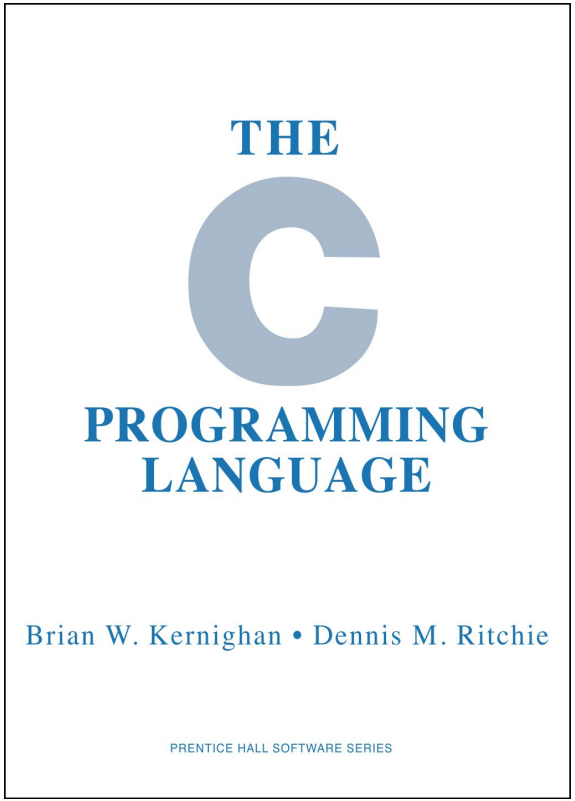

MONTANA
STATE UNIVERSITY

Strings & Structs

Complex Data Types In C

Data Types in C

- We have been working with data types in C for a few lessons now
- Mostly the simple data types, and mainly integers
- Time to look at some more complex data types!



C Strings

- In C, strings are a one-dimensional array of characters, terminated by a null (zero) character
- Each character is 8 bits or 1 byte

```
char *str = "Hello";
```

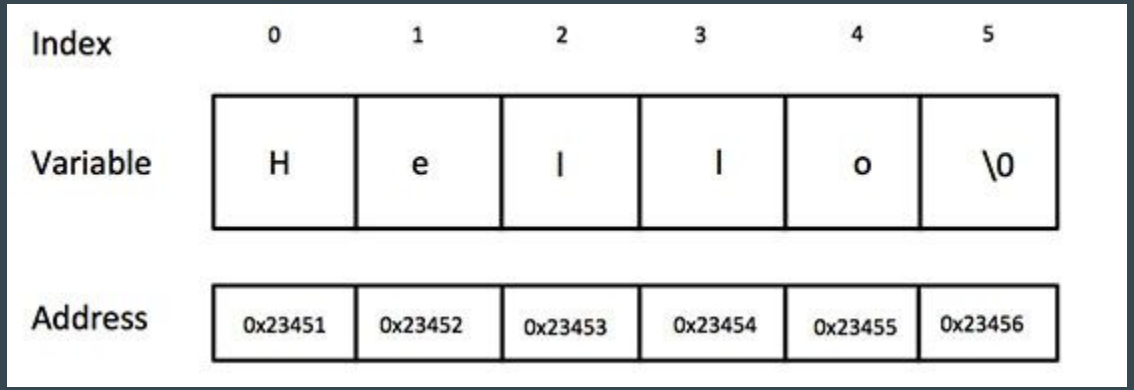
C Strings

- There are actually *six* characters in this string
  - A hidden 0 terminates the string!

```
char *str = "Hello";
```

C Strings

- There are actually *six* characters in this string
  - A hidden 0 terminates the string!



C Strings

- We have been working with strings mainly as *String Literals*
- It is important to understand that a string literal evaluates to a *char **

```c
#include <stdio.h>

int main() {
    char * str = "Hello, Strings!";
    printf(str);
}
```

C Strings

- From our pointer lecture we know this means we can do pointer arithmetic with strings

```
void string_ptr_fun_1() {
    char * str = "Hello, Strings!";
    for (int i = 0; i < 15; ++i) {
        printf( format: "Char at %d is %c\n", i, *(str + i)
    }
}
```

C Strings

- Here we put the string length in the for loop
- How does printf know where to stop?

```
void string_ptr_fun_1() {
    char * str = "Hello, Strings!";
    for (int i = 0; i < 15; ++i) {
        printf( format: "Char at %d is %c\n", i, *(str + i)
    }
}
```

