A brief intro to pointers for the purposes of passing reference parameters

With respect to functions, we have only talked about pass by value parameters. Today we will discuss pass by reference parameters. But, in order to use these, we have to understand a bit about how pointers work in C.

A computer's memory can be visualized as a very long numbered array of bytes. Each byte of memory is numbered with it's address. Perhaps a simple analogy would be that each byte of memory is a house on a really long street. Each of these houses are numbered in order. If you ever want to retrieve a value in a house, one way you could do so is use the correct street address. This is what a pointer is - a street address directing you to where some information is stored.

However, this is not how you access or manipulate information usually. Instead, you create a variable through a declaration. Internally, when you declare a variable, what the computer does is find a "house" to store that variable. Once a house is found, it keeps a list of all the declared variables and the houses they live in. Anytime in your program you refer to a variable, it automatically goes to the contents of the house that match the address given on the main list. In some sense, you are referring to the house by who lives there. (The computer takes care of looking up where that person lives...)

What a pointer allows you to do is access a variable, without using a name for it. Instead, a pointer just stores a location in memory, and through that pointer, you are allowed to change the value of the variable stored at that location.

First, let's go through the basic syntax of how to declare a pointer:

int *p;

This literally says to create a variable p. The type of the variable p is a "pointer to an integer," not just a "pointer."

p can not store a value, but it can store a memory address.

Now, let's say we have the added declaration:

int a = 7;

One statement that is useful with pointers is to assign them a location to point. However, this statement:

p = a; is illegal? Why?

Instead, we need a way to find the memory address of where the variable a is stored to make this a valid statement. We can do this with the "address of" operator, &. The correct statement is

p = &a;

Basically here is what these three statements are doing in memory:

The other thing we want to be able to do is to access a variable that a pointer is pointing to. We can do this through the "dereferencing" operator, *. Notice that * is used in two different ways. When we declare a pointer, we use * to denote that a variable is a pointer. But NOW, we see that the * allows us to dereference a pointer as well. Here is an example of what we can do:

```
*p = 3;
```

This says to find the variable that p is pointing to and change it to 3. Since this variable is a, you'll see that if we printed out the value of a:

```
printf("a = %d\n",a)
```

The output would be a = 3. Thus, we were able to change the value of a without directly using the identifier a. This is because p was pointing to where a was stored.

These are essentially all the mechanics we need in order to use pass by reference variables.

How could pass by reference variables help us?

One "subtask" that is common in many computer programs is swapping the value of two variables. (One of our solutions to the desk problem included a swap.) Imagine writing a function that performs such a swap with two integer variables, it would look something like this:

```
void swap(int a, int b) {

   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

Now, imagine a segment of code in main as follows:

```
int x = 3, y = 7;
swap(x,y);
printf("x = %d, y= %d\n",x,y);
```

What would get printed out? Why?

So the problem here is that no matter what swap does, as long as it takes in pass by value parameters, the values of x and y are guaranteed to be what they were before the swap function call.

In essence, it may be desirable to have a function that has the ability to manipulate variables that are locally declared in other functions.

Pointers allow you to refer to a variable without using its name. Thus, if we can pass pointers are parameters, perhaps there is a way around the apparent pass-by-value problem:

```
void swap(int *p, int *q) {

    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
}
```

It's important to note that in the formal parameter definition, * means pointer, but in the code, it is the dereferencing operator.

So, now the question becomes, how do we CALL this function. You'll notice that the types of the formal parameters are NOT ints, but "pointers to ints." Thus, when we call the swap function, we can not pass actual parameters that are integers, we must pass parameters that are pointers to ints. Here is how we would do that:

```
int x = 3, y = 7;
swap(&x, &y);
printf("x = %d, y= %d\n",x,y);
```

Remember that &x is read as "address of x." This is a pointer - exactly what we want to pass to the function swap. Let's trace through the code written above:

A simple pass by reference example

```
#include <stdio.h>

void deposit(int* accnt_bal);
void withdraw(int* accnt_bal);
void menu();
int main() {

    int balance;
    char ans;

    // Read in starting balance
    printf("Enter your starting balance.\n");
    scanf("%d", &balance);

     // Loop until user quits main menu.
     menu();
     scanf("%c", &ans);
     while (ans != '3') {

          // Execute appropriate menu option.
          if (ans == '1')
               deposit(&balance);
          else if (ans == '2')
               withdraw(&balance);
          else if (ans != '3')
                printf("Invalid menu choice, please try again.\n")

          menu();
          scanf("%c", &ans);
     }
     return 0;
}
```

```
// Pre-condition: accnt_bal is the current bank balance.
// Post-condition: The bank balance will be adjusted according
//                 to what the user enters as their deposit.
//                 No negative deposits are processed.
void deposit(int* accnt_bal) {

    // Read in deposit.
    int dep;
    printf("Enter the amount of your deposit.\n");
    scanf("%d", &dep);

    // Only adjust balance if necessary.
    if (dep > 0)
        *accnt_bal += dep;
    else
        printf("That deposit can not be executed.\n");
}

// Pre-conditions: none
// Post-condition: Menu for the bank program will get printed.
void menu() {

    printf("Please enter your next menu selection.\n");
    printf("1. Deposit money to your account.\n");
    printf("2. Withdraw money from your account.\n");
    printf("3. Quit this program.\n");
}
```

Class Exercise: Write the withdraw function.

Tracing Question for next time

```
int f1(int *a, int b);
int f2(int a, int *b);

int main() {

   int a = 5, b = 2, c = 7, d = 9;

   c = f1(&d, a);
   printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);
   a = f2(c - d, &a);
   printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);
   b = f1(&c, 8);
   printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);
   d = f2(b, &a);
   printf("a=%d,b=%d,c=%d,d=%d\n",a,b,c,d);

}

int f1(int *a, int b) {
   *a = b - 8;
     b = b*2 - (*a);
     printf("a=%d,b=%d\n",*a,b);
     return b - *a;
}

int f2(int a, int *b) {

     a = *b + a;
     *b = 37 - *b;
     printf("a=%d,b=%d\n",a,*b);
     return a;
}
```