CS107 Lecture 9
Arrays and Pointers

Reading: K&R (5.2-5.5) or Essential C section 6

The Seven Commandments of C Strings

1. If we create a **new string** as a local **`char[]`**, we can modify its characters because the relevant, writable memory resides in the stack frame of the declaring function.
2. We cannot set the name of a **`char[]`** equal to another value, because it is not a **true pointer**, as it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's **automatically converted** to a **`char *`**.
4. If we create a **new string** from a string literal as a **`char *`**, we cannot modify its characters because the characters reside in the **read-only data segment**.
5. We can set a **`char *`** equal to another value, because it is an **assignable pointer**.
6. Adding an offset to a C string delivers another string—often a suffix string—**beginning many positions beyond the leading character**.
7. If we change characters through a string parameter, **those changes persist beyond the function call**, because the **pointer aliases memory that existed before the call**.

The First Commandment

String Behavior #1: If we create a **new string** as a local **`char[]`**, we can modify its characters because the relevant, writable memory resides in the stack frame of the declaring function.

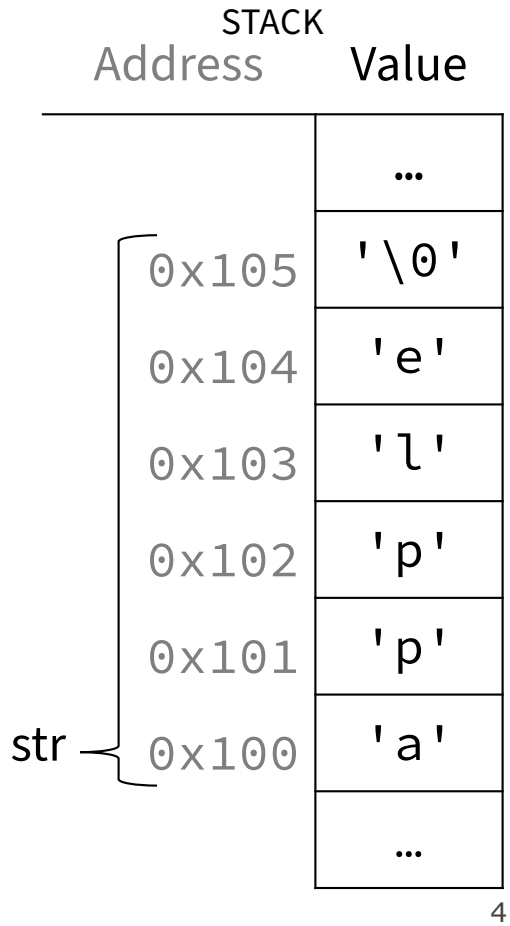
The First Commandment

When we declare an array of characters, memory is reserved in the stack to store the contents of the entire array, and we can modify the characters stored there.

```
char str[6];
strcpy(str, "apple");
```

alternatively, with same effect

```
char str[] = "apple";
```



The Second Commandment

String Behavior #2: We cannot set the name of a **`char[]`** equal to another value, because it is not a **true pointer**, as it refers to the block of memory reserved for the original array.

The Second Commandment

An array variable refers to an entire block of memory. We can't reassign the name of an existing array to refer to another.

```
char good[12];
strcpy(good, "Dr. Jekyll");
char evil[] = "Mr. Hyde";
good = evil; // nope, Jekyll is safe
```

Once declared, an array's size cannot be changed.
We must create another array instead.

The Third Commandment

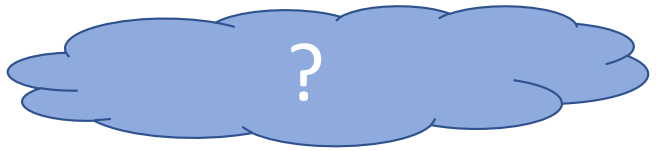
String Behavior #3: If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic using it, it's **automatically converted** to a **`char*`**.

