For all of these topics, a short summary of the basics is provided. Consult your C book for further details, and above all, practice writing C code!

◊ Pointers

- Each variable in C is the *name* of a particular memory address that can store the appropriate type of data. For example, after declaring an variable like

  ```
  int x;
  ```

  `x` is the name of a memory address that can store an `int`. What if you want to know the actual numerical address of `x`, not just its name? That's what pointers are for.

- Recall from the lecture that if `x` is a variable, then `&x` is the numerical address of `x` in memory. The inverse of `&` is `*`, called the *indirection* or *dereferencing* operator. If `p` is a pointer, then `*p` is the data that `p` points to. For example, if we have the following declarations:

  ```
  int i, *p;
  ```

  then `p = &i;` is a valid assignment, but `p = i;` is not. Similarly, `i = *p;` is valid but `i = p;` is not.

  Pointers can be copied directly: if

  ```
  int *p, *q;
  ```

  has been declared, then `p = q;` is a valid assignment.

- Question: Would `*p = *q;` be a legal assignment? What would it accomplish?

- Exercise: One of the exercises from Lecture #1 was to write a program that reverses a string. Here is a function `reverse()` which does this using arrays:

  ```
  void reverse(char s[]) {
    int i, len = strlen(s);
    char temp;

    for (i = 0; i < len / 2; i++) {
      temp = s[len - i - 1];
      s[len - i - 1] = s[i];
      s[i] = temp;
    }
  }
  ```

  How can we do the same thing with pointers?

  ```
  void reverse(char *s) {
    char *p = s;  /* p points to beginning of string */
    char temp;
  ```

```
  /* Find the trailing null character */
  while (*p != '\0')
    p++;

  if (p == s)   /* Empty string -- nothing to do! */
     return;

  p--;  /* Now p points to the last character of the string */

  /* Keep looping until we hit the middle of the string */
  while (s < p) {
    /* Swap *s and *p */
    temp = *s;
    *s = *p;
    *p = temp;

    /* Each pointer moves towards the middle of the string */
    s++;
    p--;
  }
}
```

We can take advantage of some quirks of C (*e.g.*, operator precedence, nonzero is true) to streamline this function:

```
void reverse(char *s) {
  char *p = s;
  char temp;

  while (*p)
    p++;

  while (s < p) {
    temp = *s;
    *s++ = *--p;
    *p = temp;
  }
}
```

◇ Formatted input

- To use formatted I/O, you need to `#include <stdio.h>`.
- Basic output function: `printf()`. `printf()` takes a *format string* and a sequence of expressions as arguments. In the format string, characters are printed as they appear, except for certain special *conversion specifications* which are used to display the expressions following the format strings.
- Example:

```
printf("The value of x is: %d.\n", x);
```

produces output

```
The value of x is 5.
```

followed by a newline (if `x` indeed has value 5).

- Basic conversion specifications:
  - `%c`: format a single character (`char`)
  - `%s`: format a string (pointer to or array of `char`)
  - `%i` or `%d`: format a `signed int` in decimal (*i.e.*, base 10)
  - `%u`: format an `unsigned int` in decimal (*i.e.*, base 10)
  - `%f`: format a `double` in regular notation (no exponent)
  - `%e`: format a `double` in exponential notation
  - `%g`: format a `double` in either `%f` or `%e` format (whichever is more appropriate; the number gets an exponent only if it needs one)

  So for example, the following bit of code:

  ```
  double x = 12.953, y = 0.0000000000134;
  printf("%f %e %g\n%f %e %g\n", x, x, x, y, y, y);
  ```

  produces the following output on the screen:

  ```
  12.953000 1.295300e+01 12.953
  0.000000 1.340000e-11 1.34e-11
  ```

  You can see that generally, `%g` produces the nicest output for floating-point numbers.

- Why aren't there separate conversion specifications for `float` and `double`? Because `float` is automatically promoted to `double`.

- Here's an example with characters and strings:

  ```
  char s[] = "animal";
  printf("The first letter of %s is %c.\n", s, s[0]);
  ```

  The output is:

  ```
  The first letter of animal is a.
  ```

- Important: the number and types of arguments after the format string must exactly match the number and types of conversion specifications in the format string! Most C compilers will not complain if you send too few/too many arguments, or arguments of the wrong type. Your program will compile, but it may crash, or at best produce erroneous output.

- There are certain modifiers you can apply to the conversion specifications to alter the appearance of the arguments. For example, you can specify a precision and field width for floating-point numbers. Consult your C book or type `man 3c printf` at the Unix prompt to learn about them.

◇ Formatted output

- Basic input function: `scanf()`. `scanf()` takes a format string and a sequence of *pointers* as arguments. (Remember that you can get a pointer to a given variable by prefixing the variable with the *address-of* operator, `&`. For example, `&x` is a pointer to the variable `x`.)
- The format string tells `scanf()` what kind of data to expect, and the pointers tell it where to store the data it reads. For example,

  ```
  int a, b;
  scanf("%d%d", &a, &b);
  ```

  will read two integers written in decimal notation and store them in variables `a` and `b`.
- Basic conversion specifications:
  - `%d` for `signed int`
  - `%u` for `unsigned int`
  - `%f` for `float`
  - `%lf` for `double`

  All other characters in the format string must be matched *exactly*, except for whitespace characters, which stand for *any* amount of whitespace in the input (including none).
- `scanf()` stops once it's read all the data it's supposed to, or when it encounters an unexpected character. Beware—unexpected characters are left in the input buffer to be read by the next `scanf()` call!
- `scanf()` returns the number of conversion specifications successfully matched, or the special identifier `EOF` if the end of input was reached.
- Why are there separate conversion specifications for `float` and `double`? Because the arguments to `scanf()` are pointers, and we can't automatically promote `float *` to `double *`. (More generally, we can almost never promote one type of pointer to another.)
- Exercises:
  1. `scanf("%d",&a);`
     What happens with input `5667` and with the input `12hi`?
  2. Write a program that accepts a date from the user in the form *mm/dd/yy* and then displays it in the form *yymmdd*.