Objectives:
(a) Explain the operation of the address operator.
(b) Given the source code of a C program which uses pointers, and the output of the debugger, locate the associated variables in a memory diagram.
(c) Describe the relationships that exist between pointers, arrays and strings.
(d) Differentiate between the value of a pointer and the address of a pointer.

I. Pointers

1. Why Use Pointers?
Consider your favorite database: MIDS. Suppose you tell the registrar that you want to register for the following courses:

   HU222: Football Theory
   HH333: History of Embarrassing Air Force Scandals
   FP444: Cicero’s Impact on Renaissance Poetry as Developed for Theatre circa 1700
   EC312: Cyber Security II

Now, other people also have to use (and possibly modify) your record in the MIDS database. Your advisor has to check that you are on track for your major. Your instructors have to enter grades into your record. The Dean may have to annotate your matrix with a note saying that one course counts for another course. Various people may have to enter conduct offenses.

So, let’s say that it comes time for your HU222 instructor to enter your six-week grade. How can this be accomplished? Recall that the MIDS database, as with everything else a computer uses, resides in memory.

One option would be for the registrar to send your HU222 instructor a copy of the entire MIDS database, with a note: “Update the database, and then send it back to me.”

This involves the duplication and movement of vast amounts of data. Can you think of a better way?

A better way would be to let the instructor modify the actual database… not a copy of the database. But how can that be done?

Suppose your instructor was given the address of the MIDS database in memory. Then, any changes made to the contents of a memory location are actual changes to the MIDS database! In other words, instead of sending someone the MIDS database, we simply tell them: "Here is the address of the MIDS database, go make your changes directly".

2. Addresses

Recall from week one of the course that all variables have

- a type
- a name (also called an identifier)
- a value (possibly a garbage value)
- an address

We will focus on this notion of a variable’s address. Addresses are 4 bytes (32 bits) long. Since people don’t like reading 32-bit binary numbers, the shorthand hexadecimal notation is used. As mentioned previously, memory locations are usually given in hexadecimal notation.
Practice Problem
If the first byte of a variable is stored at memory location numbered:

```
000000000001001011111111011111100
```
what is this address in hexadecimal notation?

Solution: 0000 0000 0001 0010 1111 1111 0111 1100

Practice Problem
For our x86 architecture, how many hexadecimal digits are in an address?

Solution:

3. The address operator &

& (the ampersand sign) is the address operator which is used to access the address of a variable. &variable returns the address of variable. So, if we have a variable named y, then &y returns the address of y.

To examine how the ampersand operator behaves, consider the program below, along with its associated output.

```
#include<stdio.h>
int main()
{
    int current_year = 2014;
    printf("\nThe year is %d and the address is %x \n" , current_year, &current_year);
}
```

```
midshipman@EC310:~/work $ ./a.out
The year is 2014 and the address is bffff854
midshipman@EC310:~/work $ 
```

The variable named current_year has the value 2014 and is stored at this address.

This variable is an integer and so consumes four bytes on the stack—that section of memory the program has available to hold its variables and other data.
4. **Pointers**  A pointer is a variable that holds a memory address, usually the address of another variable. Put another way, a pointer is a data type that holds the hexadecimal address of another variable. Put a third way: A pointer variable “points to” another variable. A pointer variable is itself stored in 4 bytes.

A. **Declaring Pointers**  A pointer variable is declared by using the asterisk. For example, the declaration

```c
data_type  *pointer_name ;
```

means:

"I am declaring a pointer named pointer_name that will point to another variable of type
data_type".

B. **Example**  Consider the declaration

```c
float *a_ptr ;
```

*What does this mean?*  It means that we have told C that we would like to have a pointer variable named
a_ptr that will be used to point to some float variable (but it does not point anywhere yet!).

C. **Assigning Values to Pointers.**  Recall that a pointer variable holds an address. To assign the correct address to a pointer variable, we use the & operator (the address operator). Let's consider the code snippet below:

```c
int a = 131;  
int *a_ptr ;  
a_ptr = &a;  
```

Let's look at this code snippet line-by-line.

