HANDOUT to accompany H15: (Dynamic Memory Allocation, Etter, Section 6.7) CS16, 10W, UCSB
Available online at: http://www.cs.ucsb.edu/~pconrad/cs16/10W/homework/H15/handout (PDF)

Goes with homework assignment: http://www.cs.ucsb.edu/~pconrad/cs16/10W/homework/H15 (PDF)

This homework assignment requires
(1) some **reading in your textbook**
(2) reading some additional notes on this sheet,
(3) working through some problems **online, using Ch on CSIL**

To access Ch on CSIL— you can:

- visit CSIL in person,
- access the CSIL command line on "csil.cs.ucsb.edu" via PuTTY on Windows
- or use "ssh username@csil.cs.ucsb.edu" on Mac or Linux

(1) The textbook part

Read Section 6.7 in your Etter textbook about character strings—but as you do, keep these things in mind.

- The reading will talk about a ***cast operator***—an operation that converts one type to another.
  - For example: `average = (float) sum / count;`
    where `(float)` in front of `sum` converts `sum` from `int` to `float` before the division happens.
  - You may want to review this idea at the top of page 46 in your textbook.
  - In Section 6.7, cast operators are used to convert pointers, e.g. from (void *) to (int *)
- The reading also talks about `(void *)` pointers
  - Despite the name, a `(void *)` doesn't mean a "pointer to nothing"
  - Rather it means a pointer to "any old type"—i.e. a "generic pointer"
  - We frequently have to convert `(void *)` pointers to specific pointers, e.g. `(int *)` or `(char *)` before we can use them—that's where the cast operator comes in.
- Finally, the reading mentions the `size_t` data type.
  - The book says that `size_t` "is system dependent, but is usually either an `unsigned int`, or an `unsigned long`."
  - If you want to review the meaning of `unsigned int` and `unsigned long`, check the first paragraph on p. 40.
- You are responsible for all of 6.7, except for the part about the `realloc()` function—we aren't going to cover that in CS16.
  - Read all of p. 313, and p. 314 almost to the end.
  - Skip all this:
    from near the bottom of p. 314, paragraph starting
    "*The* `realloc` *function can be used to ....* "
    to near the top of p. 315, ending with the words:
    "*... original space is unchanged and the function returns a value of* `NULL`".
  - Continue reading with the first full paragraph on p. 315, starting with
    "*With the use of dynamic memory allocation...*"
  - Continue reading to the end of the section, at the end of p. 316

(2) Some additional notes: heap vs. stack

Although the textbook never mentions this word explicitly, it is important to know that all of the space allocated via dynamic memory allocation—i.e. the space allocated with malloc() and calloc()—comes from an area of memory known as ***the heap.*** The heap is also sometimes called the ***free store***.

The ***heap*** stands in constrast to the ***stack***, which is where parameters to functions and local variables inside functions are stored in memory.

To summarize: for the final exam, known these facts:

> *space for* ***local variables*** *is on the* ***stack***, and *space allocated with* ***malloc() and calloc*** *is on the* ***heap***

The reason this is important to know is that this topic will come up again in CS24 and CS32—in CS16 we are just laying the groundwork.

(3) Work with some pointer variables in Ch

As you read the next part of this handout, you should be logged into CSIL so you can work with Ch. You may want to start by creating a **`~/cs16/H15`** directory and cd'ing into it. Then, create a file in your current director called **`names.dat`** (e.g. emacs names.dat) and put the names of the three most recent California governors into it, one per line:

```
Pete Wilson
Gray Davis
Arnold Schwarzenegger
```

Then try the following, using Ch:

First, start up Ch—and optionally, simplify your Ch prompt—by doing this:

```
-bash-3.2$ ch
                                Ch
                Professional edition, version 6.1.0.13751
              (C) Copyright 2001-2009 SoftIntegration, Inc.
                     http://www.softintegration.com
/cs/faculty/pconrad/cs16/H15> _prompt="Ch> "
Ch>
```

Next try declaring three character strings, like this:

```
Ch> char s1[5] ="UCSB";
Ch> char s2[5] ="UCLA";
Ch> char s3[5] ="UCSD";
```

Note that each of these has a strlen of 4, but an array size of 5 (because of the null terminating character '\0'). More important to our discussion today, all of three of these character arrays have a **fixed length** of 5. It is not possible, once they are declared to change their length.

