Announcements

- Assignment 1
  - *Cellular Automata* with strings in C
  - Due 5:00 p.m., Friday, September 26
- Test 1
  - Friday, September 26
  - Covers up to Lecture 8
  - Format
    - Part I: Multiple-choice (30%)
    - Part II: Short-answer (70%)

Lecture 9

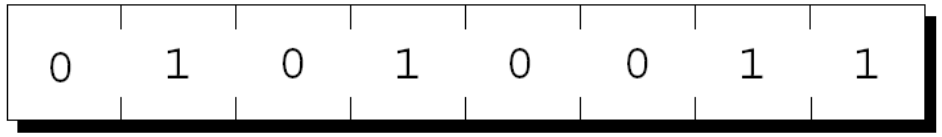
Pointers in C

CPSC 275
Introduction to Computer Systems

Building Blocks of Memory

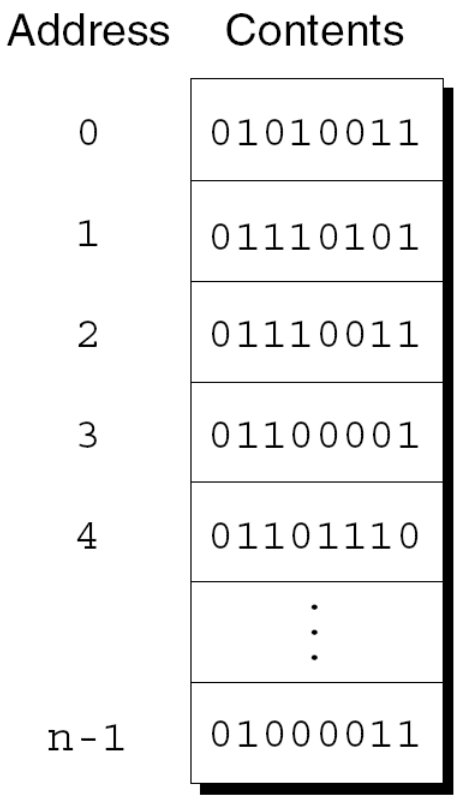
- In most modern computers, main memory is divided into ***bytes***.

- Each byte has a unique ***address.***



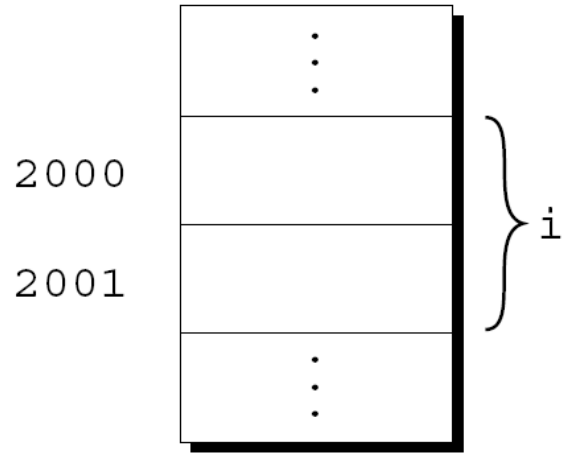
Memory Address

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



Variables in Memory

- 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.



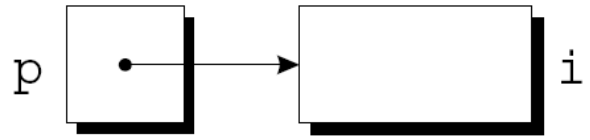
the address of the variable `i` is 2000, where `i` is a `short int`

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

```
int *p;
```

- A graphical representation:



Declaring Pointer Variables

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

```
int *p;
double *q;
char *r;
```

