Lecture 08 - Strings in C (last updated: 2023-10-02)

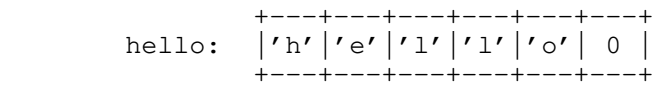
Char arrays as strings

C does not have a separate string type. Strings are represented using char arrays ending with 0. We can declare a string "hello" on the stack like this:

```
char hello[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Recall that character literals like 'h' and 'e' are just notation for the numbers 104 and 101. Also recall that '\0' is another notation for 0. The byte '\0' at the end of a string is commonly referred to as a null character or a null terminator.

The stack array hello looks like this in memory:

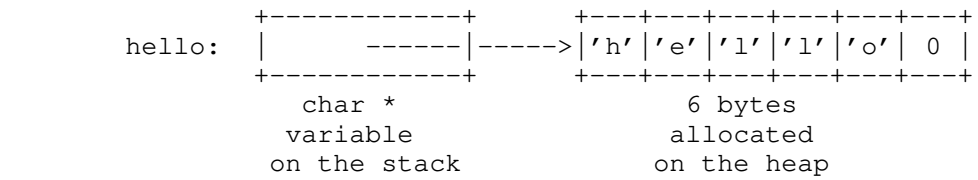


If we want a string "hello" declared on the heap, we can do the following:

```
char *hello = malloc(6);

hello[0] = 'h';
hello[1] = 'e';
hello[2] = 'l';
hello[3] = 'l';
hello[4] = 'o';
hello[5] = '\0';
```

We will then have the following in memory:



In both stack and heap cases, we need 6 bytes to store a string of 5 characters because of the null terminator at the end.

We can print the string using printf() with the "%s" format specifier:

```
printf("%s\n", hello);
```

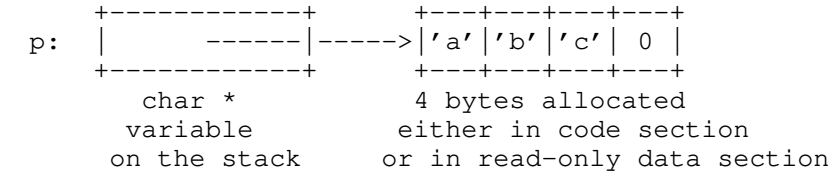
"%s" tells the printf() function to expect a variable of type 'char *' (i.e., a pointer to a char). The printf() function will print out each character stored in memory locations starting from hello to the byte right before the null terminator. (Or more precisely, it prints the character whose ASCII code corresponds to the number stored in the memory locations.) In the stack case, hello is an array name which gets converted to a pointer to 'h'; in the heap case, hello is indeed a pointer to 'h'.

String literals

Consider the following declaration:

```
char *p = "abc";
```

It results in the following in memory:



Let's unpack what's going on there one thing at a time. First, the string literal "abc" is an expression, and its type is an array of 4 chars. That is, it has the same type as the following array a:

```
char a[4];
```

Both the array a and the string literal "abc" are of type 'char [4]' (i.e., an array of 4 chars), but the difference is that the string literal "abc" does not have a name associated with it.

There is another difference. C decrees that the underlying array for a string literal is read-only. In most systems, string literals are stored either in the code section along with the program code or in a special read-only data section located between code section and static data section. Attempts to modify string literals will likely result in segmentation fault:

```
*p = 'A';  // result undefined (probably segmentation fault)
```

Now, how then did the pointer p get to point to the first element of "abc"? Recall that an array name (like the array a above) will be converted to a pointer to the 1st element in most expressions. The same applies to the string literals. In the declaration of p above, the string literal expression "abc", whose type is 'char [4]', will be converted to a 'char *' pointing to the first byte of the array, and then gets assigned to the pointer variable p.

All of the following boolean expressions evaluate to true. (The assert() function will abort the program execution if the given boolean value is 0; it's handy for debugging.)

```
assert("abc"[1]     == 'b');
assert(*("abc" + 1) == 'b');  // note that this follows from above, per GUT

assert(*"abc"   == 'a');
assert("abc"[3] == '\0');
```

Recall that named arrays do not get converted to pointers when you apply sizeof() operator. This is the case for string literals as well:

```
assert(sizeof("hello") == 6);  // string literals remain arrays in sizeof()
```

In addition, a string literal does not get converted to a pointer when it is used as an initializer in an array declaration:

```
    char a[4] = "abc";  // short-hand for: char a[4] = {'a', 'b', 'c', '\0'};
```

Note that the declaration above declares a stack array a. It is very different from the following declaration of a pointer p that we have examined above:

```
    char *p = "abc";
```

String functions in the Standard C Library

The strlen() function returns the length of a string, not including the null terminator:

```
    int strlen(char *s) {
        int i = 0;
        while (*s++) {
            i++;
        }
        return i;
    }
```

The strcpy() function copies a string into another array:

```
    void strcpy(char *t, char *s) {
        int i = 0;
        while ((t[i] = s[i]) != 0) {
            i++;
        }
    }
```

We can also write it using pointer notation like this:

```
    void strcpy(char *t, char *s) {
        while ((*t = *s) != 0) {
            s++;
            t++;
        }
    }
```

Or even like this:

