Strings in C
A Programmer's Nightmare

Benjamin Brewster

C Strings

- Strings in C are sequences of characters contiguously stored
  - Not a native type like `int` or `float` in more advanced languages

- A “string” terminates with the null character
  `\0`

- That’s *it!* Any further programmatic use of strings requires functions and procedures that work within this format

Displaying C Strings with Formatted Printing

- Formatted means numbers correctly printed with text

- Formatted printing is done with:
  - `printf()` Prints to standard out
  - `sprintf()` Prints to a string (a char array)
  - `fprintf()` Prints to a file

- These functions look for null terminators to know when to stop

Basic C String Functions

Warning: do not use the == operator!

- Use the string library functions:
  - `strcmp()` Compares two strings for equality
  - `strlen()` Returns the length of the string in characters, not including null terminator
  - `strcpy()` Copies one string into another
  - `strcat()` Returns one string that is a concatenation of itself with another string
- n-character versions:
  - `strncpy()` Copy only n characters - won’t null-terminate a full array, or actually prevent you from over-writing an array
  - `strncat()` Appends only a portion of a string to another

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?
- Because this one difference shows how close C is to the underlying memory management being performed by UNIX

Declaring C Strings

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- Are they really the same? And why do we care in OS?
- Because this one difference shows how close C is to the underlying memory management being performed by UNIX
- I.e. you need to know this, because otherwise you'll break all the things and not know why

Declaring C Strings – Method 1

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- At compile time, creates a sequence of bytes in the **read-only initialized data segment** portion of memory with the contents `"my string"`
- During execution, creates a pointer on the **stack** (automatic variable) called `mystring` that points to the read-only sequence of characters in the **data segment**
- `mystring` can be pointed to other addresses (it doesn’t hold chars by itself, as it’s a pointer)

Declaring C Strings – Method 1 – Example

`mystring` is a pointer put on the stack

`"my string"` is a string literal defined and stored in the read-only portion of the data segment

Index 3 bytes off of where `mystring` is pointing too, then change whatever is there to ‘Q’…

```
#include <stdio.h>

void main()
{
        char* mystring = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

Result:

Declaring C Strings – Method 1 – Example

`mystring` is a pointer put on the stack

```
#include <stdio.h>

void main()
{
        char* mystring = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

`"my string"` is a string literal defined and stored in the read-only portion of the data segment

Result:

```
Var is: my string
Segmentation fault (core dumped)
```

… except you can’t do that, because your program cannot change memory in the read-only portion of the data segment

Declaring C Strings – Method 2

- Three ways of declaring the same string

```
1. char* mystring     = "my string";
2. char  mystring[]   = "my string";
3. char  mystring[20] = "my string";
```

- During execution, creates space for 10 bytes on the **stack** as an automatic variable, names that variable `mystring`
- Puts **`"my string"`** into the variable `mystring` with a null terminator after it
- The variable `mystring` is editable, as it is an array

Declaring C Strings – Method 2 – Example

mystring is an array

```
#include <stdio.h>

void main()
{
        char mystring[] = "my string";
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
}
```

Result:

```
Var is: my string
Var is: my Qtring
```

Declaring C Strings – Method 3

- Three ways of declaring the same string

```
1. char* mystring      = "my string";
2. char  mystring[]    = "my string";
3. char  mystring[20] = "my string";
```

- Creates space for 20 bytes on the **stack** as an automatic variable, names that variable `mystring`
- Puts **`"my string"`** into the variable `mystring` with a null terminator after it
- The variable `mystring` is editable, as it is an array

String Literals (Again)

- What’s wrong with this code:

```
char* mystring = "my string";
strcpy(mystring, "AA string");
printf(mystring);
```

String Literals (Again)

- What’s wrong with this code:

```
char* mystring = "my string";
strcpy(mystring, "AA string");
printf(mystring);
```

- String literals cannot be changed in C - they are initialized in the **read-only** section of the **initialized data segment**
- When is this error caught?
  - Only at run-time, as a seg-fault; this compiles just fine

Buffer Overrun

- What’s wrong with this?

```
char fiveStr[5] = “five";
strcpy(fiveStr, “five6");
printf(fiveStr);
```

Buffer Overrun

- What’s wrong with this?

```
char fiveStr[5] = “five";
strcpy(fiveStr, “five6");
printf(fiveStr);
```

- “`five6`” is too long to store in `fiveStr`

- When is this error caught?
  - Never!
  - Unless something you needed is overwritten and a segfault occurs because a just-accessed pointer no longer points to where it was supposed to!

