int a[40];

Rewrite the following using pointer notation instead of array notation:

• a[0]
• a[1]
• a[2]
• a[39]
• a[40]
Quiz on Ch.9

```c
int a[40];
```

Rewrite the following using pointer notation instead of array notation:

- `a[0]++;`
- `++a[0]`
- `a[2]--;`
- `a[39] = 42;`
Chapter 10:
Characters and Strings
Character codes

- Characters such as letters, digits and punctuation are stored as integers according to a certain numerical code.
- One such code is the **ASCII** code; the numeric values are from 0 to 127, and it requires 7 bits.
- **Extended ASCII** requires 8 bits: see table [here](#) and App.A of our text.
- Note well:
  - While the 7-bit ASCII is a **standard**, the 8-bit Extended ASCII is not.
  - There exist several versions of Extended ASCII! The version shown in our App.A is called “IBM code page 437”.
QUIZ ASCII

Use the ASCII table to decode the following string:

67 79 83 67 32 49 51 49 48

Use App. A of text, or search ASCII table online.
Character codes

• The C data type `char` stores Extended ASCII codes as 8-bit integers.
• `char` variables can be initialized as either integers, or quote-delimited characters:

```
    char c = 'a';
    char c = 'A';
    char c = 97;
    char c = 65;
```
Use the ASCII table to figure out a simple relationship between the codes for uppercase and lowercase letters!
Characters

How to test if a character variable holds a lowercase letter:

```c
if (c >= 'a' && c <= 'z');
```

Your turn: test if c is

• An uppercase letter
• A decimal digit
12.2 Character Input and Output

The following functions have prototypes in `<stdio.h>`

1) `int getchar(void);`
   Reads the next character from the standard input.

2) `int putchar(int c);`
   Print the character stored in c

3) `int scanf(const char *, ...);`
   with conversion specifier `c`.
   ```
   char c; scanf("%c", &c);
   ```

4) `int printf(char *, ...);`
   with conversion specifier `c`.
   ```
   char c='a'; printf("%c", c);
   ```
```c
#include <stdio.h>

void main() {
    char c;

    printf("Please input a character\n");
    c = getchar();
    printf("ASCII c = %d, c = %c\n", c, c);
    putchar(c);

    printf("\n");
    printf("Note: ASCII '\n' = %d, '\n' = %c\n", '\n', '\n');

    /* obtain the character '\n' */
    c = getchar();
    printf("ASCII c = %d, c = %c\n", c, c);
    putchar(c);
}
```
Character-Handling Functions

– The header file `ctype.h` declares several functions to perform testing and manipulations of characters.

– Each function has one argument of character (int type). The value of argument shall be representable as an unsigned char or shall equal to the value of the macro `EOF`, defined in header file `stdio.h`.

– Each function returns a nonzero value for true or zero for false.
<table>
<thead>
<tr>
<th>Function Prototype</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int isdigit(int c)</code></td>
<td>Returns true if c is a digit and false otherwise.</td>
</tr>
<tr>
<td><code>int isalpha(int c)</code></td>
<td>Returns true if c is a letter and false otherwise.</td>
</tr>
<tr>
<td><code>int isalnum(int c)</code></td>
<td>Returns true if c is a digit or a letter and false otherwise.</td>
</tr>
<tr>
<td><code>int isxdigit(int c)</code></td>
<td>Returns true if c is a hexadecimal digit character and false otherwise.</td>
</tr>
<tr>
<td><code>int islower(int c)</code></td>
<td>Returns true if c is a lowercase letter and false otherwise.</td>
</tr>
<tr>
<td><code>int isupper(int c)</code></td>
<td>Returns true if c is an uppercase letter; false otherwise.</td>
</tr>
<tr>
<td><code>int tolower(int c)</code></td>
<td>If c is an uppercase letter, tolower returns c as a lowercase letter. Otherwise, tolower returns the argument unchanged.</td>
</tr>
<tr>
<td><code>int toupper(int c)</code></td>
<td>If c is a lowercase letter, toupper returns c as an uppercase letter. Otherwise, toupper returns the argument unchanged.</td>
</tr>
<tr>
<td><code>int isspace(int c)</code></td>
<td>Returns true if c is a white-space character—newline ('\n'), space (' '), form feed ('\f'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v')—and false otherwise.</td>
</tr>
<tr>
<td><code>int iscntrl(int c)</code></td>
<td>Returns true if c is a control character and false otherwise.</td>
</tr>
<tr>
<td><code>int ispunct(int c)</code></td>
<td>Returns true if c is a printing character other than a space, a digit, or a letter and false otherwise.</td>
</tr>
<tr>
<td><code>int isprint(int c)</code></td>
<td>Returns true value if c is a printing character including space (' ') and false otherwise.</td>
</tr>
<tr>
<td><code>int isgraph(int c)</code></td>
<td>Returns true if c is a printing character other than space (' ') and false otherwise.</td>
</tr>
</tbody>
</table>
#include <ctype.h>
#include <stdio.h>