The very first line (int a = 131;) tells the compiler that you want to use a variable named a, of type integer, that will be initialized to the value of 131. The compiler will place the value of a on the program's stack (recall that the stack is the section of memory that a program has available to hold its variables and any other data it needs to perform its operation).

The operating system (not you, the programmer) will decide the precise address of where the variable a will be placed. Let's presume that the compiler has chosen to store the variable a at address 0056DE73.

The second line of code (int *a_ptr;) tells the compiler that you want to use a pointer variable named a_ptr, that will (at some time later in its life), point to variable of type int. This variable must also be stored on the stack. Let's presume that the compiler has chosen to store the variable a_ptr at address 0056DA61.
Finally, the third line of code (`a_ptr = &a;`) places the address of `a` into the variable `a_ptr`.

Practice Problem

Consider the program shown below, along with its corresponding output.

```c
#include<stdio.h>
int main()
{
    int a = 4;
    int *a_ptr;
    a_ptr = &a;

    printf("\nThe value of a is %d and the address is %x \n", a, &a);

    printf("\nThe value of a_ptr is %x and the address is %x \n\n", a_ptr, &a_ptr);
}
```

```
midshipman@EE488-VM:~/booksrc $ ./a.out
The value of a is 4 and the address is bffff854
The value of a_ptr is bffff854 and the address is bffff850
```
In the picture shown below:

(a) Fill in the two red circles.
(b) Draw an arrow showing where a_ptr is stored on the stack.
(c) Annotate the figure to show the value of a_ptr.

Pointers are confusing! Some argue that pointers are the greatest source of bugs in C programs. In fact, some modern programming languages (such as Java) have eliminated pointers altogether since pointers are so confusing and lead to so many errors.

But it is precisely because pointers are confusing that leads to their use by adversaries. Almost all program attacks involve the use or abuse of a pointer.

The Morris worm... Conficker... Stuxnet... these all employ a buffer overflow attack. The key to understanding this attack involves understanding pointers.

5. **Arrays and Pointers** Recall our discussion of arrays from last lecture. We mentioned that if we declare an array with

```c
float pay[4];
```

the compiler will reserve four consecutive memory location, each of which will hold a float variable. The four variables are pay[0], pay[1], pay[2] and pay[3]. The array of all four variables is named pay.

Recall also that in the C programming language, strings—i.e., sequences of characters—are stored as arrays of characters. Here is an example of declaring a string variable using a character array:

```c
char school[20] = "US Naval Academy";
```

The array named school holds 17 characters:

```c
school[0] = 'U'
school[1] = 'S'
school[2] = '
'school[3] = 'N'
```

... etc., etc.,

school[16] = 0
So... we can see what school[1] is (it's a character... the character 'S' to be exact), but what exactly is the array name school by its lonesome...without an index? Let's see!

```c
#include<stdio.h>
int main()
{
    char school[20] = "US Naval Academy" ;
    printf("\nThe value of school is %x \n\n" , school ) ;
}
```

We have output an address...school was holding an address... school is a pointer.

The bottom line is: An array name is a pointer!

When we declare an array, C generates a pointer, which is assigned the address of the first element of the array.

Consider the declaration

```c
int a [4];
```

What really happens is

```c
int *a = &a[0];
```

For strings, we can tell C to print out a string by using the array name and the string format specifier. If we changed our program to:

```c
#include<stdio.h>
int main()
{
    char school[20] = " US Naval Academy" ;
    printf("\nThe value of school is %s \n\n" , school ) ;
}
```

Only change! The old x is changed to an s.

---

1 It should be pointed out that this is not a line of valid C code. Our meaning is to convey: It’s as if this happened...
Practice Problem

Recall that in RAM you have stored the machine language code for your program as well as additional memory allocated for your variables within the program. This latter additional memory is called the stack.

You type into the debugger the command

```
  i r ebp
```

and get the result 0xbffff818. The register ebp points to the "bottom" of the stack.