Fully Initializing C String Arrays

```
char mystring[20];
strcpy(mystring, "my string");
```

m	**y**		**s**	**t**	**r**	**i**	**n**	**g**	**\0**	***B***	***O***	***G***	***U***	***S***	***B***	***O***	***G***	***U***	***S***

```
printf("%s", mystring);
```

How do we deal with this uninitialized data?

Result:

```
my string
```

What’s In that Uninitialized Data?

```
$ cat cstring-array-unint.c
#include <stdio.h>
#include <string.h>

void main()
{
  int i = -5;
  char mystring[20];

  strcpy(mystring, "my string");

  printf("Char => Int :: ASCII Table Lookup\n");

  for (i = 0; i < 19; i++)
    printf("%c    => %d\n", mystring[i], mystring[i]);
}
$ gcc -o cstring-array-unint cstring-array-unint.c
```

```
$ cstring-array-unint
Char => Int :: ASCII Table Lookup
m    => 109
y    => 121
     => 32
s    => 115
t    => 116
r    => 114
i    => 105
n    => 110
g    => 103
    => 0
@    => 64
    => 0
    => 0
    => 0
    => 0
    => 0
@    => 64
W    => 87
▒    => -26
```

Printing chars as ints is a great way to debug C string arrays!

ASCII -26 summons Cthulu

Initializing C String Arrays - The Bad

- Depending on how you declare them, C string arrays may be full of uninitialized data - it’s best to clear them before use

- What happens if we somehow remove the automatic null terminator?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20];
        strcpy(mystring, "my string");

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}
```

Uninitialized!

Initializing C String Arrays - The Bad

- Depending on how you declare them, C string arrays may be full of uninitialized data - it's best to clear them before use

- What happens if we somehow remove the automatic null terminator?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>
void main()
{
        int i = -5;
        char mystring[20];
        strcpy(mystring, "my string");
        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);
        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';
        printf("Var is: %s\n", mystring);
}
$ gcc -o cstring-array cstring-array.c
$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring#@#####▒t▒
```

Uninitialized!

Different almost every time it runs, as memory is used

Initializing C String Arrays - The Suspicious

- Depending on how you declare them, C string arrays may be full of uninitialized data - it’s best to clear them before use

Seems to initialize entire array to \0 but is this portable?

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20] = "my string";

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}

$ gcc -o cstring-array cstring-array.c

$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring##########
```

Initializing C String Arrays - The Preferred

- Depending on how you declare them, C string arrays may be full of uninitialized data - it’s best to clear them before use

Fully Initialized

```
$ cat cstring-array.c
#include <stdio.h>
#include <string.h>

void main()
{
        int i = -5;
        char mystring[20];
        memset(mystring, '\0', 20);

        strcpy(mystring, "my string");

        printf("Var is: %s\n", mystring);
        mystring[3] = 'Q';
        printf("Var is: %s\n", mystring);

        mystring[9] = '#';
        mystring[19] = '\0';
        for (i = 10; i < 19; i++)
                if (mystring[i] == '\0')
                        mystring[i] = '#';

        printf("Var is: %s\n", mystring);
}
$ gcc -o cstring-array cstring-array.c
$ cstring-array
Var is: my string
Var is: my Qtring
Var is: my Qtring##########
```

Meanwhile Back on the Ranch…

- C continues to provide dangerous string functions

`strtok()` :: String tokenizer

- Splits strings into chunks
- Makes your hair fall out
- Maxes out your credit cards
- Unfriends all your social media friends
- Sometimes the best/only tool for the job :/

strtok Example

```
char input[18] = "This.is my/string";

char* token = strtok(input, " ./");
```

This

```
token = strtok(NULL, " ./");
```

is

```
token = strtok(NULL, " ");
```

my/string

Changing the delimiter as strtok() tokenizes the string is neat

A Major `strtok()` Drawback *(the first of many)*

```
char* input = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Fails miserably. Why?

A Major `strtok()` Drawback *(the first of many)*

```
char* input = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Fails miserably, crashing on execution: Why?
  - Because `input` is a string literal, and `strtok()` is about to mess with your strings

```
#include <stdio.h>
#include <string.h>

void main()
{
        char input[50];
        char* token = 0; // Set null pointer
        int inputsize = -5;
        int currChar = -5;

        memset(input, '\0', 50);
        strcpy(input, "A.B C/D");
        inputsize = strlen(input);

        printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
        printf(" = \"%s\", token: \"%s\"\n", input, token);
        token = strtok(input, " ./");
        printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
        printf(" = \"%s\", token: \"%s\"\n", input, token);
        token = strtok(NULL, " ./");
        printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
        printf(" = \"%s\", token: \"%s\"\n", input, token);
        token = strtok(NULL, " ./");
        printf("Input: "); for (currChar = 0; currChar < inputsize; currChar++) printf("%2d ", input[currChar]);
        printf(" = \"%s\", token: \"%s\"\n", input, token);
}
```

strtok Example Results