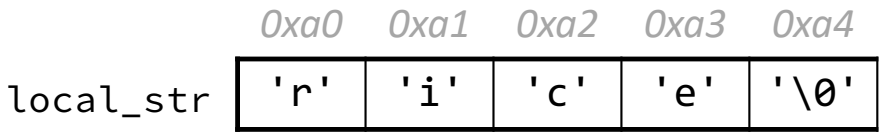
Third Commandment Etudes

How do you think the parameter **`str`** is being represented?

```
void fun_times(char *str) {
    ...
}

int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    fun_times(local_str);
    return 0;
}
```





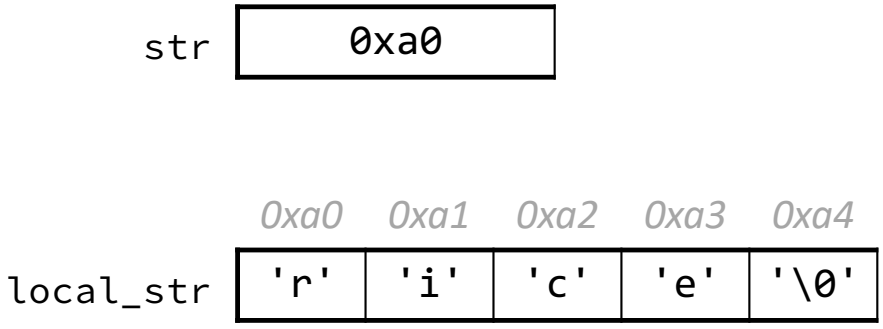
A. A copy of the array **`local_str`**
B. A pointer containing an address to the first element in **`local_str`**

Third Commandment Etudes

How do you think the parameter **`str`** is being represented?

```
void fun_times(char *str) {
    ...
}

int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    fun_times(local_str);
    return 0;
}
```



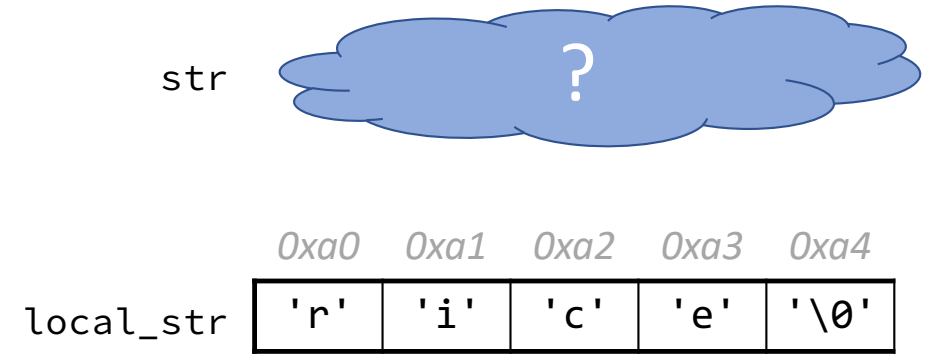
A. A copy of the array **`local_str`**

B. A pointer containing an address to the first element in **`local_str`**

Third Commandment Etudes

How do you think the parameter **`str`** is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```

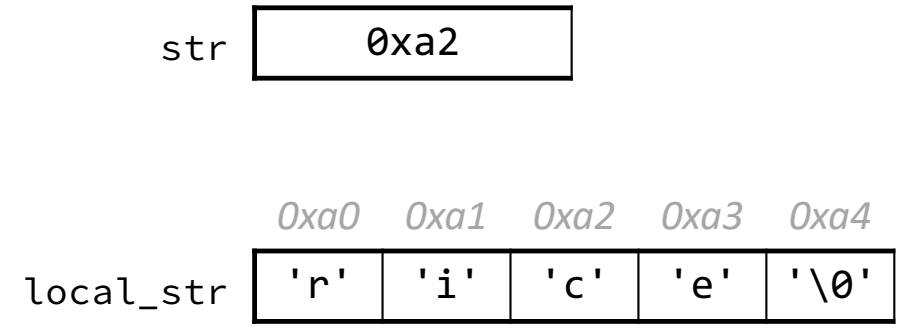


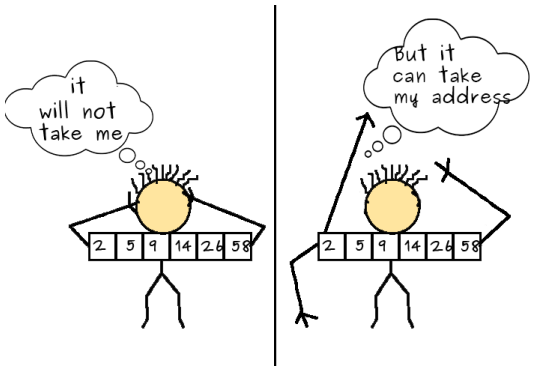
A. A copy of the array **`local_str + 2`**
B. A pointer containing an address to the third element in **`local_str`**

