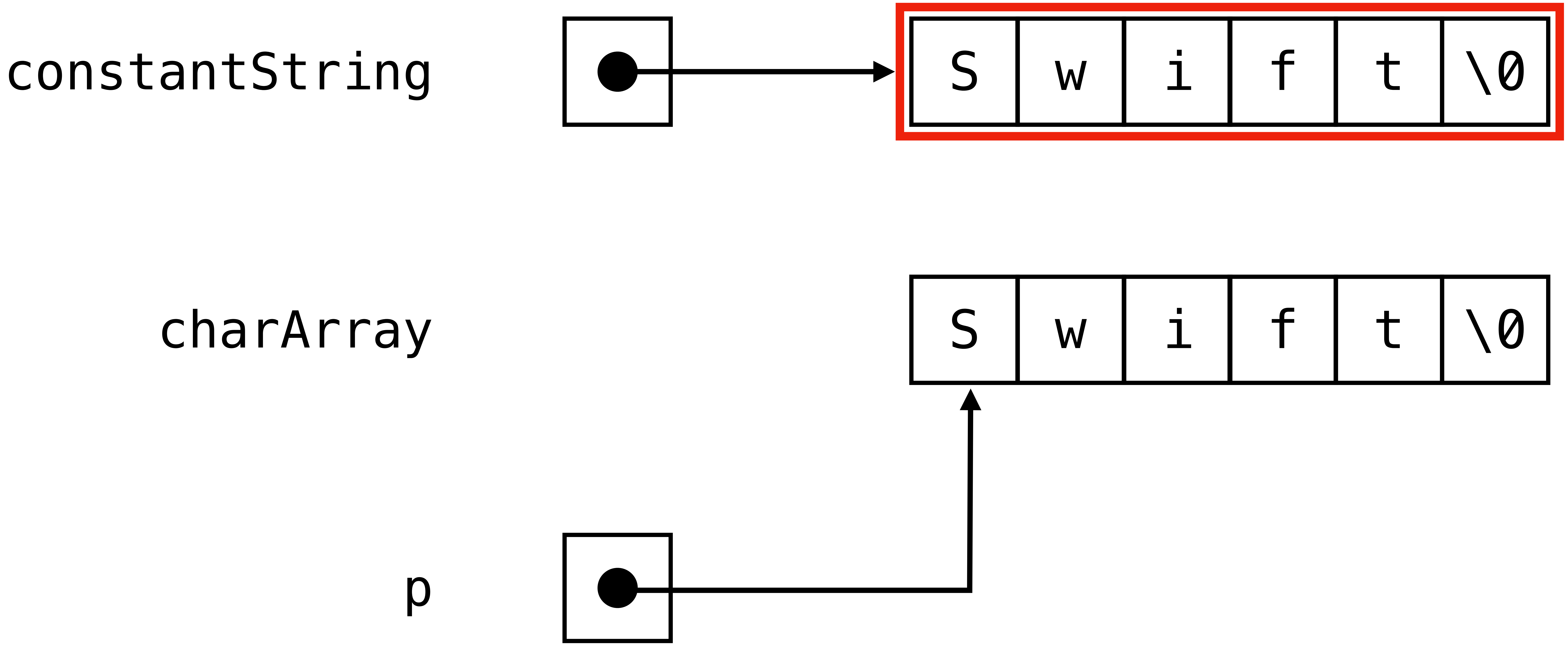
Exercise 20 — Solution

Define a constant string, a char array and a pointer to the latter.

```
// constant string
char * constantString = "Swift";

// char array
char charArray[] = "Swift";

// pointer to charArray
char * p = charArray;
```



constantString

S	w	i	f	t	\0

charArray

S	w	i	f	t	\0

p

1. Does it make any difference when printing these strings?

- entirely
- character by character
- using the pointer syntax
- using the brackets syntax

No difference! Printing is a non-mutating action. 👌

All syntaxes have the same effect for the three variables!