```
Input: 65 46 66 32 67 47 68  = "A.B C/D", token: "(null)"
Input: 65  0 66 32 67 47 68  = "A", token: "A"
Input: 65  0 66  0 67 47 68  = "A", token: "B"
Input: 65  0 66  0 67  0 68  = "A", token: "C‶
```

- `input` gets jacked up by `strtok()` as the delimiters encountered during parsing get nulled
- Further, this can only work because `strtok()` keeps a hidden static variable in the data segment up to date while parsing

Further strtok Horrors

- Not only does strtok() modify the input…

  (You don’t even specify which string to tokenize past the first call! Hidden vars!)

```
char input[18] = "This.is my/string";
char* token = strtok(input, " ./");
token = strtok(NULL, " ./");
token = strtok(NULL, " ");
```

- Mixing calls of strtok() between different strings is not allowed because it can only process ONE string with its hidden variables!
  - But there is a strtok_r() that achieves re-entrancy, allowing the mixing of calls, by requiring you pass in a pointer to a temp variable for it to use

Horrors Explained

- This mixing of `strtok()` calls is easy to do on accident in a large program, especially with functions involved:

```
strtok(input1, …)
function()
    strtok(input2, …)
strtok(input1, …)
```

- The solution is to simply use a more modern language with a string type

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Declare an array of pointers, each of which points to a string; each of these pointers can be pointed at either array names *or* string literals

Remember that an array name is a pointer to the first element’s address in memory

Arrays of Pointers to Strings - Example

```
#include <stdio.h>
#include <string.h>

void main()
{
  int currElem = -5;
  int numElems = 3;
  char* mystring[numElems];
  char myarray[10];

  strcpy(myarray, "1ARRAY");

  printf("Size of char*: %d\n", sizeof(char*));
  printf("Size of one array element: %d\n", sizeof(mystring[0]));
  printf("Size of all array elements: %d\n", sizeof(mystring));
  printf("Number of elements in array: %d = %d\n", sizeof(mystring) / sizeof(mystring[0]), numElems);

  //strcpy(mystring[0], "strcpy string");        // Causes seg fault, that's a pointer!
  //printf("mystring[0]: %s\n", mystring[0]);
  mystring[0] = "string literal";                 // Set the first pointer to point to the address of a string literal
  printf("mystring[0]: %s\n", mystring[0]);        // (which is the address of the literal's first element)
  mystring[0] = myarray;                          // Set the first pointer to point to the name of a C string array
  printf("mystring[0]: %s\n", mystring[0]);        // (which is the address of the array's first element)
}
```

Results:

```
Size of char*: 8
Size of one array element: 8
Size of all array elements: 24
Number of elements in array: 3 = 3
mystring[0]: string literal
mystring[0]: 1ARRAY
```

Combining Declaration Methods

- What does this mean:

```
char* mystring[3];
```

Declare an array of pointers, each of which points to a string; each of these pointers can be pointed at either array names *or* string literals

Remember that an array name is a pointer to the first element's address in memory

We can change where the pointers point, but how do we create new strings for this new array to hold?

Dynamically Allocating a String

- To create a string variable dynamically, and thus use it like an array, use `malloc()` and `free()`:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main()
{
  char* mystring;
  char* literal = "literal";

  mystring = malloc(20 * sizeof(char));

  if (mystring == 0)
    printf("malloc() failed!\n");

  memset(mystring, '\0', 20);

  sprintf(mystring, "yay! %s\n", literal);
  printf("%s", mystring);
  mystring[3] = 'Q';
  printf("%s", mystring);

  free(mystring);
}
```

Note that `char* mystring` is editable!

```
$ gcc -o malloctest malloctest.c
$ malloctest
yay! literal
yayQ literal
```

Malloc Memory Leaks

- If you don’t free dynamically allocated memory, it still takes up space
- If you have a long-running program, like a server process, this could eventually use up all of your memory
- Process memory is normally all freed automatically when a process is terminated
  - At least in UNIX, Windows, etc. - some real-time operating systems don’t!

