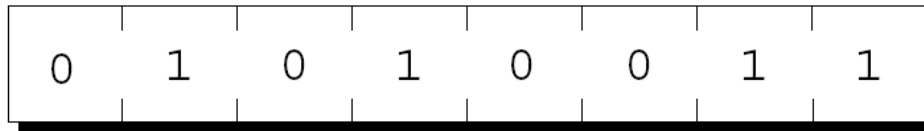
C Pointers

Goals of this Lecture

- Help you learn about:
 - Pointers and application
 - Pointer variables
 - Operators & relation to arrays

Pointer Variables

- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into *bytes*, with each byte capable of storing eight bits of information:



- Each byte has a unique *address*.

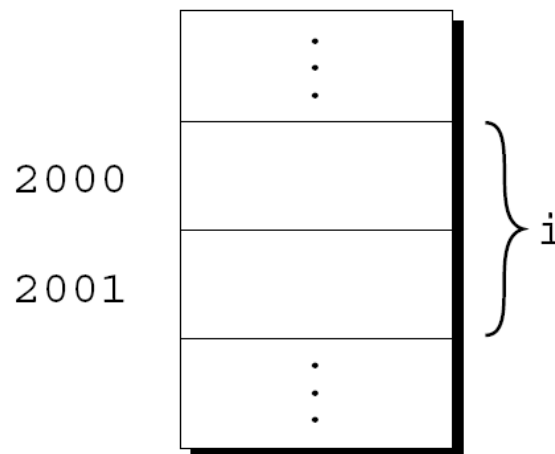
Pointer Variables

- If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
$n-1$	01000011

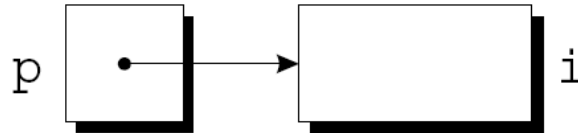
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special *pointer variables*.
- When we store the address of a variable i in the pointer variable p , we say that p "points to" i .
- A graphical representation:



Declaring Pointer Variables

- When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- `p` is a pointer variable capable of pointing to *objects* of type `int`.
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- C requires that every pointer variable point only to objects of a particular type (the *referenced type*):

```
int *p;      /* points only to integers    */  
double *q;   /* points only to doubles    */  
char *r;     /* points only to characters  */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
 - To find the address of a variable, we use the `&` (**address**) operator.
 - To gain access to the object that a pointer points to, we use the `*` (*indirection, dereference*) operator.

The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p; /* points nowhere in particular */
```

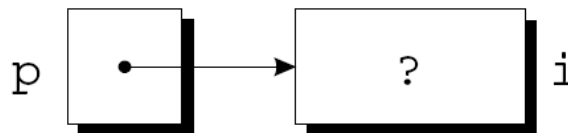
- It's crucial to initialize `p` before we use it.

The Address Operator

- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;  
...  
p = &i;
```

- Assigning the address of `i` to the variable `p` makes `p` point to `i`:



The Address Operator

- It's also possible to initialize a pointer variable at the time it's declared:

```
int i;  
int *p = &i;
```

- The declaration of `i` can even be combined with the declaration of `p`:

```
int i, *p = &i;
```

The Indirection Operator

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.

- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

- Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

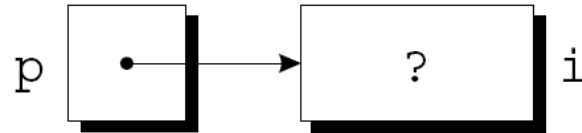
```
j = *&i;    /* same as j = i; */
```

The Indirection Operator

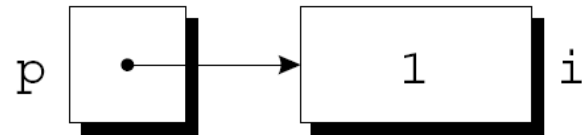
- As long as p points to i , $*p$ is an *alias* for i .
 - $*p$ has the same value as i .
 - Changing the value of $*p$ changes the value of i .
- The example on the next slide illustrates the equivalence of $*p$ and i .

The Indirection Operator

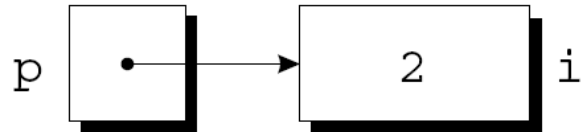
```
p = &i;
```