Third Commandment Etudes

How do you think the parameter **`str`** is being represented?

```
int main(int argc, char *argv[]) {
    char local_str[5];
    strcpy(local_str, "rice");
    char *str = local_str + 2;
    ...
    return 0;
}
```





A. A copy of the array **`local_str + 2`**

B. A pointer containing an address to the third element in **`local_str`**

Commandments 1 – 3: Redux

All standard string functions expect pointers to characters (either **`char *`** or **`const char *`**) as parameters. They might accept a **`char[]`** instead, but they are **implicitly converted** to **`char *`** or **`const char *`** before being passed.

```
size_t strlen(const char *s);
int strcmp(const char *s1, const char *s2);
char *strstr(const char *haystack, const char *needle);
char *strncat(char *dest, const char *src, size_t n);
```

```
size_t strlen(const char s[]);
int strcmp(const char s1[], const char s2[]);
char *strstr(const char haystack[], const char needle[]);
char *strncat(char dest[], const char src[], size_t n);
```

A **`char *`** can still be a string **in all the core ways** a **`char[]`** is.

Takeaway: We can **create** strings using **`char[]`** and still **pass them around as parameters** using either **`char[]`** or **`char *`**.

Array and pointers are **not the same**. They do, however, **cooperate**.

The Fourth Commandment

String Behavior #4: If we create a **new string** from a string literal as a **`char *`**, we cannot modify its characters because the characters reside in the **read-only data segment**.

The Fourth Commandment

There is another convenient way to create a string if we do not need to modify it later. We can create a **char *** and set it directly equal to a string literal.

These first four lines run without drama. **salutation**'s characters are mutable, and **greeting** is used in a **read-only manner**.

```
char salutation[] = "Good day!";
char *greeting = "Hello, world!";
salutation[3] = 'f';
printf("%s", greeting);

greeting[0] = 'h';
```

Because **greeting** is declared as a **char *** and initialized to a string literal, those characters are read-only and can't be safely overwritten. This fifth of five lines compiles on the **myth** machines, but when executed results in undefined behavior (e.g., on the **myth**s, this crash crashes).

Fourth Commandment Etudes

For each code snippet below, can we modify characters through **`str`**?

Key Questions: Where do the characters live? In the read-only data segment? Or in writable memory like the stack? What else determines mutability?

```
char str[6];
```

```
char *str = "rutabaga";
```

```
char silly[12];
strcpy(silly, "hootenanny");
char *str = silly;
```

```
char *s = "barnacle";
char *str = s;
```

```
void one(char *str) { ... }
```

```
void two(const char *str) { ... }
```

```
int main(int argc, char *argv[]) {
    char satword[15];
    strcpy(satword, "absquatulation");
    one(satword);
    two(satword);
    return 0;
}
```

Fourth Commandment Gotchas

Question: Is there a runtime check that can be used to determine whether a string's characters are modifiable?

Answer: Not reliably.  This is something you can only tell by looking at the code itself and knowing how the string was created.

> In principle, you might be able to guess from the character addresses, since string literals reside in a wildly different part of memory than stack arrays do. But this is neither reliable nor exhaustive.

Question: So then if I am writing a string function that overwrites characters, how can I tell if the string passed in can be safely modified?

Answer: You can't!  This is something you can clarify as an assumption in documentation.

> If someone calls your function on a read-only string, it will crash on the **`myth`**s.
> That, however, is not your function's fault if it's clear read-only strings are a no-no.