Printing it entirely ...

```
printf("Constant string: %s\n", constantString);
printf("Char array: %s\n", charArray);
printf("Pointer to char array: %s\n\n", p);
```

Characterwise using brackets ...

```
for(int i = 0; i < strlen(constantString); i++)
    printf("%c", constantString[i]);
```

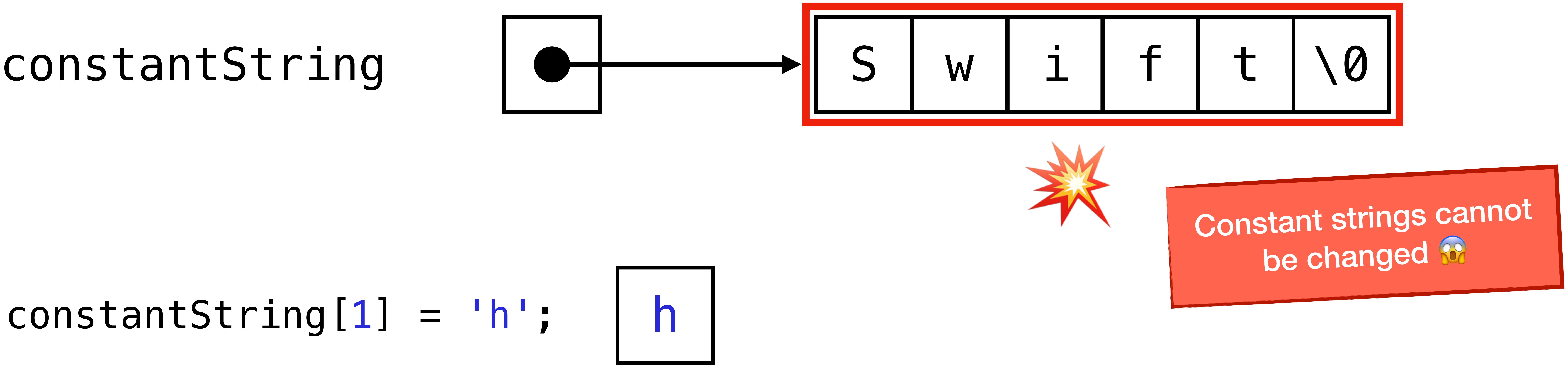
⚠ **Caution:** `sizeof` here would return the size of a *pointer* for `constantString` and `p`!