void main() {
    char c = 'A';
    printf("isdigit('%c') returns %d\n", c, isdigit(c));
    if(isdigit(c))
        printf("'\%c' is a digit.\n", c);
    else
        printf("'\%c' is not a digit.\n", c);
}
c = '2';
printf("isdigit('%c') returns %d\n", c, isdigit(c));
if(isdigit(c))
    printf("'\%c' is a digit.\n",c);
else
    printf("'\%c' is not a digit.\n",c);
}

Output:

isdigit('A') returns 0
'A' is not a digit.
isdigit('2') returns 1
'2' is a digit.
Strings

“YOUNG APPRENTICE, KEEP THIS KNOWLEDGE CLOSE TO THY HEART: STRINGS ARE ARRAYS OF CHARACTERS”

(Inscription found on a secret tablet from the mythical Temple of C)
Discussion of exam problems
Quiz

Declare a character variable `ch`, initialize it with the ASCII code for the character `@`, and then print both the character and the ASCII code.
Strings

Strings are sequences of characters treated as a single unit

May include:

- Letters a, b, A, B, ...
- Digits 0, 1, 2, ...
- “Normal” characters underscore, space, !, ?, ;
- Certain special characters (*, /, $)

C lingo: String literal (a.k.a. string constant) = sequence of characters enclosed in double quotes, e.g. "Hello, World!"

Similar to the numeric literals, e.g. -42, 1.55
Two ways to initialize strings

• With an initializer list, like any other array:

```c
char str1[7]={'S','t','r','i','n','g','\0'};
char str2[ ]={'S','t','r','i','n','g','\0'};
```

• With a string literal:

```c
char str3[7] = "String";
char str3[ ] = "String";
char *strPtr = "String";
```
Declare a string variable `st`, initialize it with your name, and then print it.

• Do the initialization all 5 ways!
Strings – using sizeof

The value returned by sizeof depends on how the string was declared:

```c
char str1[7] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
char str2[] = {'S', 't', 'r', 'i', 'n', 'g', '\0'};
char str3[] = "String";
char *strPtr = "String";

sizeof(str1);    //7
sizeof(str2);    //7
sizeof(str3);    //7
sizeof(strPtr);  //4
```
# 12.5 String Input and Output

These functions have prototypes in `<stdio.h>`

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>Function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *gets(char *s);</td>
<td>Inputs characters from the standard input into the array <code>s</code> until a newline or end-of-file character is encountered. A terminating null character is appended to the array.</td>
</tr>
<tr>
<td>char *fgets(char *s, int n, FILE *stream);</td>
<td>Input maximum number of <code>n-1</code> characters from the stream pointed to by <code>stream</code> into the array <code>s</code>. No additional characters are read after a newline or end-of-file character is encountered. A terminating null character is appended to the array.</td>
</tr>
<tr>
<td>int scanf(const char *, ...);</td>
<td>Reads a string with conversion specifier <code>s</code>.</td>
</tr>
<tr>
<td>int printf(const char *, ...);</td>
<td>Prints individual strings with conversion specifier <code>s</code>.</td>
</tr>
<tr>
<td>int puts(const char *s);</td>
<td>Prints the string <code>s</code> followed by a newline character.</td>
</tr>
<tr>
<td>int sprintf(char *s, const char *format, ...);</td>
<td>Equivalent to <code>printf</code>, except the output is stored in the array <code>s</code> instead of printing it on the screen.</td>
</tr>
<tr>
<td>int sscanf(const char *s, const char *format, ...);</td>
<td>Equivalent to <code>scanf</code>, except the input is read from the array <code>s</code> instead of reading it from the keyboard.</td>
</tr>
</tbody>
</table>

Note: The functions `scanf()` and `gets()` are vulnerable to buffer overflow! Use `fgets()` in production code.
/* Demonstrates displaying strings with puts(). */

#include <stdio.h>

char *message1 = "C";
char *message2 = "is the";
char *message3 = "best";
char *message4 = "programming";
char *message5 = "language!!";

int main( void )
{
    puts(message1);
    puts(message2);
    puts(message3);
    puts(message4);
    puts(message5);
    return 0;
}
/* Demonstrates using the gets() library function. */