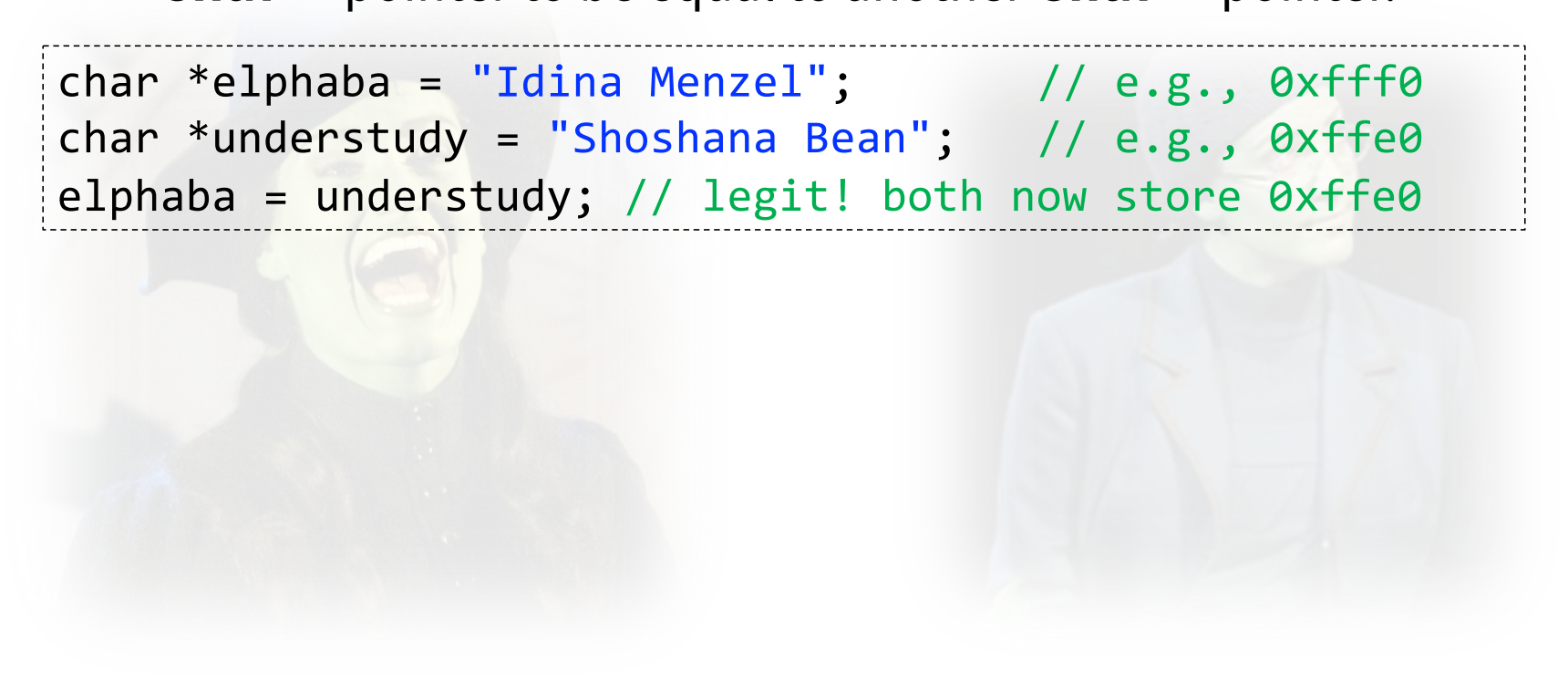
The Fifth Commandment

String Behavior #5: We can set a **`char *`** equal to another value, because it is an **assignable pointer**.

The Fifth Commandment

A **char *** variable refers to a single character. We can reassign an existing **char *** pointer to be equal to another **char *** pointer.

```
char *elphaba = "Idina Menzel";         // e.g., 0xfff0
char *understudy = "Shoshana Bean";     // e.g., 0xffe0
elphaba = understudy; // legit! both now store 0xffe0
```