Characterwise using pointers ...

```
while(*p)
    printf("%c", *p++);
```

Will go on until `\0` is reached

2. Modify the content of these variables. Which calls are valid?



Constant strings cannot be changed 😱

```
constantString[1] = 'h';
```

2. Modify the content of these variables. Which calls are valid?

charArray

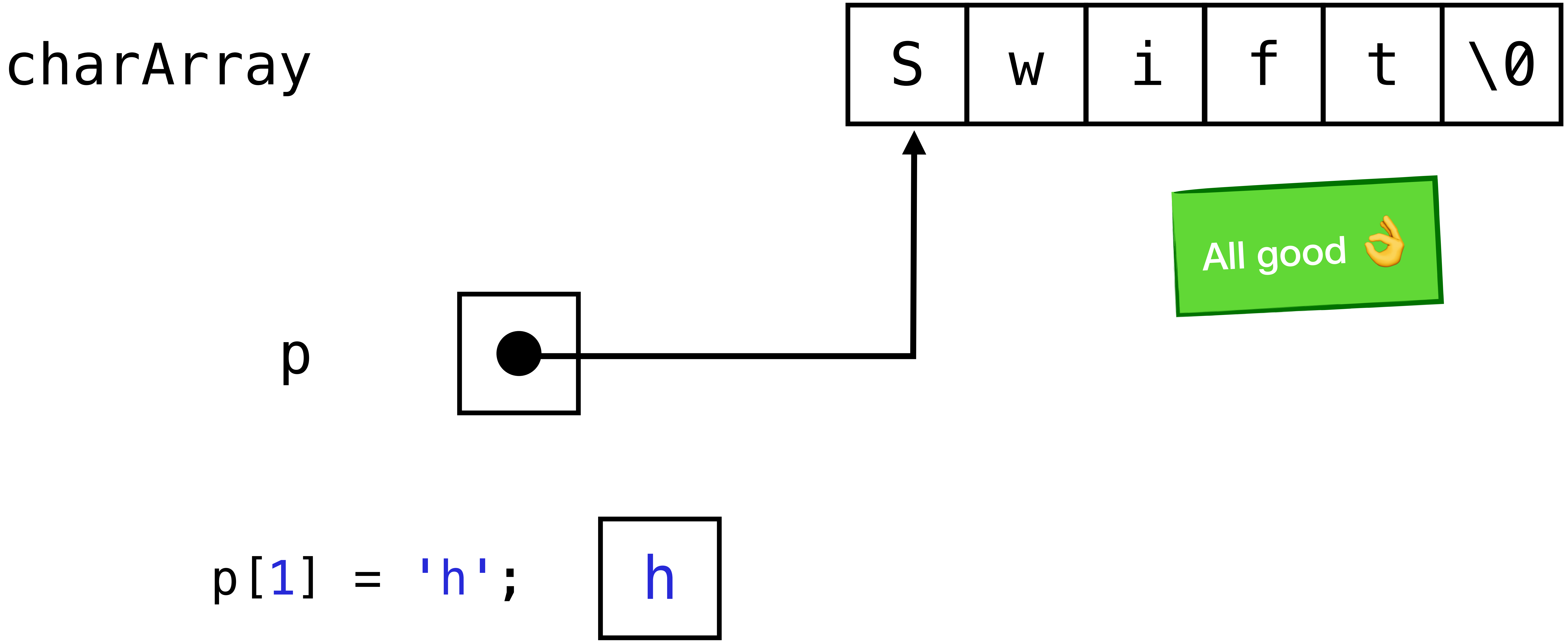
S	w	i	f	t	\0

All good 👌

```