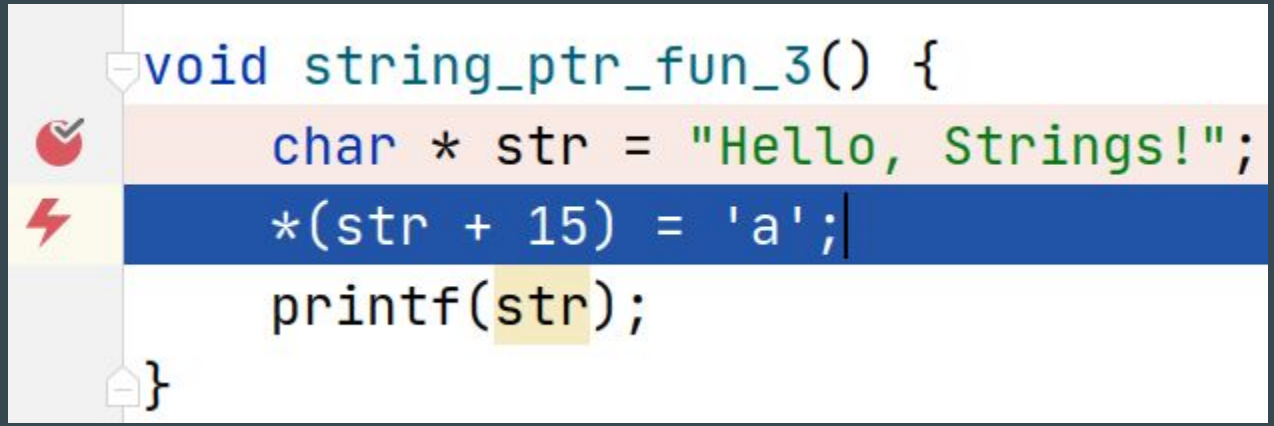
C Strings

- Note the last number printed in this loop is a zero
- String in C are *null terminated*
- Recall that null, in C, means 0 (which also means false)

```
void string_ptr_fun_2() {
    char * str = "Hello, Strings!";
    for (int i = 0; i < 16; ++i) {
        printf( format: "Char at %d is %d\n", i, *(str + i));
    }
}
```

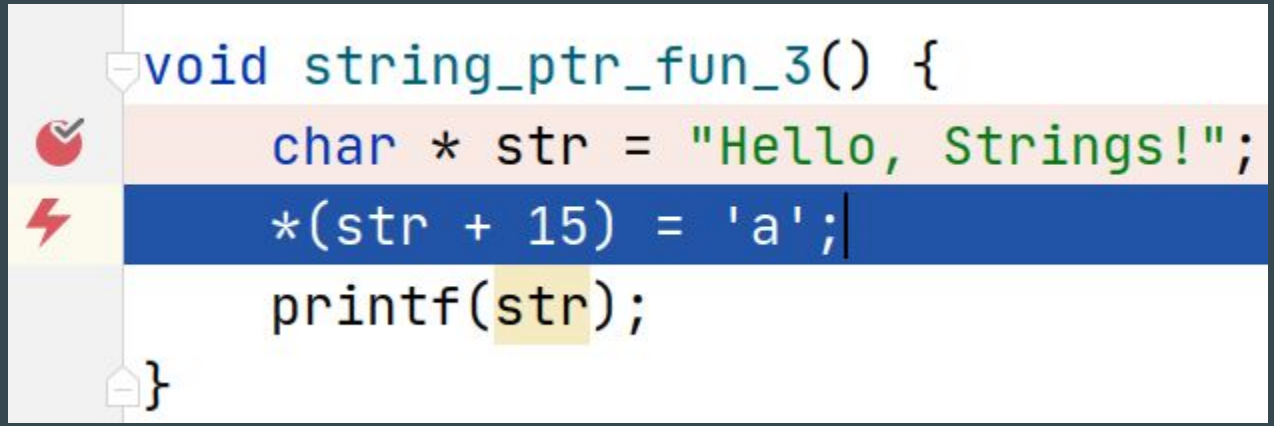
C Strings

- Fun with segfaults!
- Let’s try to set that terminator to a non-null value
- With string literals, you get a segfault



C Strings

- This is because the region of memory where string literals are stored is read only



C Strings - Buffer Overflows

- Buffer overflows are one of the most common attacks on the internet
- Because C does not check array or pointer writing, it is possible to write data off the end of a string or array
- Things have gotten better as it has gotten harder to write into executable locations but this is still an issue
- Buffer overflow Demo...

C Strings - Standard String Functions

- Many string-related functions can be found in <string.h>

strcpy(s1, s2)	Copies string s2 into string s1
strcat(s1, s2);	Concatenates string s2 onto the end of string s1
strlen(s1);	Returns the length of string s1
strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2
strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1
strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1

C Strings - Standard String Functions

- Safe versions of functions tend to start with *strn*

strncpy(s1, s2, n)	Copies string s2 into string s1
strncat(s1, s2, n);	Concatenates string s2 onto the end of string s1
strnlen(s1, n);	Returns the length of string s1
strncmp(s1, s2, n);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2

C Strings

- OK, enough about strings
- String manipulation is one of the more difficult aspects of C
- It is tedious and error prone, but that’s just the way it is...

C Structs

- As we have discussed, C is a functional language
- The main component of a C program is *functions*
- C does have a concept of structures, however
- Structures can be thought of as an object, but with no methods

```
struct Book {
    char  title[10];
    char  author[50];
    char  subject[100];
    int   book_id;
};
```

C Structs

- You can declare struct variables with the struct keyword

```
struct Book bk;
strcpy(bk.title, src: "Fun with C");
printf( format: "Title: %s\n", bk.title);
```

C Structs

- You can also use the *typedef* to make it easier to declare and work with a struct

```
typedef struct Book {
    char  title[10];
    char  author[50];
    char  subject[100];
    int   book_id;
} Book;
```

C Structs

- You can also use the *typedef* to make it easier to declare and work with a struct

```
Book bk;
strcpy(bk.title, src: "Fun with C");
printf( format: "Title: %s\n", bk.title);
```

C Structs

- It is common to pass around pointers to structs
- To access fields you need to do a derefrence the struct pointer and then access the field

```
void bk_print_title(Book *book){
    printf( format: "Title: %s\n", (*book).title);
}
```

C Structs

- This is a sufficiently common pattern that C provides the arrow operator (->) to do both the dereference and the field access:

```
void bk_print_title(Book *book){
    printf( format: "Title: %s\n", book->title);
}
```

C Structs

- This is the inspiration of the C++ syntax, if you are familiar with that language

```
void bk_print_title(Book *book){
    printf( format: "Title: %s\n", book->title);
}
```

Review

- C strings are a series of characters, terminated by a null (0) character
- C strings are touchy and error prone
- The vast majority of security issues in the wild are due to buffer overruns, often related to C Strings
- Structs provide a way to define simple data structures in C
- Structs contain only data, they are not objects like you are used to in Python or Java


MONTANA
STATE UNIVERSITY