We can change the contents, for example by doing:

```
Ch> strcpy(s2,"UCI");
```

to change `s2` to the abbreviationn for UC Irvine, with the result that `strlen(s2)` will now return 3 instead of 4—try it!—but the size of the array `s2` is 5, and will always be 5 for the lifetime of the `s2` variable.

Strings like this are referred to as fixed length strings—their size cannot change during their lifetime. However, using pointers and dynamic memory allocation, we have another option—as discussed in Section 6.7 of your textbook.

Review: Reading strings from a file

As review, we can read values from a file using a FILE * variable. (Re-read Section 3.6 in your Etter text if you need a review of this).

First, declare a FILE * variable in Ch, like this:
(the word FILE must be in all uppercase)

```
Ch> FILE * in;
Ch>
```

You can then open a file for reading with the fopen() function, using the file name and the "r" flag:

```
Ch> in = fopen("names.dat","r");
0x863c350
Ch>
```

The hex value that gets printed reminds us that in is a `FILE *`, i.e. a pointer, the address of a place in memory where information about the file `names.dat` is stored. (If you get NULL instead, it means the file didn't open—exit from Ch, check the file and start again.)

We have a problem if we want to store names in an efficient way. As you may realize, names can vary greatly in their length. Looking just at the last two governors of California, we see that
`"Gray Davis"` would have a strlen of 10, while
`"Arnold Schwarzenegger"` would have a strlen of 21.

If we want to be *sure* that a name will fit, we could create a very large field, like this:

```
const int NAMELEN=128;
char name[NAMELEN];
```

But what value do we choose for `NAMELEN`? The value 128 gives us a very high probability that the name will fit, but results in lots of wasted space—especially if we declare an array of 1000 names like this.

On the other hand, if we choose a smaller value, there is a chance that a name will either overflow the variable, or be chopped off.

A more efficient choice is to use variable length strings, which we can do with dynamic memory allocation.

In this technique, we might still start with a huge buffer to read the name from the file—but then, we use strlen to determine the actual length of the name, and allocate an array that is exactly the right size for the name (plus the null character.)

Type in these commands to declare an integer constant (const int) for the buffer size, and a char array called buffer that can store a large string value.

```
Ch> const int BUFSIZE=1024;
Ch> char buffer[BUFSIZE];
```

Then, use the fgets function to read the first line of the file from the array:

```
Ch> fgets(buffer, BUFSIZE, in);
Pete Wilson

Ch>
```

Notice that you get an extra blank line after the name `Pete Wilson`—this is because when you use fgets to read from a file, the \n character at the end of the line comes with the input. We can see that Pete Wilson only has 11 characters, but we get strlen of 12:

```
Ch> strlen(buffer)
12
Ch>
```

In memory, the contents of the first few elements of buffer look like this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	...
'P'	'e'	't'	'e'	' '	'W'	'i'	'l'	's'	'o'	'n'	**'\n'**	**'\0'**	?	?	

We can do a simple trick to get rid of the extra newline.

- Note that strlen(buffer) returns the number of characters in the buffer, not counting the '\0' (e.g. 12 in this case)
- Those characters are numbered 0 through strlen(buffer) -1,
  i.e. 0 through 11 in this case.
- Thus, buffer[strlen(buffer-1)] refers to the last character in the string buffer (not counting the '\0').

So, we can use this line of code:

```
Ch> buffer[strlen(buffer)-1]='\0';
```

to change the contents of buffer to this:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	...
'P'	'e'	't'	'e'	' '	'W'	'i'	'l'	's'	'o'	'n'	**'\0'**	**'\0'**	?	?	