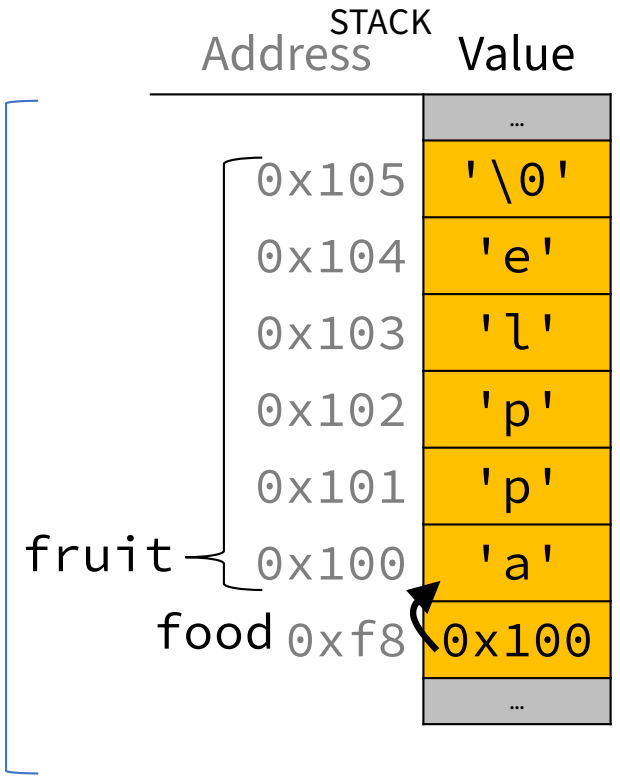


The Fifth Commandment

We can also set a pointer equal to an array name so it addresses the first element of that array.

```
int main(int argc, char *argv[]) {
    char fruit[6];
    strcpy(fruit, "apple");
    char *food = str;
    // equivalent
    char *food = &str[0];
    // equivalent but misleading, avoid
    char *food = &str;
    ...
}
```

main



The Sixth Commandment

String Behavior #6: **Adding an offset** to a C string delivers another string—often a suffix string—**beginning many positions beyond the leading character**.

Sixth Commandment Trivia

When we compute pointer arithmetic, we advance a pointer by a certain number of characters.

```
char *a = "peach";          // e.g., 0xff0
char *b = str + 1;          // e.g., 0xff1
char *c = str + 3;          // e.g., 0xff3

printf("%s", a);            // prints peach
printf("%s", b);            // prints each
printf("%s", c);            // prints ch
```

READ-ONLY DATA

Address	Value
	…
0xff5	'\0'
0xff4	'h'
0xff3	'c'
0xff2	'a'
0xff1	'e'
0xff0	'p'
	…

Sixth Commandment Trivia

When we use bracket notation on anything array-like, we are really performing pointer arithmetic and dereferencing.

```
char *str = "booze";     // e.g., 0xff0
char ch1 = str[4];       // 'e'
char ch2 = *(str + 4);   // 'e'
// both add 4 to 0xff0, then dereference
// dereference 0xff4 to surface the 'e'

// unorthodox alternatives to the above
// that you can use, but should not
char ch3 = *(4 + str);
char ch4 = 4[str];
```

The compiler generally **translates array notation to the equivalent pointer arithmetic**, then generates assembly from the pointer version.
In fact, when looking at the assembly, you **can't truly know whether array or pointer notation was used**.

READ-ONLY DATA

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'z'
0xff2	'o'
0xff1	'o'
0xff0	'b'
	…

The Seventh Commandment

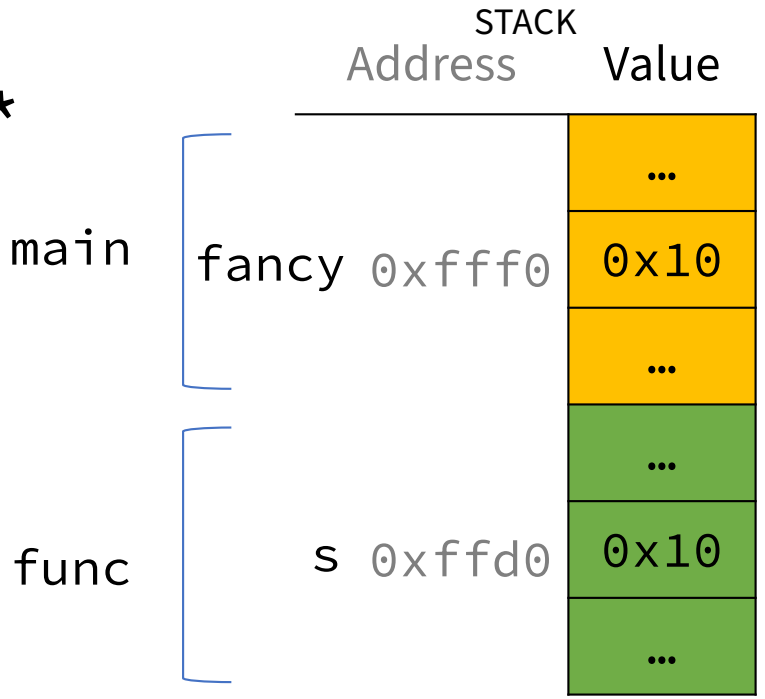
String Behavior #7: If we change characters through a string parameter, **those changes persist beyond the function call**, because the **pointer aliases memory that existed before the call**.