```
    void strcpy(char *t, char *s) {
        while ((*t++ = *s++) != 0)
            ;
    }
```

Make sure you understand how each of these implementations works.

They can be used like this:

```
    char a[4];
    strcpy(a, "abc");

    assert(strlen(a) == 3);
    assert(strcmp(a, "abc") == 0);  // strcmp() returns 0 if the two strings
                                    // contain the same characters
```

Note that '\0' marks the end of a string, even if it is in the middle of an array. For example:

```
    char a[6];
    strcpy(a, "hello");
    a[4] = 0;

    assert(strlen(a) == 4);
    assert(strcmp(a, "hell") == 0);
```

The Standard C Library provides implementations for all of the functions above, and many more.

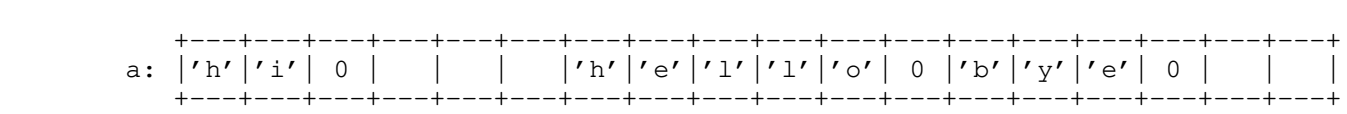
Pointers to pointers

Say we wanted an array of strings. What would that look like?

Here is one possibility:

```
    char a[3][6] = { "hi", "hello", "bye" };
```

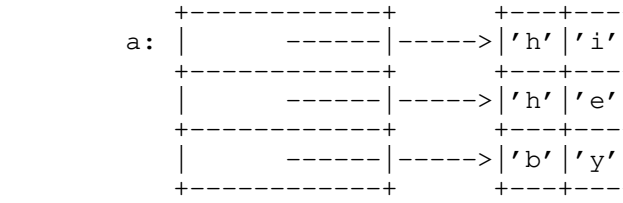
It declares an array of 3 elements, where each element is an array of 6 chars. It looks like this in memory:



A more memory-efficient approach would be the following:

```
    char *a[] = { "hi", "hello", "bye" };
```

It declares an array of 3 pointers to char, which looks like this in memory:



This is a common paradigm in C for declaring an array of strings. In fact, command line arguments are passed to a program in this way. If we define the main() function as:

```
    int main(int argc, char **argv) { ... }
```

instead of:

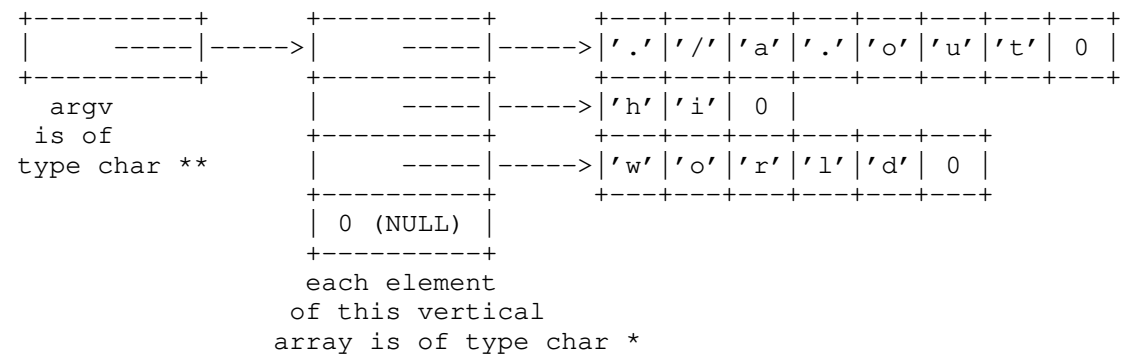
```
    int main(void) { ... }
```

we can access the command line arguments through argc ("argument count") and argv ("argument vector").

For example, when we run:

```
    ./a.out hi world
```

the following data structure is passed to the main() function:



Each element in the vertical array is of type 'char *' (because they point to the strings on the right.) The argv variable, which points to the first element of the vertical array, is therefore of type 'char **', a pointer to 'char *'.

The expression *argv is the first element of the vertical array. The expression *(argv + i) or argv[i] is the i-th element of the vertical array. The expression *argv[i] would be the first character of the string that argv[i] points to. The expression argv[i][j] would be the j-th character of the string that argv[i] points to.

The argc variable is set to 3, the number of command line arguments, plus one for the program name itself. The argv array has 4 elements, which is one more than argc, because there is a NULL pointer at the end. That is, argv[argc] is always NULL. This NULL pointer comes in handy when we iterate through this array (as we will see below.) Note that this NULL pointer is NOT the null terminator. The value is the same (i.e., zero), but this is of type 'char *', not char. It takes up 8 bytes, not 1 byte.

To put it all together, here are a few different ways to print all command line arguments, not including argv[0] (the program name itself):

```
int main(int argc, char **argv) {
    for (i = 1; i < argc; i++)
        printf("%s\n", argv[i]);
}

int main(int argc, char **argv) {
    while (--argc > 0)
        printf("%s\n", *++argv;);
}

int main(int argc, char **argv) {
    argv++;
    while (*argv)
        printf("%s\n", *argv++);
}
```

Make sure you understand how each of these implementations works!