An important fact to recognize is that for a C string, the first '\0' character is the only one that matters—after that, the remaining characters are all generally ignored. So there is no problem with having two '\0' at the end of the string—the first one now terminates the string. So, what do you think strlen(buffer) will return now? Try it and see if you are correct:

```
Ch> strlen(buffer)
answer omitted—discover it for yourself!
Ch>
```

Storing a value using Dynamic Memory Allocation

The variable `buffer` is only a temporary storage place for the name—our goal is to allocate a permanent storage place that is exactly the right size for this name—using dynamic memory allocation.

That is, we want to store the name `"Pete Wilson"` on the heap, so that we only exactly 12 characters—11 for `"Pete Wilson"` and 1 for the null terminator.

To store something on the heap, there are always three steps:

1. declare a pointer variable
2. allocate space
3. store something in that space

Here are the three steps in more detail—follow along in Ch with each of these steps:

1. Set up a char * variable—we'll call it char *namePtr

   ```
   Ch> char *namePtr;
   Ch>
   ```

   This ONLY ALLOCATES THE POINTER—it doesn't allocate any space for the thing that namePtr points to. That's a *separate step*, and we'll do it next.

2. Allocate space—exactly the right amount—using this line of code:

   ```
   Ch> namePtr = (char *) malloc(strlen(buffer) + 1));
   ```

   That line of code is moderately complex, so let's break it down, starting on the right hand side inside the inner-most parens—which is exactly where C will also start to evaluate this line of code:

   - `strlen(buffer)` returns 11—the length of `"Pete Wilson"`
   - we add one because we need space for the `'\0'`, giving us 12
   - so the effect is that of malloc(12), allocating space for 12 bytes
   - malloc returns a pointer to 12 bytes as a `(void *)`, i.e. the address of 12 bytes of memory we can use.
   - we use `(char *)` to let C know that we are going to use this to store `char` data, and then we store the address in the `char *` variable `namePtr`.
3. Copy the characters into the newly allocated space. For this, we use `strcpy`:

   ```
   Ch> strcpy (namePtr,buffer);
   ```

Now, we can see the value stored at namePtr, the value of the pointer itself, and the length of the string with these lines of code typed into Ch:

```
Ch> namePtr
Pete Wilson
Ch> (void *) namePtr
0x9073730
Ch> strlen(namePtr)
11
Ch>
```

Reading a file full of names

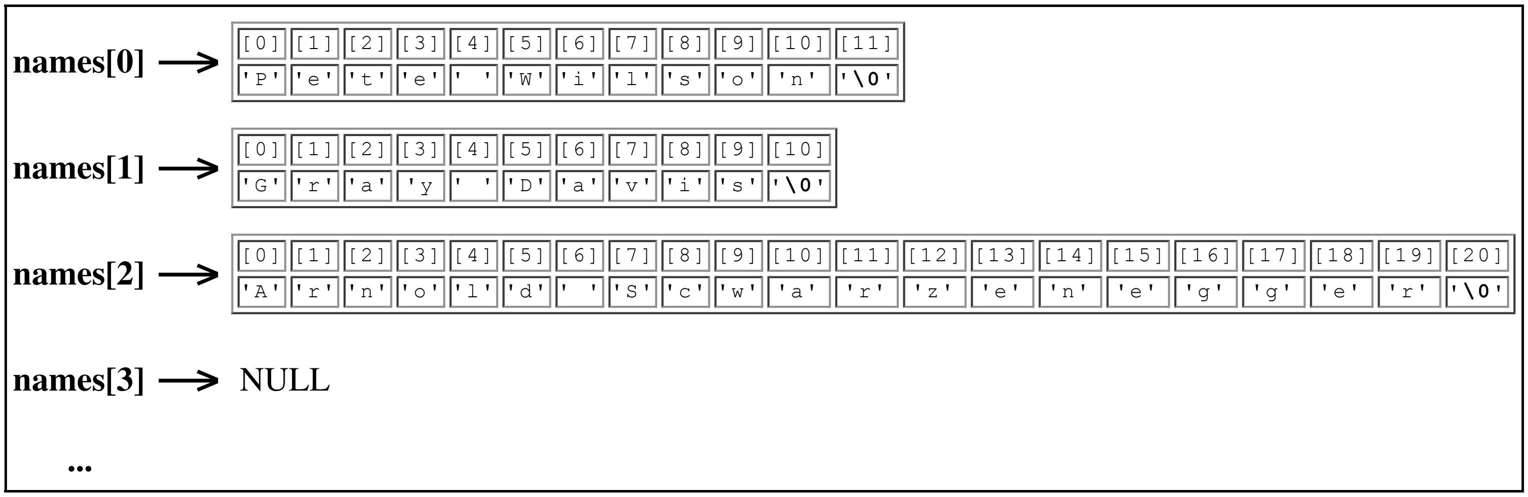
As an example of working with dynamically allocated strings, consider a file of names, where we may know in advance that we don't expect more than 10 names, but we don't know in advance how long each name will be. To store the names efficiently, we can allocate an array of `(char *)` pointers:

```
Ch> const int NAMES_CAPACITY=10;
Ch> char * names[NAMES_CAPACITY] = {0};
Ch>
```