#include <stdio.h>

/* Allocate a character array to hold input. */

char input[257];

int main( void )
{
    puts("Enter some text, then press Enter: ");
    gets(input);
    printf("You entered: %s\n", input);
    return 0;
}
How to read function prototypes

I entered “gets prototype” in a web search engine and found this at
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.12.html#gets

2.12.5.7 gets

Declaration:

```c
char *gets(char *str);
```
```c
/* getback.c--Demonstrates using the gets() return value. */

#include <stdio.h>

/* Declare a character array to hold input, and a pointer. */

char input[257], *ptr;

int main(void)
{
    /* Display instructions. */
    puts("Enter text a line at a time, then press Enter.");
    puts("Enter a blank line when done.");

    /* Loop as long as input is not a blank line. */
    while (*ptr = gets(input)) != '\0')
        printf("You entered %s\n", input);

    puts("Thank you and good-bye\n");
    return 0;
}
```

Explained on the next slide
How to read function prototypes

The value returned by `gets()` is rarely used, except for this typical “C idiosyncrasy”:

```c
/* Loop as long as input is not a blank line. */

while ( *(ptr = gets(input)) != '\0' )
    printf("You entered %s\n", input);
```
How to read function prototypes

I entered “gets prototype” in a web search engine and found this at
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.12.html#gets

2.12.5.7 gets

Declaration:

```c
char *gets(char *str);
```
How to read function prototypes

Can `fgets()` be used in a similar way?

2.12.5.2 `fgets`

Declaration:

```c
char *fgets(char *str, int n, FILE *stream);
```
The truth:

Both scanf() and gets() are **unsafe** because they allow for **buffer overflow**!!

Do not use them in your code!!
char * fgets ( char * str, int num, FILE * stream );

Reads characters from stream and stores them as a C string into `str` until either `(num-1)` characters have been read or a newline or the end-of-file is reached, whichever happens first.

A newline character makes `fgets` stop reading, but it is considered a valid character by the function and included in the string!

A terminating null character is automatically appended after the characters copied to `str`. 
fgets() example

#include <stdio.h>
#include <string.h>

void main() {
    char str[10];

    printf("Please input a string.\n");
    fgets(str, sizeof(str), stdin);  //assumption user enters Hi
    printf("string len = %d, str= %s\n", strlen(str), str);
    puts(str);
    /* remove the return character at the end of string */
    str[strlen(str)-1] = '\0';
    printf("string len = %d, str= %s\n", strlen(str), str);
    puts(str);
    puts("foo");
}
Execution:

Please input a string.
Hello
string len = 6, str= Hello

Hello
string len = 5, str= Hello
Hello
foo

These empty lines show that \n is part of the string!
The empty lines are now missing, showing that \n was removed!
**LISTING 14.6  fgets.c: Using the fgets() Function for Keyboard Input**

```c
/* Demonstrates the fgets() function. */

#include <stdio.h>

#define MAXLEN 10

int main( void )
{
    char buffer[MAXLEN];

    puts("Enter text a line at a time; enter a blank to exit.");

    while (1)
    {
        fgets(buffer, MAXLEN, stdin);

        if (buffer[0] == '\n')
            break;

        puts(buffer);
    }
    return 0;
}
```
QUIZ

Declare a string of 30 characters, and then ask user to initialize it from the kbd., using:

- `scanf`
- `gets`
- `fgets`

- Which is the preferred method?
- Why?
Quiz

Declare a **string** variable `st`, initialize it with the string “Strings!” (in two ways!) and then print it (using two different output functions!).
QUIZ

What is the major difference between `scanf/gets` and `fgets`?

• Which is the preferred input function?
• Why?
Why do some variables have &, and some don’t?
Difference between (scanf) and (gets, fgets)

This program demonstrates a limitation of the `scanf()` function. Suppose the first name you needed to enter had two names like Mary Ellen? The `scanf()` function would put only Mary into the `fname` variable due to the space, even though “Mary Ellen” is a completely acceptable string. So you would either need to create two variables for the two parts of the first name or ask the user to enter the name without a space. For this reason, `gets()` can be a more effective tool to retrieve strings from users, particularly if the strings will have spaces in them.
Determining string length

```c
size_t strlen(const char *s);
```

- Returns the number of characters (before `NULL`) in string `s`
- Data type `size_t` is declared as `unsigned int` type for 32-bit machines and unsigned long long type for 64-bit machines
char str[10] = "Hello";

sizeof(str); // 10
strlen(str); // 5
8. **BUG BUSTER:** Is anything wrong with the following?

```c
char a_string[10] = "This is a string";
```

9. **BUG BUSTER:** Is anything wrong with the following?

```c
char *quote[100] = { "Smile, Friday is almost here!" };
```
The Line Continuation Character \\n
It’s used to span a string over multiple lines.