Upon further review of the assembly code you determine that two strings are stored in memory, one at address ebp-40 and the other at ebp-24. (Note that the numbers 40 and 24 are ordinary base 10 numbers, not base-16.)

What are the two hidden words?

Solution:
A large program contains the following lines of code

```c
int a = 11;
int b[2];
b[0] = 10;
b[1] = 6;
```

A section of this program's stack is shown below.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xBFFFF8F0</td>
<td>0x3E</td>
</tr>
<tr>
<td>0xBFFFF8F1</td>
<td>0x3F</td>
</tr>
<tr>
<td>0xBFFFF8F2</td>
<td>0x4A</td>
</tr>
<tr>
<td>0xBFFFF8F3</td>
<td>0x0A</td>
</tr>
<tr>
<td>0xBFFFF8F4</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8F5</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8F6</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8F7</td>
<td>0x06</td>
</tr>
<tr>
<td>0xBFFFF8F8</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8F9</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8FA</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8FB</td>
<td>0x0B</td>
</tr>
<tr>
<td>0xBFFFF8FC</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8FD</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8FE</td>
<td>0x00</td>
</tr>
<tr>
<td>0xBFFFF8FF</td>
<td>0x4D</td>
</tr>
<tr>
<td>0xBFFFF900</td>
<td>0x08</td>
</tr>
<tr>
<td>0xBFFFF901</td>
<td>0x2C</td>
</tr>
<tr>
<td>0xBFFFF902</td>
<td>0x33</td>
</tr>
</tbody>
</table>

What would be the result of the statement:

```c
printf(“The address of array b is %x \n”, b);
```

Solution:
Practice Problem

Recall that in RAM you have stored the object code for your program as well as additional memory allocated for your variables within the program.

You type into the debugger the command

\[ \text{i r ebp} \]

and get the result 0xbffff810. Upon further review of the assembly code you determine that two integers are stored in memory, one at address ebp−8 and the other at ebp+4. What are the hidden decimal numbers?
Why does `scanf` sometimes require an ampersand in front of the variable's name and sometimes not?

Recall that `scanf` is another name for the keyboard. When the writers of the C programming language designed the `scanf` statement, the intent was that the item into which the keyboard input is placed would be provided as an address.

What does that mean? That means that when we have a `scanf` statement, such as

```c
scanf("%f", &year_number);
```

the place where we deposit the keyboard input, shown as a red box in the statement above, must be an address. That is simply the way the C language was written.

So if we declare a variable of, say, type `float` as in

```c
float EC310_torture_factor;
```

and we want to read in the value of the `EC310_torture_factor` from the keyboard, we cannot do this:

```c
scanf("%f", EC310_torture_factor);
```

The reason we cannot do this is because `EC310_torture_factor` is not an address, and the `scanf` statement always expects that you will provide it an address into which to deposit the keyboard input.

So, we can place the value read in from the keyboard into the variable `EC310_torture_factor` by providing the address of `EC310_torture_factor` using the address operator:

```c
scanf("%f", &EC310_torture_factor);
```

Knowing how the `scanf` statement was designed should provide insight into how strings are entered from the keyboard. Last lecture that we mentioned that when entering strings from the keyboard, you don’t use the ampersand: `&`. For example, to enter a midshipman's last name from the keyboard, you would use:

```c
char mid_name[24];
scanf("%s", mid_name);
```

The reason: Recall from page 142 above that an array name is a pointer—when we declare an array, C generates a pointer, which is assigned the address of the first element of the array. So, in this particular case, `mid_name` is an address already, so we do not add the ampersand!

---

**An Aside**

Error! Bad midshipman!

Correct! Good midshipman!

---

**OPTIONAL:** For your reading and viewing pleasure, here is an example of a “buffer overflow attack”. We will discuss the buffer overflow in greater detail, but for now, let's just watch a Pre-Snowden video:

[http://www.cbsnews.com/video/watch/?id=7400904n](http://www.cbsnews.com/video/watch/?id=7400904n) (first 6-7 minutes)