```
i = 1;
```



```
printf("%d\n", i);      /* prints 1 */  
printf("%d\n", *p);    /* prints 1 */  
*p = 2;
```



```
printf("%d\n", i);      /* prints 2 */  
printf("%d\n", *p);    /* prints 2 */
```

The Indirection Operator

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;  
printf("%d", *p);    /* ** WRONG ** */
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;  
*p = 1;    /* ** WRONG ** */
```


Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type.
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

- Example of pointer assignment:

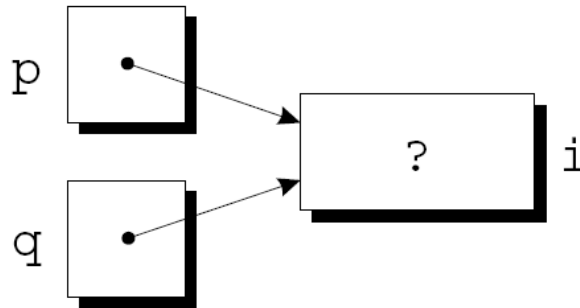
```
p = &i;
```

Pointer Assignment

- Another example of pointer assignment:

`q = p;`

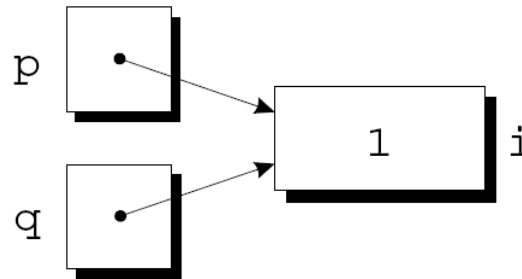
`q` now points to the same place as `p`:



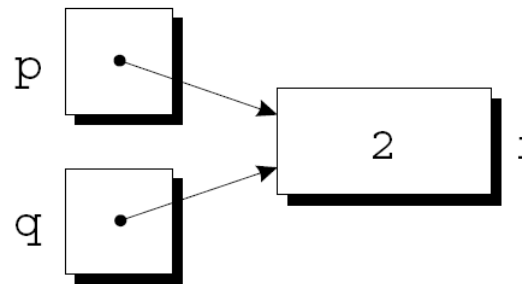
Pointer Assignment

- If p and q both point to i , we can change i by assigning a new value to either $*p$ or $*q$:

$*p = 1;$



$*q = 2;$



- Any number of pointer variables may point to the same object.

Pointer Assignment

- Be careful not to confuse

`q = p;`

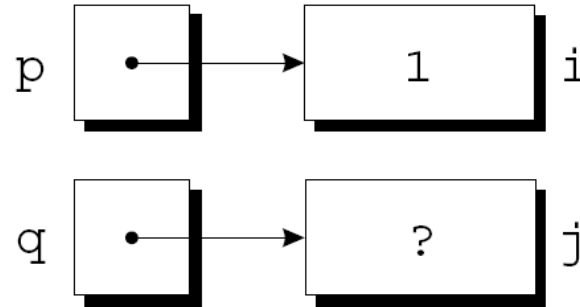
with

`*q = *p;`

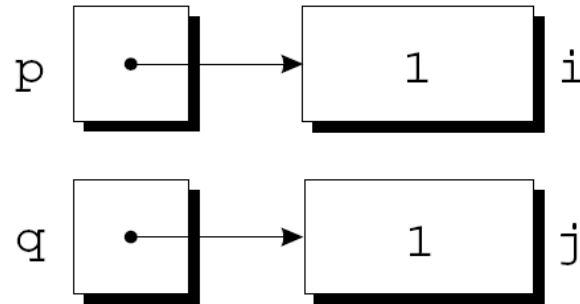
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;  
q = &j;  
i = 1;
```



```
*q = *p;
```



Pointers as Arguments

- Arguments in calls of `scanf` are pointers:

```
int i;
```

```
...
```

```
scanf("%d", &i);
```

Without the `&`, `scanf` would be supplied with the *value* of `i`.

Pointers as Arguments

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

```
int i, *p;  
...  
p = &i;  
scanf("%d", p);
```

- Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /* ** WRONG ** */
```

Using **const** to Protect Arguments

- When an argument is a pointer to a variable x , we normally assume that x will be modified:

$f(&x);$

- It's possible, though, that f merely needs to examine the value of x , not change it.
- The reason for the pointer might be efficiency: passing the value of a variable can waste time and space if the variable requires a large amount of storage.

