Auto-allocating arrays

Often you need to allocate memory.

```
{
    int x[10];    // Auto allocate 10 ints on the stack
    x[5] = 3;
    ...
}                 // Auto deallocate
```

~ No function call: fast and convenient.
~ Runtime stack limited to 2-8KB *total*. Careful!

Manually allocating arrays

```
{
    int *y = malloc(10 * sizeof(int));
    y[5] = 3;
    ...
    free(y);
}
```

~ Function call. Remember to free when done.
~ Any size allowed. Exists after scope exits.
~ Array notation okay applied to pointer.

Array/pointer interchangeable

```
int x[10];
int *y = malloc(10 * sizeof(int));
x[1] = 5;
y[1] = 5;
```

In both cases above, x and y should be thought of as holding an address.

Array notation [] can be applied to both.

Neither has a "length" function.

Example: Reverse array

```
// Reverse array (upto len 100), return number copied
int reverse(int *a, int elems) {
    if (elems <= 100) {
        int t[100];
        for (int i=0; i<elems; i++)
            t[i] = a[elems-i-1];
        for (int i=0; i<elems; i++)
            a[i] = t[i];
        return elems;
    } else
        return 0;
}
```

Crashes if not enough stack.

Example: Safer reverse array

```
// Reverse array. Return num copied.
int reverse(int *a, int elems) {
    int *t = malloc(elems * sizeof(int));
    if (t != NULL) {
        for (int i=0; i<elems; i++)
            t[i] = a[elems-i-1];
        for (int i=0; i<elems; i++)
            a[i] = t[i];
        free(t);
        return elems;
    } else
        return 0;
}
```

Allocation more expensive.

Example: Better reverse array

```
// Reverse array. Return num copied.
int reverse(int *a, int elems) {
    for (int i=0; i<elems/2; i++) {
        int t = a[i];
        a[i] = a[elems-i-1];
        a[elems-i-1] = t;
    }
    return elems;
}
```

Best: Robust and $O(1)$ space.

Obtaining memory addresses

Sometimes you want the address of an array element.

x+i evaluates to the address of x[i].

"pointer arithmetic" adds as many bytes to x as are needed to get to x[i].

```
int x[10];
int *y = malloc(10 * sizeof(int));
int *p = x+5;      // p holds address of x[5]
int *p = y+5;      // p holds address of y[5]
```