Storage of String Literals using gcc in a Unix Environment

Chae Jubb
`ecj2122@columbia.edu`

28 November 2014

1 Introduction

Most, if not all, C programs use string literals to communicate with the user who runs the resulting program. They are probably most often seen in print statements as format specifiers for functions such as `fprintf` and related. The use of string literals, however, is not limited to this family of print statements. String literals may be assigned to a `char` pointers or used as shorthand for an initializer for `char` arrays, among others.

Because of the vast differences in these two use cases, we investigate gcc's handling of string literals when used in these ways. How does gcc handle the differences in write permissions? How does it optimize for size vs. speed of the resulting executable? This paper attempts to answer these and other questions.

2 Background

Below I present a background description of both string literals and the layout of gcc output executables. Both will be helpful in understanding the following analysis.

That analysis is done using Ubuntu 14.04 64 bit version with gcc 4.8.2.

2.1 Description of String Literals

String literals are described in §3.1.4 of the C89 standard as a "sequence of zero more more multibyte characters enclosed in double-quotes". These string literals are required to have static storage duration and type "array of `char`". Additionally, identical string literals need not be distinct, meaning the compiler is permitted to combine and isolate them. However, we cannot edit string literals. An attempt to modify a string literal results in undefined behavior. In practice, this usually amounts to a segmentation fault.

We also see the C standard describe the `char` array initialization in §3.5.7. A `char` array may be initialized by a string literal, which is optionally enclosed in braces. If there is enough room or the size of the array unknown, a terminating null character will be copied into the array after the characters that make up the string literal.

2.2 Description of ELF

The Executable and Linkable Format file format is used as the standard binary format for Unix executables. At a high level, the executable consists of many sections which are loaded into specific memory segments upon execution. (These segments will be marked with Read/Write/Execute permissions depending on the section loaded.) For our purposes, the following segments are relevant:

- `.text` This section contains the program code. It is usually loaded in a segment with Read and Execute permissions.
- `.rodata` This section contains read-only data. As the name suggests, it is usually loaded in a segment with only Read permissions.
- `.data` This section contains writable data. Initialized static variables fall into this section, which

is usually loaded into a segment with Read and Write permissions.

`.bss` This section also contains writable data. It is very similar to the `.data` section except it stores uninitialized (or initialized to 0) static variables. At runtime, it is virtually indistinguishable from the `.data` section.

2.3 Assembly: x86 Style

As we are examining the emitted assembly code, we provide a brief introduction to the `x86` ISA. We do so by analyzing one of the most complicated instructions used below:

```
rep movs QWORD PTR es:[rdi],
         QWORD PTR ds:[rsi]
```

While this single instruction may seem intimidating (and you may wonder how it's even implemented in Silicon!), we break it down and find it more manageable.

`rep` prefix This prefix indicates that we will repeat the following command the number of times stored in the `ecx` register. Properly, we repeat until `ecx` equals zero, decrementing `ecx` after each repetition. In this case, we execute the `movs` operation `ecx` times.

`movs` instruction This instruction will move the value from the second operand into the first operand. It is important to note that a `mov` instruction does *not* clear the data from the second operand. It may be more appropriately thought of as a copy operation.

`movs` operand We analyze the first operand only for brevity. Firstly, we are operating on the value at the address stored in the `rdi` register (destination index). (The `es` prefix indicates that we are using the segment specified by the `es` register. This can be ignored for our purposes.)

The `QWORD PTR` indicates that we are treating the `rdi` value as the address of a quadword. This quadword specification is important, because it tells us how big a space to copy (4 words or 64 bits) in a single iteration. Had this been `DWORD` (2 words), `WORD` (1 word), or `BYTE` (1 byte), we would need to adjust the `ecx` register in order to copy the same amount of data.

2.3.1 Summary

The instruction mentioned above will copy `ecx` number of `QWORD`s from the section of data starting with the address in `rdi` to the section of data starting with address in `rsi`. We are copying a total of `ecx`$*4$ bytes.

2.4 Pointers vs. Arrays

Pointers are often implicitly converted to arrays, when being passed as function arguments, for example. However, the C programmer must not lose sight of their important differences.

The C standard describes the frequent conversion of arrays to pointers:

> Except when it is the operand of the sizeof operator or the unary & operator, or is a character string literal used to initialize an array of character type [...] an lvalue that has type "array of type " is converted to an expression that has type "pointer to type " that points to the initial member of the array object and is not an lvalue.

This implicit conversion happens often, but the underlying structure is fundamentally different. We must remember that when an array is filled with a string literal, that array is editable; however, when a pointer points to a string literal, that string literal is read-only. This difference (and others) motivates the different ways in which the compiler handles string literals.