Example:

```
printf("This is a very very very very very very very very long string that is taking up multiple lines of text to write.");
```

Another method is to separate the string into shorter ones:
```
printf("This is also a very very very very very very very long string, but it is separated into three shorter strings.");
```
QUIZ

Finish this code to print the string in reverse order. Do not assume a known length for str:

```c
char str[] = "Hello";
```
QUIZ

Finish this code to print the string in reverse order. Do not assume a known length for str!

```c
char str[] = "Hello";
int i;
for (i=4; i>=0; i--)
    putchar(str[i]);
```
Print the string in reverse order. Do not assume a known length for str:

```c
#include <string.h>

char str[] = "Hello";
int i;
for (i=strlen(str)-1; i>=0; i--) 
    putchar(str[i]);
```
Strings w/o arrays

char st[ ] = "String";
char st[7] = "String";
char *stPtr = "String";

?????????????????????????? ;
Strings w/o arrays

char st[ ] = "String";
char st[7] = "String";

char *stPtr = "String";

??????????????????????????

Compile-time vs. run-time
Strings w/o arrays: `malloc()`

```c
char *stPtr = "String";
```

Example 1

```c
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
    /* allocate memory for a 100-character string */
    char *str;
    str = (char *) malloc(100);
}
```

Read from R to L: “pointer to a char”
Remember discussion of null pointers from Ch.9! Here, NULL is a symbolic constant defined in stdlib.h.

For the first time in our class, this is needed!

This 1 is returned by main() to the OS! (To the “C runtime”, really ... )
The professional/flexible/portable way to allocate memory with `malloc()`

Literal values were used to allocate space for characters in the above examples. You should always multiply the size of the data type you are allocating space for by the amount of space you want. The previous allocations assumed that a character was stored in just 1 byte. If a character is stored in more than 1 byte, then the previous examples overwrite other areas of memory. For example:

```c
ptr = malloc(100);
```

should actually be declared as

```c
ptr = malloc( 100 * sizeof(char));
```
QUIZ: malloc()

Allocate memory for an array of:

• 42 integers
• 24 floats
• 10 doubles
• 10,000 long longs
QUIZ

Declare a string of 30 characters using `malloc()`

• What do we need to do right after calling `malloc()`?

• Why?
malloc() example

LISTING 10.3 memalloc.c: Using the malloc() Function to Allocate Storage Space for String Data

1:  /* memalloc.c--Demonstrates the use of malloc() to allocate storage */
2:  /* space for string data. */
3:  
4:  #include <stdio.h>
5:  #include <stdlib.h>
6:  
7:  char count, *ptr, *p;
malloc() example

```c
9:  int main( void )
10: {
11:   /* Allocate a block of 35 bytes. Test for success. */
12:   /* The exit() library function terminates the program. */
13:   ptr = malloc(35 * sizeof(char));
14: 
15:   if (ptr == NULL)
16:   {
17:     puts("Memory allocation error.");
18:     return 1;
19:   }
20: }
```
malloc() example

```c
29:    p = ptr;
30:
31:    for (count = 65; count < 91 ; count++)
32:        *p++ = count;
33:
34:    /* Add the terminating null character. */
35:
36:    *p = '\0';
37:
38:    /* Display the string on the screen. */
39:
40:    puts(ptr);
41:
42:    free(ptr);
43:
44:    return 0;
45: }
```

Warning: Do not try to assign an entire string like this:

```c
ptr = "ABCDDE";
```
malloc() example
Q Why shouldn’t I just declare big arrays to hold values instead of using a memory allocation function such as malloc()?

A Although it might seem easier to declare large arrays, this isn’t an effective use of memory. When you write small programs, such as those in this lesson, it might seem trivial to use a function such as malloc() instead of arrays, but as your programs get bigger, you want to allocate memory only as needed. When you’re done with memory, you can put it back by freeing it. When you free memory, some other variable or array in a different part of the program can use it. (Lesson 21, “Working with Memory,” covers freeing allocated memory.)
Homework for Ch. 10:

Ch. 10: 6, 10, 11, 12

Due Friday, March 18
The second exam is the week after Spring Break!

Monday after S.B. is review!