Malloc Memory Leaks

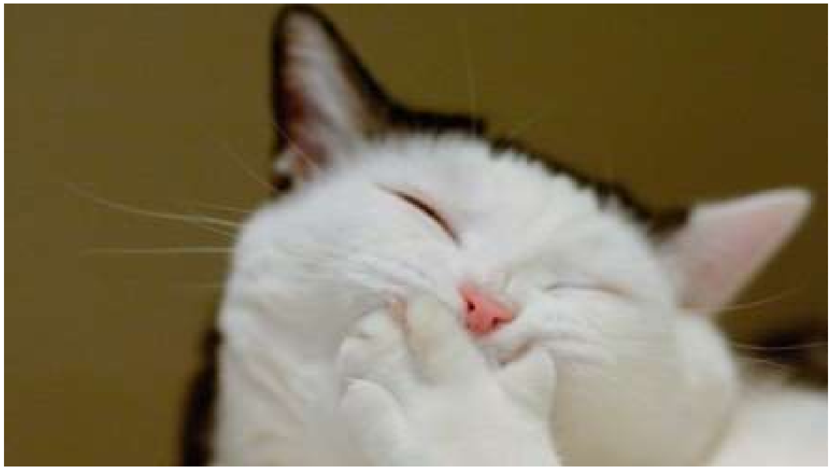
- Here’s a classic way to hide and cause a leak:

```
char* mystring = malloc(20 * sizeof(char));
...
mystring = "hello";
```

- This leaks because you no longer have the start address of the dynamically allocated space; `mystring` now points to a string literal

```
free(mystring);  // And if you try this later, it fails spectacularly
```

Spectacular Failing



Same program, but let's just put this right in here…

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main()
{
  char* mystring;
  char* literal = "literal";

  mystring = malloc(20 * sizeof(char));

  if (mystring == 0)
    printf("malloc() failed!\n");

  memset(mystring, '\0', 20);

  sprintf(mystring, "yay! %s\n", literal);
  printf("%s", mystring);
  mystring[3] = 'Q';
  printf("%s", mystring);
  mystring = "test\n";
  free(mystring);
}
```

Spectacular Failing Results

```
$ malloctest
yay! literal
yayQ literal
*** Error in `malloctest': free(): invalid pointer: 0x0000000000400805 ***
======= Backtrace: =========
/lib64/libc.so.6(+0x7d053)[0x7fc92849c053]
malloctest[0x40074a]
/lib64/libc.so.6(__libc_start_main+0xf5)[0x7fc928440b15]
malloctest[0x4005d9]
======= Memory map: ========
00400000-00401000 r-xp 00000000 00:39 3238103636           /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
00600000-00601000 r--p 00000000 00:39 3238103636           /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
00601000-00602000 rw-p 00001000 00:39 3238103636           /nfs/stak/faculty/b/brewsteb/tempdir/malloctest
01195000-011b6000 rw-p 00000000 00:00 0                    [heap]
7fc924000000-7fc924021000 rw-p 00000000 00:00 0
7fc924021000-7fc928000000 ---p 00000000 00:00 0
7fc928209000-7fc92821e000 r-xp 00000000 fd:02 16777347     /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92821e000-7fc92841d000 ---p 00015000 fd:02 16777347     /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841d000-7fc92841e000 r--p 00014000 fd:02 16777347     /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841e000-7fc92841f000 rw-p 00015000 fd:02 16777347     /usr/lib64/libgcc_s-4.8.5-20150702.so.1
7fc92841f000-7fc9285d6000 r-xp 00000000 fd:02 16811513     /usr/lib64/libc-2.17.so
7fc9285d6000-7fc9287d6000 ---p 001b7000 fd:02 16811513     /usr/lib64/libc-2.17.so
7fc9287d6000-7fc9287da000 r--p 001b7000 fd:02 16811513     /usr/lib64/libc-2.17.so
7fc9287da000-7fc9287dc000 rw-p 001bb000 fd:02 16811513     /usr/lib64/libc-2.17.so
7fc9287dc000-7fc9287e1000 rw-p 00000000 00:00 0
7fc9287e1000-7fc928802000 r-xp 00000000 fd:02 16811685     /usr/lib64/ld-2.17.so
7fc9289cf000-7fc9289d2000 rw-p 00000000 00:00 0
7fc9289ff000-7fc928a02000 rw-p 00000000 00:00 0
7fc928a02000-7fc928a03000 r--p 00021000 fd:02 16811685     /usr/lib64/ld-2.17.so
7fc928a03000-7fc928a04000 rw-p 00022000 fd:02 16811685     /usr/lib64/ld-2.17.so
7fc928a04000-7fc928a05000 rw-p 00000000 00:00 0
7ffe32184000-7ffe321a5000 rw-p 00000000 00:00 0            [stack]
7ffe321cd000-7ffe321cf000 r-xp 00000000 00:00 0            [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0    [vsyscall]
Aborted (core dumped)
```