3 Outline

To examine the behavior most clearly, we design the test program such that we are simply assigning a string literal and then printing it using `printf`. The `printf` will better ensure that the variable we assign the string literal to is not optimized away.[1]

[1]By invoking `printf` on our variable, we ensure the optimizing compiler will not deem it unnecssary and remove it.

```c
#include <stdio.h>

int main() {
  char c[5] = "word";
  printf("%s\n", c);
  return 0;
}
```

Figure 1: Simple program with single array

```c
#include <stdio.h>

int main() {
  char c1[211] =
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords"
    "wordsandwordsandwords";
  char *c2 = "wordsandwordsandwords";
  printf("%s %s\n", c1, c2);
  return 0;
}
```

Figure 2: Moderately complicated program with array and pointer

Experimentation is done using three different size strings (including null character): 5, 22, 211. These arbitrarily chosen values allow us to experiment with short, medium, and long strings. All strings of each length will be identical.

The simplest programs have only one string literal, assigned to either a `char` array or a `char` pointer. We see an example in Figure 1.

More complicated programs have each one `char` array and one `char` pointer. Enough of these types of programs were tested to allow for each combination of sizes. We see an example in Figure 2.

Finally, more versions are tested such that we have two identical arrays of the same size. A variation is done on this such that one array will be labeled `const` and the other not. We see an example in Figure 3.

```c
#include <stdio.h>

int main() {
  const char c1[22] =
    "wordsandwordsandwords";
  char c2[22] = "wordsandwordsandwords";
  char *c3 = "wordsandwordsandwords";

  printf("%s %s %s\n", c1, c2, c3);
  return 0;
}
```

Figure 3: Complicated program with multiple arrays and a pointer

We are thus testing a total of 21 programs.

Each program will be compiled with various optimization flags. The sets of flags used include:

`O0` Default optimization (none).

`O1` Basic optimizations

`O2` More optimizations. Nearly all optimizations without a space-speed tradeoff

`O3` Even more optimizations.

`Os` Optimize for size

`O3 fmerge-all-constants` Includes `O3` optimizations plus an optimization to merge identical constants, including constant initialized arrays.

4 Analysis

I first lay out an analysis for the default case: no optimization by highlighting the result of each test. From that, I move on to the differences that each subsequent level of optimization causes.

4.1 Default Optimization `O0`

The analysis of the single string literal was by far the most straightforward.

Using a `char` pointer produced a simple `mov` instruction:

```
mov    QWORD PTR [rbp-0x8], 0x4005d4
```

where `0x4005d4` is an address in the `.rodata` section that is the beginning of a string of bytes representing that string literal. We note this behavior was consistent regardless of the size of the string. The only difference was the size allocated in the `.rodata` section for the string.

4.1.1 `char` arrays

Using a short `char` array, however, produced `mov` instructions that filled space on the stack with an immediate:

```
mov    DWORD PTR [rbp-0x10], 0x64726f77
mov    BYTE  PTR [rbp-0xc],  0x0
```

The first instruction here loads in "word" and the second the null terminating byte.

Similarly, a medium-sized `char` array produced the following instructions:

```
movabs rax, 0x646e617364726f77
mov    QWORD PTR [rbp-0x20], rax
movabs rax, 0x646e617364726f77
mov    QWORD PTR [rbp-0x18], rax
mov    DWORD PTR [rbp-0x10] ,0x64726f77
mov    WORD  PTR [rbp-0xc], 0x73
```

These instructions load the medium-sized string "wordsandwordsandwords" onto the stack. We importantly note that this does include loading the null byte because with the final `mov` instruction we are loading a `WORD` size not a `BYTE`, meaning we are really loading `0x0073` onto the stack to complete the array.

Finally, we examine a long string. This case is much more interesting as we are in fact copying the string from the `.rodata` section onto the stack.

```
lea    rdx,[rbp-0xf0]
mov    eax,0x4006b8
mov    ecx,0x1a
mov    rdi,rdx
mov    rsi,rax
rep movs QWORD PTR es:[rdi],
         QWORD PTR ds:[rsi]
mov    rax,rsi
mov    rdx,rdi
movzx  ecx, WORD PTR [rax]
mov    WORD PTR [rdx],cx
add    rdx,0x2
add    rax,0x2
movzx  ecx, BYTE PTR [rax]
mov    BYTE PTR [rdx],cl
add    rdx,0x1
add    rax,0x1
```

Let's demystify the above snippet. We first load the address of the stack array into `rdx`. Next, we load the address of the 211 character string in the `.rodata` section. After this, we move the number of `QWORD`s (8 bytes) that we'll copy using the `rep` command. We then move from the `.rodata` section to the stack.

The commands after this are simply to fill in the bytes we did not copy because we were copying with `QWORD` length. The syntax is a bit unusual to fit the syntax of the `rep` paradigm.

4.1.2 Combined Results

When we compile a program with multiple string literals we interestingly see independent behavior. That is the arrays will behave as above and the pointers will behave as above, with each having no effect on the others.