Using `const` to Protect Arguments

- We can use `const` to document that a function won't change an object whose address is passed to the function.
- `const` goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
    *p = 0;    /*** WRONG ***/
}
```

Attempting to modify `*p` is an error that the compiler will detect.

Pointers as Return Values

- Functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
```

- A call of the `max` function:

```
int *p, i, j;
...
p = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

Pointers as Return Values

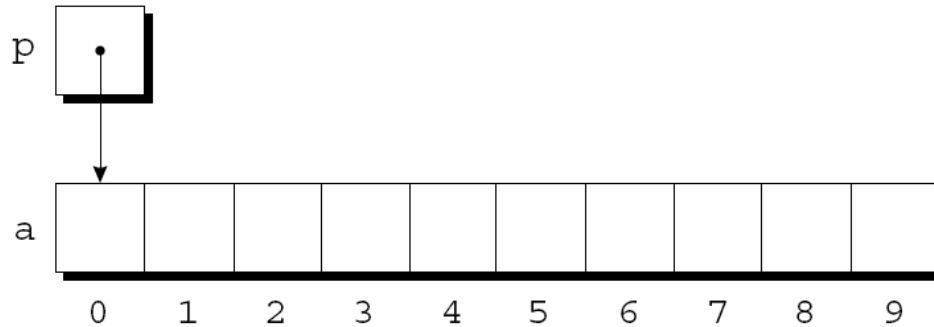
- Pointers can point to array elements.
- If `a` is an array, then `&a[i]` is a pointer to element `i` of `a`.
- It's sometimes useful for a function to return a pointer to one of the elements in an array.
- A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n)
{
    return &a[n/2];
}
```

Pointer Arithmetic

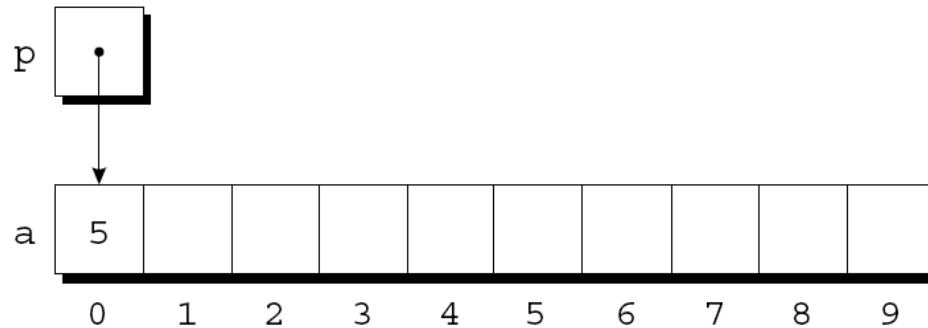
```
int a[10], *p;  
p = &a[0];
```

- 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

- *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

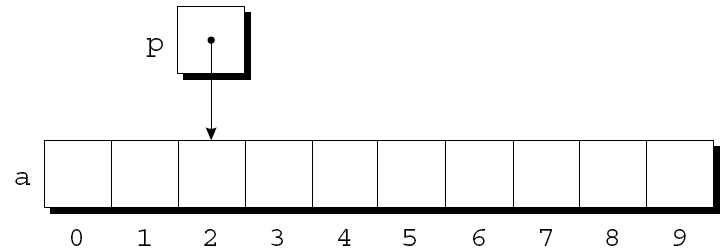
- Adding an integer j to a pointer p yields a pointer to the element j places after the one that p points to.
- More precisely, if p points to the array element $a[i]$, then $p + j$ points to $a[i+j]$.
- Assume that the following declarations are in effect:

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

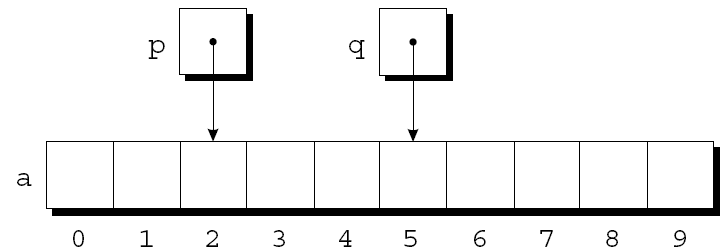
Adding an Integer to a Pointer

- Example of pointer addition:

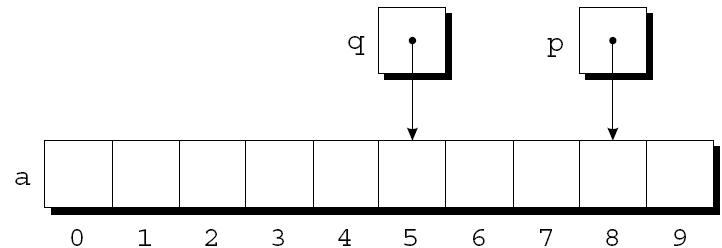
```
p = &a[2];
```



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



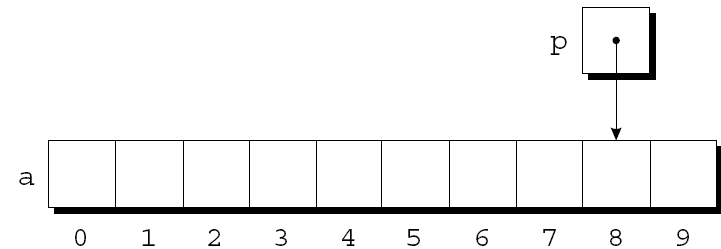
```
p += 6;
```



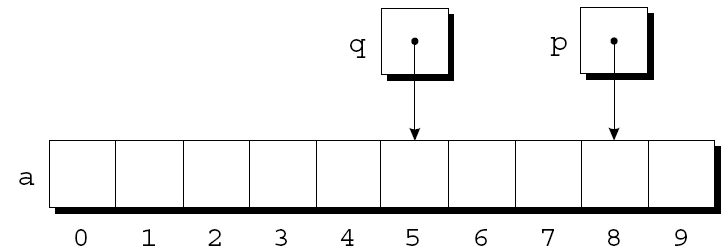
Subtracting an Integer from a Pointer

- If p points to $a[i]$, then $p - j$ points to $a[i - j]$.
- Example:

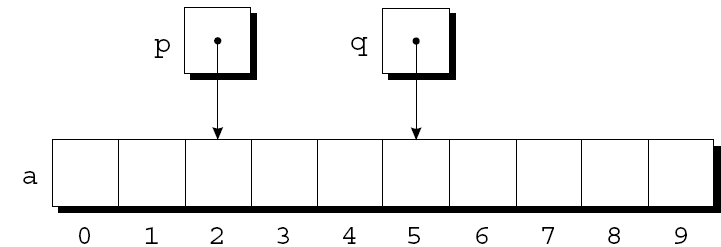
```
p = &a[8];
```



```
q = p - 3;
```



```
p -= 6;
```



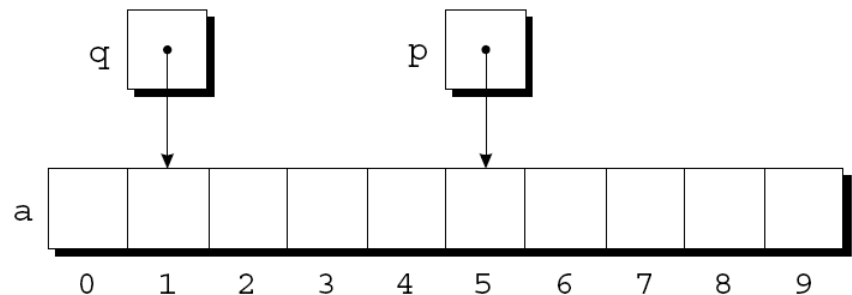
Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- If p points to $a[i]$ and q points to $a[j]$, then $p - q$ is equal to $i - j$.

