Pointer Review

No matter what your 106A experience has been, probably a little more experience with pointers wouldn't hurt. I put together some reminders about various pointer facts that you want to have a solid grasp on before we move on. The first assignment also includes a few problems dealing with pointers to help get you back into the swing of things.

Declaring a pointer only allocates space for the pointer itself

Before you try to dereference a pointer, you need to be sure that it is initialized to point somewhere valid. You can initialize a pointer in a variety of ways: set it to the address of a variable, assign it from another pointer to create an alias to the same location, or dynamically allocate new memory from the heap. Trying to dereference an uninitialized pointer is a sure way to cause a lot of trouble. It won't be caught by the compiler but will cause havoc during execution.

```
{
      int *p, *q, num;

      *p = 10;      /* bad news */
      p = &num;     /* initialize one way or another! */
      p = GetBlock(sizeof(int));
      p = NewArray(5, int);
      q = p;
}
```

Arrays and pointers— the same thing or not?

In many situations, arrays and pointers can be treated equivalently. An array can be thought of as the address of a particular variable in memory, which is exactly what a pointer is. It may be that there is just one variable at that location or a contiguous sequence of variables, depending on how the space was allocated and set up. You can dereference either a pointer or array with a * or with array bracket notation.

One crucial difference between arrays and pointers concerns their declarations. Declaring a stack array makes space for the entire sequence of variables whereas declaring a pointer only makes space for the (uninitialized) pointer. For example:

```
{
      int *p, arr[10];

      arr[4] = 8;   /* ok! there are 10 integers in arr */
      p[4] = 8;     /* not ok! p is uninit pointer! */
      p = arr;      /* now it's ok */
      p = &arr[4];
      p = 10;       /* assigns 10 to arr[4] */
      *arr = 10;    /* assigns 10 to arr[0] */
}
```

If the array is declared on the stack, its storage will be deallocated on function exit. That's perfect if you don't plan on needing or using the array after this function is done. Space on the stack is efficient to obtain and you don't have to worry about deallocation. However, if you are creating an array to return out of the function, that space must be dynamically allocated in the heap and you will need to declare a pointer to hold its address and return that pointer.

One other thing to note about declaring a stack array— the declared size must be a fixed compile-time constant. This means you cannot declare **`arr[n]`** where n is some variable. The array must always of a fixed-size. If we truly want to configure an array to a specified size at runtime, our only choice is to use dynamic allocation since it allows you to determine how much space to allocate at runtime.

Pointers must be used for call-by-reference

Parameters in C are passed by value, meaning a copy is made. If we want a function to change values in a way that we can see it when the function exits, we need to use pointers to simulate call by reference. As an example, the function below exchanges the values of two integer variables passed by reference:

```
void Swap(int *p1, int *p2)
{
      int tmp;

      tmp = *p1
      *p1 = *p2;
      *p2 = tmp
}
```

When we call this function, we pass the address of the two integers to swap:

```
      Swap(&num1, &num2);
      Swap(&arr[0], &arr[n-1]);
```

Arrays are always passed by reference

Contrary to the rule above, arrays are a special case in parameter passing. An array parameter is passed by reference, which is to say, a pointer to the first element is all that is copied when calling a function that takes an array parameter. No copy of the array is made.

In fact, the two declarations

```
int Sum(int arr[], int n)     int Sum(int *arr, int n)
```

are entirely equivalent and work exactly the same. Both of them are passed a pointer that is expected to point to at lest one integer, and potentially an entire sequence of integers. (We are probably passing the effective size as the parameter n, so likely we expect arr to point to a sequence of n integers).

Inside the body of this function, we can use array bracket or * notation interchangeably, regardless of whether the parameter is declared as an array or a pointer. In this context, an array and a pointer are no different.

```
int Sum(int arr[], int n)          or          (int *arr, int n)
{
      int i, sum = 0;

      for (i = 0; i < n; i++)
            sum += *arr++;         or           sum += arr[i];
      return sum;
}
```

Dynamically allocated memory has to be explicitly freed

Unlike local variables that are automatically wiped out when the function exits, storage allocated from the heap (via **`malloc/GetBlock/New)`** is persistent and requires an explicit action on your part to be reclaimed. The **`free`** (standard C) or **`FreeBlock`** (its genlib equivalent) function is passed a pointer to dynamically allocated space that you are no longer using and returns that space to the available pool of memory. You must be very careful with freeing though— you should only free pointers that point to heap-allocated storage, the pointers should only be freed once, and you should not try to access the contents of an address after it has been freed. Although we largely ignored free last quarter, we will pay a bit more attention to efficient use of memory as we move ahead in the curriculum

For example, here is a function to reverse a string using the strlib.h functions. Each time it is called, it allocates many temporary strings and then "orphans" them in the heap forever since it never frees them:

```
string ReverseString(string s)
{
      int i;
      string result = "":

      for (i = 0; i < StringLength(s); i++)
            result = Concat(CharToString(IthChar, s, i), result);
      return result;
}
```

Here is a version that is much tidier about its use of memory, it frees each of the temporary strings before moving on (notice how the function got messier as a result):

```
string ReverseString(string s)
{
      int i;
      string ch, prev, result = CopyString("");

      for (i = 0; i < StringLength(s); i++) {
            ch = CharToString(IthChar, s, i);
            prev = result;
            result = Concat(ch, prev);
            FreeBlock(prev);
            FreeBlock(ch);
      }
      return result;
}
```

Careful: why did we make a copy of the empty string to initialize the starting result?

Another solution could make just one copy of the input string and reverse the characters in-place. Assume you have a Swap function that takes two characters by reference similar to the int one shown earlier. This solution is a bit more efficient since it has no extra allocation and deallocation, also the fewer allocation/deallocation calls, the higher likely you will get all the memory details right!

```
string ReverseString(string s)
{
      int i;
      string r = CopyString(s); /* make copy to change */

      for (i = 0; i < StringLength(r)/2; i++)
            Swap(&r[i], &r[StringLength(s)-i-1]);
      return r;
}
```