Using Strings and Pointers in C
static, local, and dynamic memory

21 March 2010

1. A Common malloc/free Error in C Programming

In our grading of assignments, we have seen a few malloc/free errors appear regularly. These errors often involve working with strings of characters. This document starts by discussing a common error, and then explains, in more general terms, strings and memory in C.

Consider the following code fragment:

```
void setInvalidArg(char *arg)
{
    invalidArg = malloc(sizeof(arg));
    strcpy(invalidArg, arg);
}
```

This function is hoping to store a string in a global variable called `invalidArg` and uses malloc to get enough space to store the sequence of chars. The problem is that the `sizeof` operator tells the amount of space used by its argument. In the code fragment shown, the argument is a pointer to a character. A pointer is a variable that holds the address of data. On a 32-bit processor, this variable is usually four bytes long (32 bits), and on a 64-bit processor, the variable may be eight bytes long (64 bits). In neither case does the `sizeof` operator find the number of chars stored in the place pointed to by arg.

The correct way to use a char pointer to get the amount of space needed to store a string is to use the C library function `strlen` as in

```
    invalidArg = malloc(1 + strlen(arg));
    if ( invalidArg == NULL )
        report_error_and_exit(...)
```

The function `strlen` goes to the memory pointed to by arg, counts the number of chars before the nul char and returns that count. You have to add 1 to that number because the new space you get has to have enough elements for the chars and for the terminating nul char.

Notice also that the corrected version checks the return value from malloc before using that address. If malloc fails, it returns a NULL pointer. An attempt to copy a string into the space pointed to by a NULL pointer will generate a segmentation violation and crash the program.

2. Storing Strings in C

You are writing a C program that needs to store some strings of text. What are your options? How do you decide which technique to use?

In C, a string is a sequence of chars stored in consecutive memory locations and terminated with a *nul* character -- a char with numeric code of zero. A char in C is just a number, the numeric code for that character. Therefore a string is a sequence of memory locations holding numeric codes for chars terminated with a memory location holding the value 0, also written as '\0' .

3. Types of Storage for Strings

C supports three types of storage: global, local, and dynamic. There are no other choices, so understanding the options is not too complicated.

global — A global string is an array of chars that persists for the duration of program execution and is available to all functions that appear below the variable definition or declaration. Here is an example;

```
    int func1(){
      ...
    }

    char name[100] = "Imogene";     /* created when program loads into memory */

    int func2(){
      ...
    }

    int func3(){
      ...
    }
```

In the example shown above, functions func2 and func3 can use the array called name, but func1 does not know about it. In practice, one rarely mixes definitions of global variables with function definitions. Usually the variables are defined first.

local A local string is an array of chars that persists within a function until the function exits. A local string is available to the function in which it is defined. Furthermore, the function in which the string lives can pass the address of that string to other functions it calls. But the space for the local string is deallocated automatically when the function exits. Here is an example:

```
   void func1()
   {
       char name[100];        /* created when function is called */

       strcpy(name, "Smith");
       strcat(name, ", John Q.");
       printf("The name is %s\n", name);
   }
```

In this example, the string called name is defined within the function. The 100-chars of space is allocated when the function starts executing and is deallocated when the function exits. In each line of the function the string is passed, by reference, to other functions: strcpy, strcat, and printf. These three functions receive the address in memory of the array. By receiving the address, these functions can modify the contents of the array, if they like. The first two functions, strcpy and strcat, do modify the array, while the third function, printf, does not modify the string.

Here is another example:

```
   void greet(int n)
   {
       char ans[n+1];
       printf("What is your name? ");
       fgets(ans, n+1, stdin);
       printf("I am pleased to meet you, %s!\n", ans);
   }
```

This example shows that the size of a local array can be set when the function is called. This feature of C combines the simplicity and efficiency of local variables with the flexibility of dynamic memory. The difference between dynamically sized local storage and malloc'ed storage is that local storage is automatically deallocated when the function exits.

dynamic A dynanic string is an array of chars allocated when needed by the programmer during the running of the program and deallocated by the programmer when the array is no longer needed. To create the dynamically allocated string, the programmer writes:

```
   char *make_name(char *first, char *last)
   {
        char *result;

        /* get space for both names, and ", " and the nul at the end */

        result = malloc( strlen(first) + strlen(last) + 2 + 1 );
        if ( result == NULL )
```

```
                fatal("out of memory");        /* print message and exit */
            sprintf(result, "%s, %s", last, first);
            return result;
        }
```

In this example, the function called make_name is passed two strings. The function then allocates enough memory to hold the two names with a comma and space between them. Then the function writes the full name into the new space and returns the address of this dynamically created array.

A dynamic string differs from a global string because it comes into being when the program calls the malloc() function and is deallocated when the program calls the free() function. On the other hand, a global string comes into being when the program starts and ceases to exist when the program ends.

A dynamic string differs a local string in that a local string is allocated when the function is called and is deallocated when the function exits. A dynamic string does not depend on the program starting or a function starting to appear, and it does not depend on a program exiting or a program returning to vanish. Instead, a dynamic string appears when you ask for it with malloc and vanishes when you call free.

4. When to Use Each Type of String

The three types of strings differ in how long each lives and how visible each type is.

global — A global string lives for the duration of the program and is visible to every function in the file in which the string is defined and can even be visible to functions in other files if those other files declare the string to be `extern` .

Therefore, use global strings for any sequence of chars if that string meets these needs:

1. You want the storage to be around for the whole program
2. You want to refer to that storage by name from any function
3. When you know in advance how much space you will need

Global variables are usually a bad idea. Other functions can modify the contents. Another programmer could add a function to the program that changes the array in ways your existing code does not expect.

But some global variables are useful and good design. In particular, static global variables are local to a file and are a good way to store state for a module. If a file implements a particular type of object, that file may define `static char n[LEN];` to make an array that is visible only to functions in that file.

local — A local string appears when the function starts and vanishes when the function exits. Use local strings when:

1. You need space for scratch work inside a function
2. You do not need the storage after the function is done
3. You do not know how much space you need until the function is called

The contents of the string will not exist after the function exits, so only use local strings for storage needed by the function. As the example above shows, you can use a local string to make a copy of a string or to combine a few strings into one longer string, pass that longer string to another function, and once done with the scratch space, just exit the function and the compiler arranges for that space to be deallocated.

dynamic — A dynamic string is allocated (instantiated) when you use malloc() to create the string, and the string is deallocated when you call free(). There are two main values to dynamic strings. Use dynamic strings when:

1. You need to store a string for an unknown length of time
2. You do not know how much memory you need until the program is running

For example, if you are writing a program to read in a list of words, sort the list, and then print out the list, you do not know in advance how many words a user might type in, nor do you know how long those words will be. Furthermore, you may write one function to read in the data and another function to sort the list, and a third function to print out the results. In this case, the storage has to be around after the first function is done reading in the data.