charArray[1] = 'h';
```

h

2. Modify the content of these variables. Which calls are valid?



charArray

S	w	i	f	t	\0

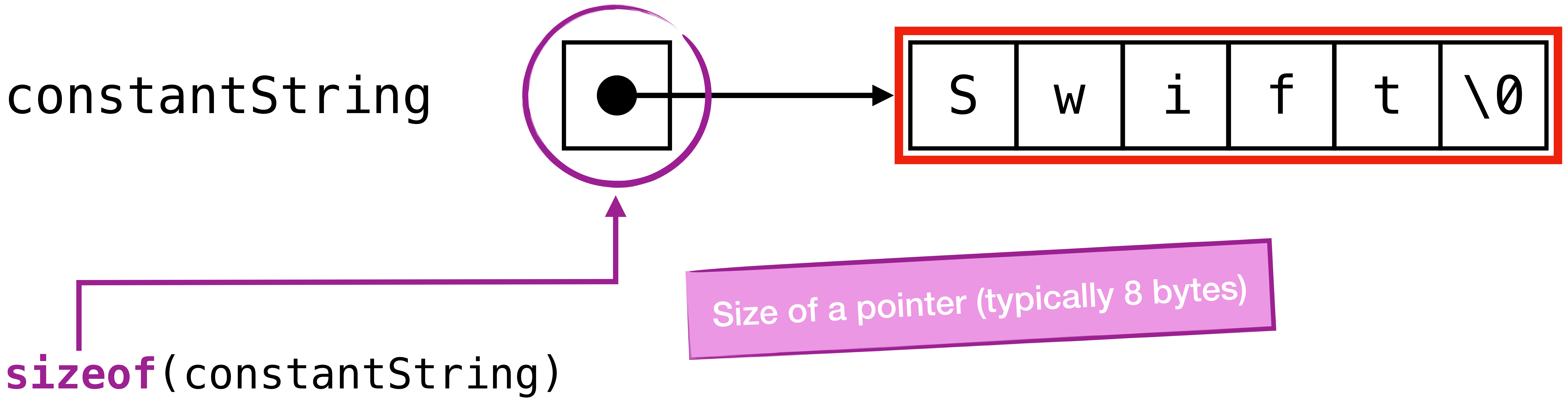
p

```
p[1] = 'h';
```

h

All good 👌

3. What does the `sizeof` operator return on these 3 variables?



`constantString`

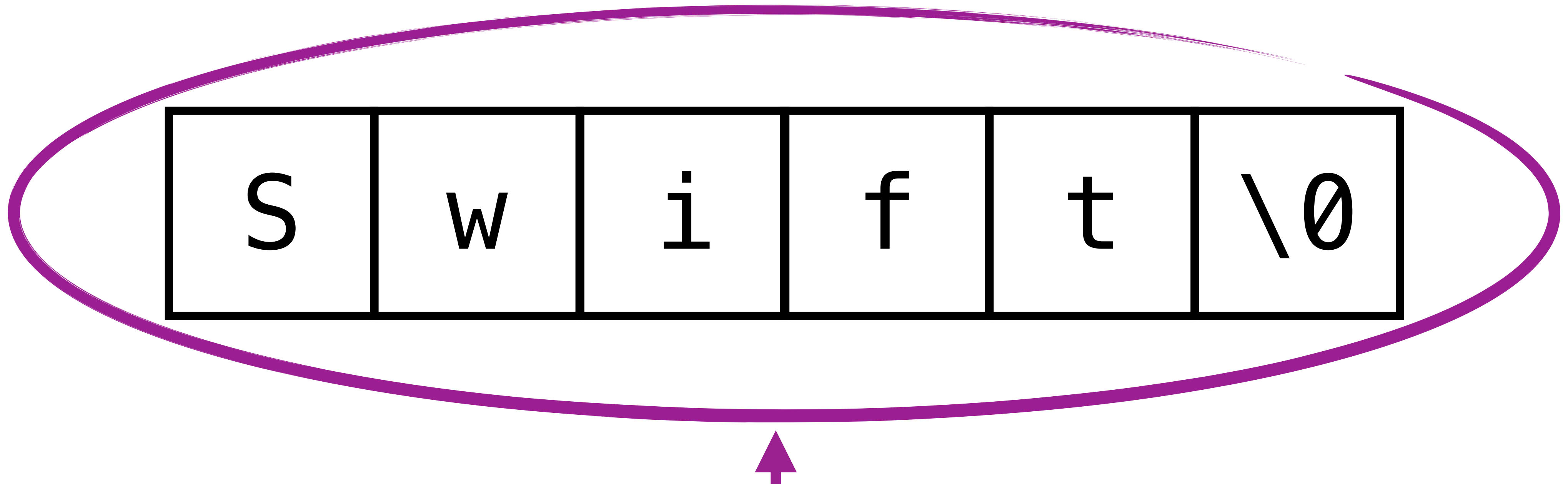
S	w	i	f	t	\0

```
sizeof(constantString)
```

Size of a pointer (typically 8 bytes)