- **Example:**

```
p = &a[5];  
q = &a[1];
```

```
i = p - q;    /* i is 4 */  
i = q - p;    /* i is -4 */
```



Subtracting One Pointer from Another

- Operations that cause undefined behavior:
 - Performing arithmetic on a pointer that doesn't point to an array element
 - Subtracting pointers unless both point to elements of the same array

Comparing Pointers

- Pointers can be compared using the relational operators ($<$, $<=$, $>$, $>=$) and the equality operators ($==$ and $!=$).
 - Using relational operators is meaningful only for pointers to elements of the same array.
- The outcome of the comparison depends on the relative positions of the two elements in the array.
- After the assignments

```
p = &a[5];  
q = &a[1];
```

the value of $p <= q$ is 0 and the value of $p >= q$ is 1.

Combining the * and ++ Operators

- C programmers often combine the * (indirection) and ++ operators.

- A statement that modifies an array element and then advances to the next element:

```
a[i++] = j;
```

- The corresponding pointer version:

```
*p++ = j;
```

- Because the postfix version of ++ takes precedence over *, the compiler sees this as

```
*(p++) = j;
```

Combining the * and ++ Operators

- Possible combinations of * and ++:

<i>Expression</i>	<i>Meaning</i>
*p++ or *(p++)	Value of expression is *p before increment; increment p later
(*p) ++	Value of expression is *p before increment; increment *p later
*++p or *(++p)	Increment p first; value of expression is *p after increment
++*p or ++(*p)	Increment *p first; value of expression is *p after increment

Combining the * and ++ Operators

- The most common combination of * and ++ is *p++, which is handy in loops.
- Instead of writing

```
for (p = &a[0]; p < &a[N]; p++) /* assume N+1 elms */  
    sum += *p;
```

to sum the elements of the array a, we could write

```
p = &a[0];  
while (p < &a[N])  
    sum += *p++;
```

Using an Array Name as a Pointer

- Pointer arithmetic is one way in which arrays and pointers are related.
- Another key relationship:
The name of an array can be used as a pointer to the first element in the array.
- This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

Using an Array Name as a Pointer

- Suppose that `a` is declared as follows:

```
int a[10];
```

- Examples of using `a` as a pointer:

```
*a = 7;           /* stores 7 in a[0] */  
*(a+1) = 12;     /* stores 12 in a[1] */
```

- In general, `a + i` is the same as `&a[i]`.
 - Both represent a pointer to element `i` of `a`.
- Also, `*(a+i)` is equivalent to `a[i]`.
 - Both represent element `i` itself.

Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it's **not possible** to assign it a new value.
- Attempting to make it point elsewhere is an error:

```
while (*a != 0)
    a++;                /*** WRONG ***/
```

- This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
    p++;
```

Array Arguments

- When passed to a function, an array name is treated as a pointer.

- **Example:**

```
int find_largest(int a[], int n)
{
    int i, max;

    max = a[0];
    for (i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}
```

Array Arguments

- The fact that an array argument is treated as a pointer has some important consequences.
- *Consequence 1:* When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.
- In contrast, an array used as an argument isn't protected against change.

Array Arguments

- To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
    ...
}
```

- If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

Array Arguments

- *Consequence 2:* The time required to pass an array to a function doesn't depend on the size of the array.
- There's no penalty for passing a large array, since no copy of the array is made.

Array Arguments

- *Consequence 3*: An array parameter can be declared as a pointer if desired.
- `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
    ...
}
```
- Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Array Arguments

- The following declaration causes the compiler to set aside space for 10 integers and assign the address of first element to `a`

```
int a[10];  
*a = 0; /* What happens? */
```

- The following declaration causes the compiler to allocate space for a pointer variable:

```
int *a;  
*a = 0; /* What happens? */
```


Array Arguments

- *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:

```
largest = find_largest(&b[5], 10);
```

Summary

- Pointers and their operations
 - Pointer has a memory address as its value
 - & is address operator
 - * is indirection/dereference operator
 - Function arguments
 - Typically used to change the value of the passed variable
 - Call-by-reference semantics
 - Relation to the arrays
 - Array name can be used as a pointer assigned with the address of its first element