Seventh Commandment Scenarios

When we pass a **char *** string as a parameter, C makes a copy of the address stored in the **char *** and passes it to the function. This means they both refer to the same memory location.

```
void func(char *s) {
   printf("%s\n", s);
}

int main(int argc, char *argv[]) {
   char *fancy = "pulchritude";
   func(fancy);
   ...
}
```

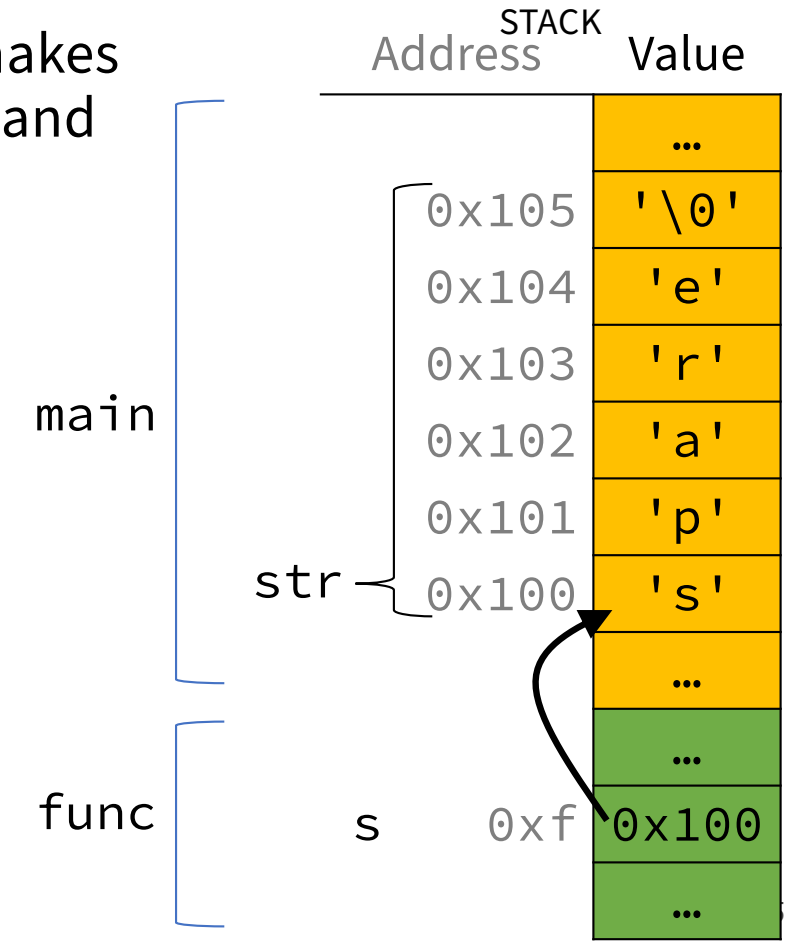


Seventh Commandment Scenarios

When we pass a **char** array as a parameter, C makes a copy of the address of the first array element and passes it (as a **char ***) to the function.

```
void func(char *s) {
    s[4] = 'k';
}

int main(int argc, char *argv[]) {
    char str[] = "spare";
    func(str);
    ...
}
```



The Seven Commandments of C Strings

1. If we create a **new string** as a local **`char[]`**, we can modify its characters because the relevant, writable memory resides in the stack frame of the declaring function.
2. We cannot set the name of a **`char[]`** equal to another value, because it is not a **true pointer**, as it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's **automatically converted** to a **`char *`**.
4. If we create a **new string** from a string literal as a **`char *`**, we cannot modify its characters because the characters reside in the **read-only data segment**.
5. We can set a **`char *`** equal to another value, because it is an **assignable pointer**.
6. Adding an offset to a C string delivers another string—often a suffix string—**beginning many positions beyond the leading character**.
7. If we change characters through a string parameter, **those changes persist beyond the function call**, because the **pointer aliases memory that existed before the call**.