In order to store a name in `names[0]` or `names[1]`, etc., we allocate space, using either `malloc` or `calloc`. We can each name from the file one at a time, and set names[0], names[1], etc. equal to a piece of memory that is just the right size for each of the names in the file. For example, if the names are the ones in **names.dat:**

```
Pete Wilson
Gray Davis
Arnold Schwarzenegger
```

then we want the names array to be set to pointers like this:



What is important to see here is that `names[10]` is NOT a `(char *)`, but rather an *array* of `(char *)` pointers.When we first declare the array `names,` we are allocated space for 10 pointers, but we have allocated NO SPACE to store the actual names themselves. In fact, at first, these 10 pointer values all point to 0—that is, they don't point anywhere! (We use the address value 0, sometimes called `NULL` or *nil* as way to indicate a pointer that doesn't point anywhere yet.)

To get the effect we want—i.e. the picture shown above, where `names[0]`, `names[1]`, and `names[2]` points to an array that is just the right size for each name—we have to call malloc() for each of the elements of names[] that we are going to use. In each case, we pass in the exact amount of space we want: the `strlen()` plus one for the null character.

Meanwhile `names[3]` and all the rest of the elements of the names array can remain *uninitialized pointers*—we show this by an arrow pointing to a question mark. We'll have a variable `count` that counts how many lines we read from the file—this is the occupancy of the `names` array—and we'll never look at any value outside `names[0]` through `names[count-1]`.

The code that sets up the names array this way, `mallocNames.c`, is shown on the next page.

mallocName.c code listing

The sample code for this program is shown below, and is also available online at:
http://www.cs.ucsb.edu/~pconrad/cs16/10W/homework/H15/files/mallocNames.c
or by copying files from ~pconrad/public_html/cs16/10W/homework/H15/files/*

After opening the file `"names.dat"` for reading, the mallocStrings.c program:

1. Reads all the names into the array (lines 37-58)
2. Calls a function printNames that prints out each name (lines 70-72, 83-94)
3. Before ending the program, executes a for loop to free each of the blocks of memory allocated (lines 74-77).