We do see, though, that when the array uses an address in the `.rodata` section, it is the same address that the pointer uses, here `0x4006d8`:

```
lea    rdx,[rbp-0xf0]
mov    eax,0x4006d8
mov    ecx,0x1a
mov    rdi,rdx
mov    rsi,rax
rep movs QWORD PTR es:[rdi],
         QWORD PTR ds:[rsi]
mov    rax,rsi
mov    rdx,rdi
movzx  ecx,WORD PTR [rax]
mov    WORD PTR [rdx],cx
add    rdx,0x2
add    rax,0x2
movzx  ecx,BYTE PTR [rax]
mov    BYTE PTR [rdx],cl
add    rdx,0x1
add    rax,0x1
mov    QWORD PTR [rbp-0xf8],0x4006d8
```

We finally note that labeling const seems to have no effect on how the string literal is loaded.

4.2 Some Optimization O1

Applying a small level of optimization has very little effect in terms of how the arrays are loaded. The only noticeable effect was the consolidation of `movs` when using a medium-length string and the elimination of a stack variable for the pointer (which is only used in the printf statement):

```
movabs rax,0x646e617364726f77
mov    QWORD PTR [rsp],rax
mov    QWORD PTR [rsp+0x8], rax
mov    DWORD PTR [rsp+0x10],0x64726f77
mov    WORD  PTR [rsp+0x14],0x73
mov    QWORD PTR [rsp+0x20],rax
mov    QWORD PTR [rsp+0x28],rax
mov    DWORD PTR [rsp+0x30],0x64726f77
mov    WORD  PTR [rsp+0x34],0x73
mov    r8d,0x4006d4
```

All but the last line show how the two medium-sized arrays are loaded. Notice only one `movabs` command, putting the long constant into a register, compared to that same constant being reloaded into the register before each use when compiled with no optimization. The final line shows the address of the string constant being loaded directly into the `r8` register.

We notice, however, that the constant value corresponding to the short string is *not* loaded into a register. It is directly `mov`d onto a stack address each time.

4.3 More Optimization O2

We see very little effect on the loading of arrays here. The single noticeable difference was the reordering of the loads into the array. At lower levels of optimization, the arrays were loaded in order: this is no longer the case. Additionally, when two arrays were present, their loads were interspersed with each other.

4.4 Most Optimization O3

We see no difference here in the structure of loads of arrays and pointers. There are differences in code, but none are related to the storage of string literals.

4.5 Merging Constants

The `fmerge-all-constants` flag seems to have no effect when used in conjunction with `O3` and `Os` (which will be discussed below).

4.6 Optimizing for Size Os

We see a clear effect when compiling with `Os`. First, we see that the compiler lowers the threshold for directly loading vs. copying into an array. Medium-sized strings are now copied from `.rodata` in the same way that long strings are.

This will reduce the size of the binary because there is a size-overheard involved with the `mov` command. That is, we must specify that it is a `mov` command and also where to move—in each instruction. With the `rep movs` approach, we have a single instruction that will repeat multiple times, thus saving space.

However, short strings are *not* loaded in this way. They still use `mov` instructions.

A final, interesting note: when using this optimization level, the compiler can detect overlapping strings.

Let us consider the case where we have a long string being stored into an array as well as a medium-sized string being stored into a pointer. Here, we will copy data into the array from the `.rodata` section, where this is a string of length 211. When assigning the pointer, however, we simply point to the appropriate byte 21 from the end of the large sequence. That is, the strings will share a null byte in `.rodata`! It is important to note that this is only possible because the strings end the same. Additionally important to note: they do not share an null byte once the sequence of bytes in copied into the array. More precisely, the source of the array copy shares a null byte with the pointer.

We only see this behavior with a combination of medium and large strings. Short strings are still directly loaded. However, a combination of medium and short strings do this regardless of which is assigned to an array and which to a pointer.

5 Findings

As expected, the compiled binaries reflect that paradigm that assigning a string literal to an array is shorthand notation for braced initializer. The strings are always copied into the stack location of that array variable. On the other hand, pointers are simply loaded with the address of a string literal in the `.rodata` section. This reflects the usual paradigm of raising a segmentation fault upon attempting to write to a string literal.

Even at no optimization, if the string is long enough that the compiler decides to store it in `.rodata` and have it copied in, it will condense the reference to that string literal such that there is only one copy in the `.rodata` section.

Optimization level has surprisingly little effect (outside of `Os`). We see very few changes as we move through `O0` to `O3`.

However, using `Os` will increase the compiler's preference for using the copy from `.rodata` over using multiple `mov` instructions. Of note, the compiler will recognize when one string ends with another, arranging the `.rodata` section such that string literals may share null bytes.

Even though arrays and pointers are sometimes interchangeably used, this investigation makes clear their underlying, important differences. The compiler handles them drastically differently, and the careful C programmer will do the same.