- There are no restrictions on what the reference 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*) 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 */
```

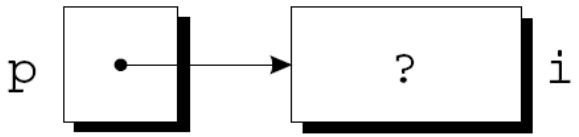
- It's crucial to initialize `p` before we use it.

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 Indirection Operator

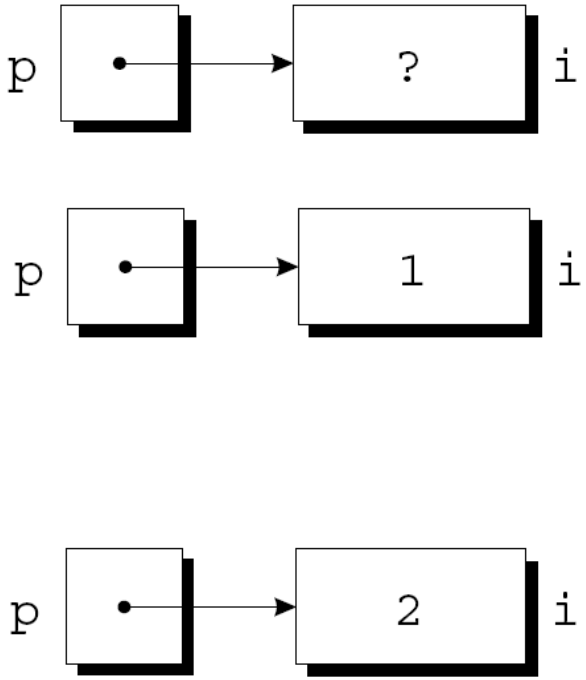
- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.

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

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

The Indirection Operator

```
int i, *p;

p = &i;

i = 1;

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

*p = 2;

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



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:

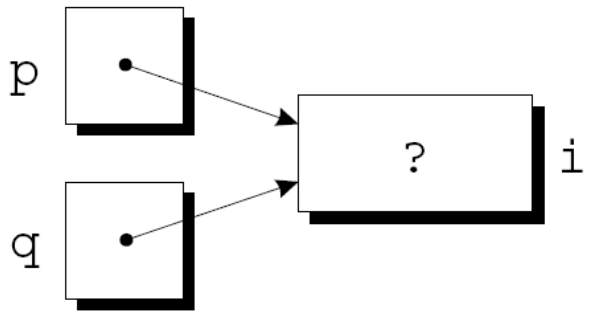
```
int *p;
*p = 1;    /*** WRONG ***/
```

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;
p = &i;
q = p;
```

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

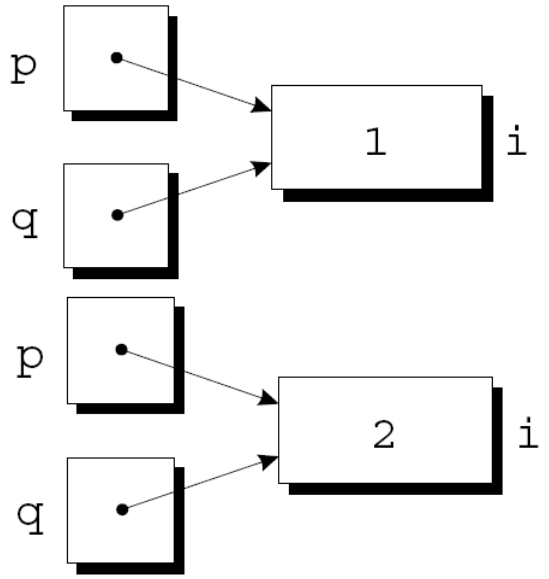


Pointer Assignment

- 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;
```



Pointer Assignment

- Be careful not to confuse

```
q = p;
```

  with

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

- The first statement is a pointer assignment, but the second is not.

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`.

The swap() function

- What's wrong with the following function?

```
int swap(int a, int b) { // swap values of a and b
  int temp = a;
  a = b;
  b = temp;
}
```

- A correct version:

```
int swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

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

- Never return a pointer to an *automatic* local variable:

```
int *f(void) {
  int i;
  …
  return &i;
}
```

Why not?

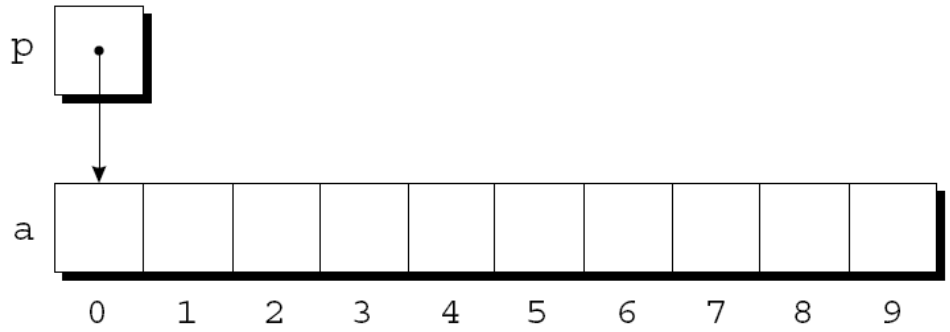
The variable `i` won't exist after `f` returns.

Accessing Arrays Using Pointers

- Pointer variables can point to array elements:

```
int a[10], *p;
p = &a[0];  // same as p = a; (why?)
```

- A graphical representation:

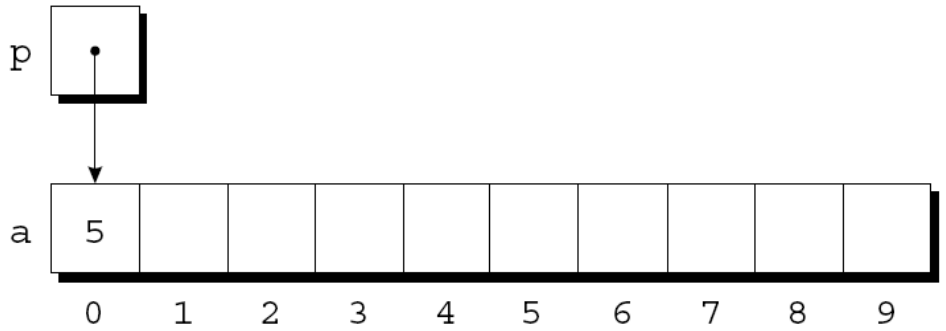


Accessing Arrays Using Pointers

- We can now access `a[0]` through `p`:

```
*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 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]`.

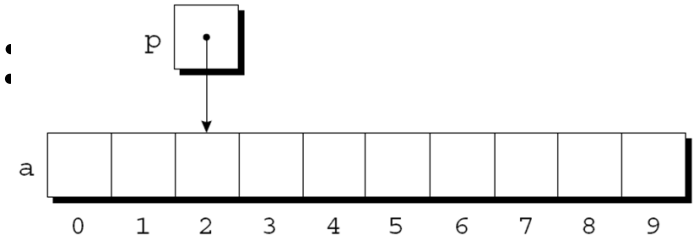
Adding an Integer to a Pointer

- Assume that the following declarations:

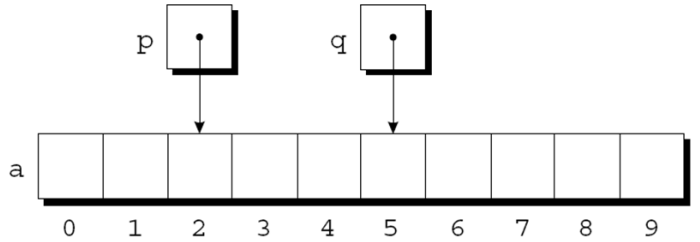
```
int a[10], *p, *q, i;
```

- Example of pointer addition:

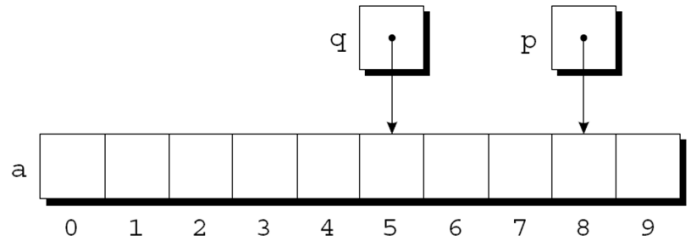
```
p = &a[2];
```




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



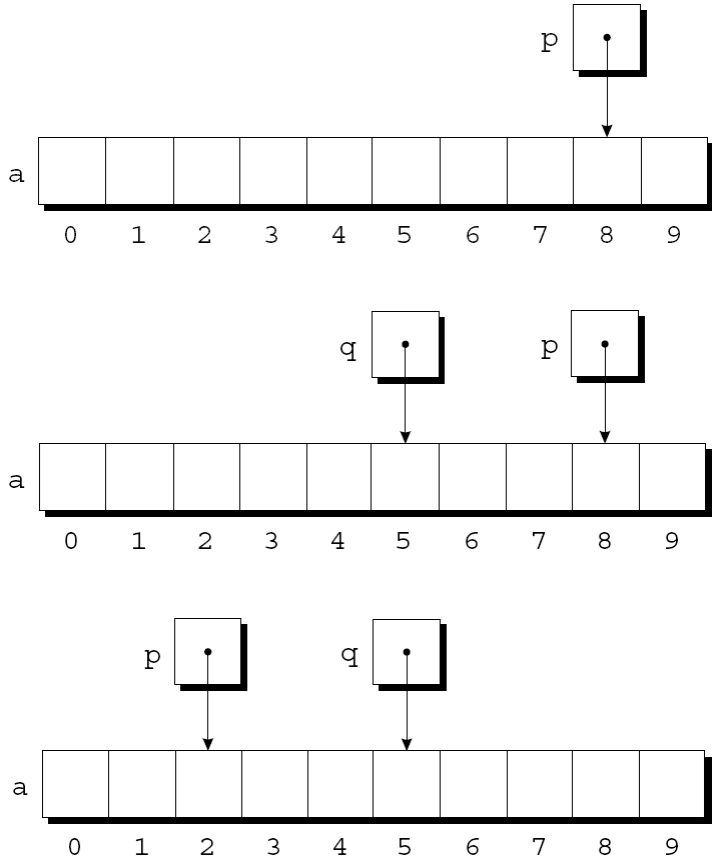

```
p += 6;
```



Subtracting an Integer from a Pointer

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

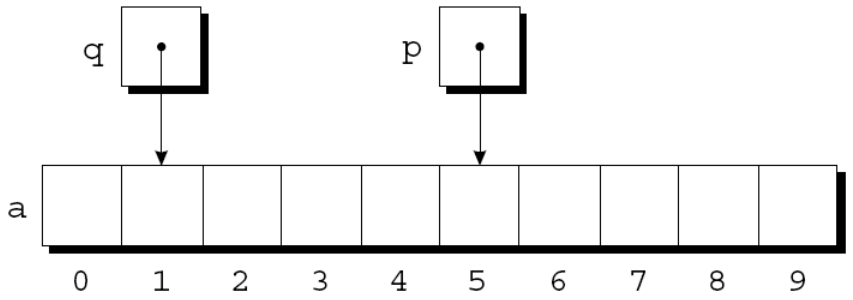
```
p = &a[8];

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 */
```



Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
- 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` ? 0

the value of `p >= q` ? 1

Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array `a`:

```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

How String Are Stored

- Since a string is stored as an array of `char`, the compiler treats it as a pointer of type `char *`.
- Example:

```
char p[] = “abc”;
```

or

```
char *p;

p = "abc";
```

- So, what is the type of the first argument of `printf` ?

String Literals vs Character Constants

- A string literal containing a single character isn't the same as a character constant.

  `"a"` is represented by a *pointer*.

  `'a'` is represented by an *integer*.

- A legal call of `printf`:

```
printf("\n");
```

- An illegal call:

```
printf('\n');    /*** WRONG ***/
```

Character Arrays vs Character Pointers

- The declaration

```
char date[] = "June 14";
```

declares `date` to be an *array*,

- The similar-looking

```
char *date = "June 14";
```

declares `date` to be a *pointer*.

Character Arrays vs Character Pointers

- However, there are significant differences between the two versions of `date`.
  - In the array version, the characters stored in `date` can be modified.
  - In the pointer version, `date` points to a string literal that shouldn't be modified.

Q: How many bytes will be allocated for each case?

Character Arrays vs Character Pointers

- The declaration

```
char *p;
```

  does not allocate space for a string.

- Before we can use `p` as a string, it must point to an array of characters.

- One possibility is to make `p` point to a string variable:

```
char str[STR_LEN+1], *p;
p = str;
```

- Another possibility is to make `p` point to a dynamically allocated string. (TBD)

Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripting :

```
int count_spaces(char *s) {
  int count = 0;

  for (; *s != '\0'; s++)
    if (*s == ' ')
      count++;
  return count;
}
```

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

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

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

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

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