Try copying this code into to your ~/cs16/H15 directory and running it.

```
 1 // mallocNames.c    P. Conrad for CS16, Winter 2010 UCSB
 2 // Demonstrate reading names from a file into an array of (char *) variables
 3 // Each (char *) variable points to space allocated on heap via malloc
 4 
 5 #include <assert.h> // for assert function
 6 #include <stdio.h>
 7 #include <stdlib.h> // for atoi
 8 #include <string.h> // needed for strlen, strcpy
 9 
10 // function prototype---could also have used char **a instead of char *a[]
11 void printStringArray(char * a[], int count);
12 
13 int main(int argc, char *argv[])
14 {
15   // declare variables
16 
17   FILE *in; // input file of integers
18   const int BUFSIZE  = 1024; // how big a line can be---make it nice and big
19   char buffer[BUFSIZE]; // holds each name temporarily as we read it in
20   const int NAMES_CAPACITY = 10; // how many names we can hold
21   char * names[NAMES_CAPACITY]; // holds each name
22   int count = 0; // number of names read from file into array so far
23   int i; // for loop counter
24 
25   // open the file
26 
27   in = fopen("names.dat","r"); // use cmd line arg to open file
28 
29   if (in==NULL)
30     {
31       perror("Could not open names.dat");
32       return 2;
33     }
34 
35   // STEP 1: READ ALL NAMES FROM FILE INTO char * ARRAY
36 
37   fgets(buffer,BUFSIZE,in); // try to read a line
38   while (!feof(in) && !ferror(in)) // not end of file, and no error
39     {
40       // We successfully read a line! So store it in the array
41       // First make sure there is room
42       if (count==NAMES_CAPACITY)
43         {
44           printf("Error: not enough room in the array\n");
45           printf("Increase NAMES_CAPACITY and recompile\n");
46           return 3;
47         }
48 
49       buffer[strlen(buffer)-1] = '\0';       // (1) Remove the newline
50       names[count] = (char *) malloc(strlen(buffer)+1); // (2) Allocate space
51       strcpy(names[count], buffer); // (3) Copy the data
52 
53       // THEN increment count (not the other way around)
54       count++;
55 
56       // try to read another line
57       fgets(buffer,BUFSIZE,in);
58     }
59 
60   // See if there is an error
61 
62   if (ferror(in))
63     {
64       printf("An unknown error occurred with names.dat\n");
65       return 4;
66     }
67 
68   assert(feof(in));  // We should be at end of file.
69 
70   // STEP 2: PRINT OUT ALL THE NAMES
71 
72   printStringArray(names, count);
73 
74   // STEP 3: FREE ALL THE MEMORY ALLOCATED WITH malloc() BY CALLING free()
75 
76   for (i=0;i<count;i++)
77     free(names[i]); // free each name that was allocated
78 
79   return 0; // success!
80 
81 }
82 
83 void printStringArray(char * a[], int count)
84 {
85   int i;
86   printf("Here are some strings: \n");
87 
88   // The notation \" inside a string means "actually print a double quote"
89 
90   for (i=0; i<count; i++)
91     printf("a[%d]=\"%s\"\n", i, a[i]);
92 
93   printf("\nThat is all.\n");
94 }
```

The third step in this program—freeing the memory (lines 74-44)—deserves a whole separate discussion.

T-shirt slogan: free the mallocs!

Some would say that freeing all the memory in this program is not strictly necessary—when a program ends, all the memory allocated with malloc or calloc gets automatically freed. And, strictly speaking, that is absolutely correct.

However, learing to free the memory you allocate it is is very good programming habit to develop.

What many students don't realize is that the small programs we write for class assignments are not typical of real world code, in the following sense—most programming assignment code is typically run on small data sets, for only a few seconds or at most minutes.

Real world programs often run for hours, days, week, months or years—think of the code for a web server, for example—and run on very large amounts of data, over time. When a program runs for a long time and/or on a large data set, if that program continuously calls `malloc()`/`calloc()`, and never calls `free()` to free up any of the memory used, eventually the heap will run out of space. This condition is sometimes called a ***memory leak*** because it is "as if" the amount of free memory is simply leaking out onto the floor.

So, wise programmers develop the following habit—anytime you write a line of code that uses `malloc()` or `calloc()`—consider also writing a line of code that calls `free()` as soon as that piece of memory is no longer needed.

You don't always need to write the code to free() every malloc()—but you ***should always consider whether you need to or not,*** and if you don't, you should have a clear understanding of why it isn't necessary.

We'll may talk more about this in lecture, as/when time permits. If we don't get to it, it is likely to be a topic in later courses such as CS24 and CS32—as it turns out, in C++ the operations known as `new` and `delete` in C++ have the same role as the functions `malloc()` and `free()` have in C, and all the same issues come up.

One more note: don't call `free()` twice on the same piece of memory—this can mess up the heap and cause strange errors to occur.

End of handout for H15