3. What does the `sizeof` operator return on these 3 variables?

`charArray`

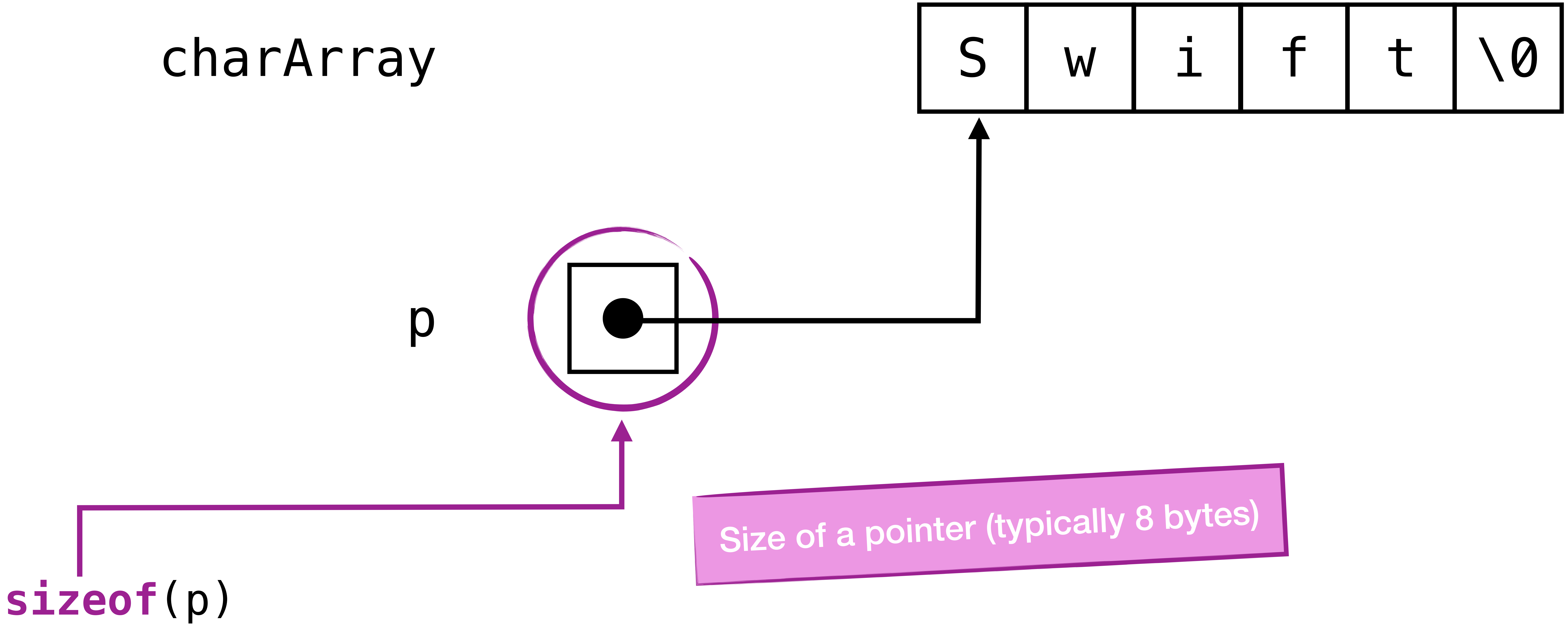


S	w	i	f	t	\0

`sizeof(charArray)`

Size of the array (here: 6 bytes)

3. What does the `sizeof` operator return on these 3 variables?



charArray

S	w	i	f	t	\0

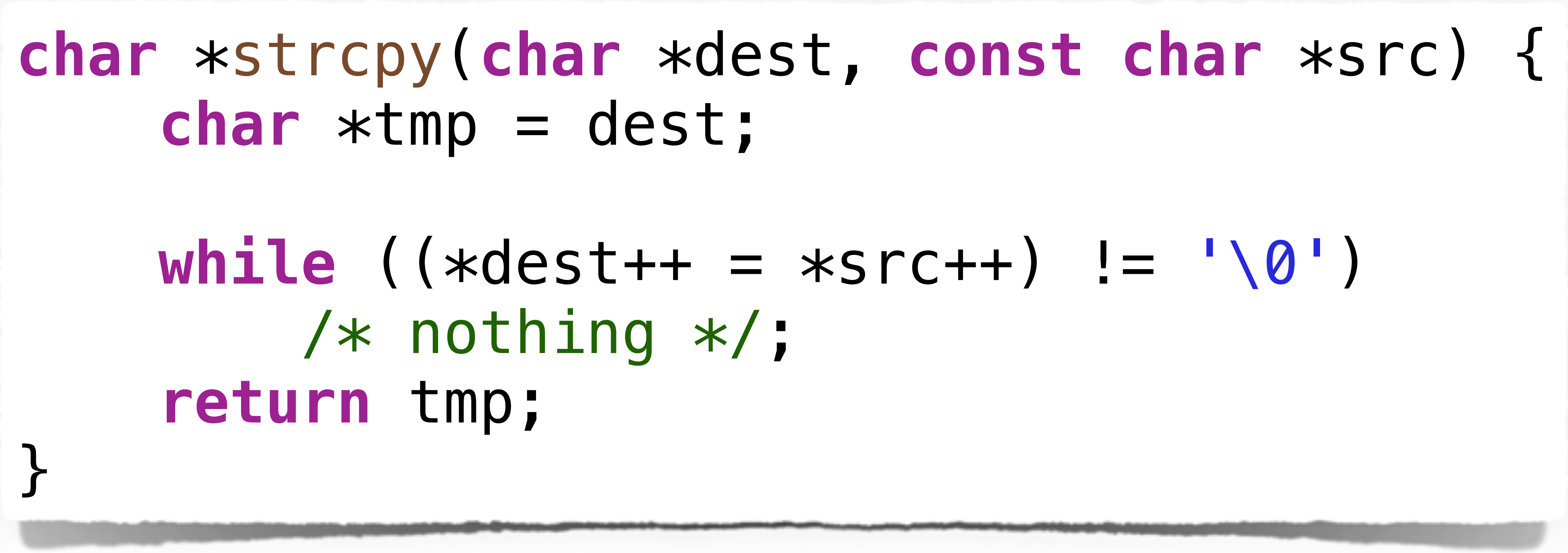
p

`sizeof(p)`

Size of a pointer (typically 8 bytes)

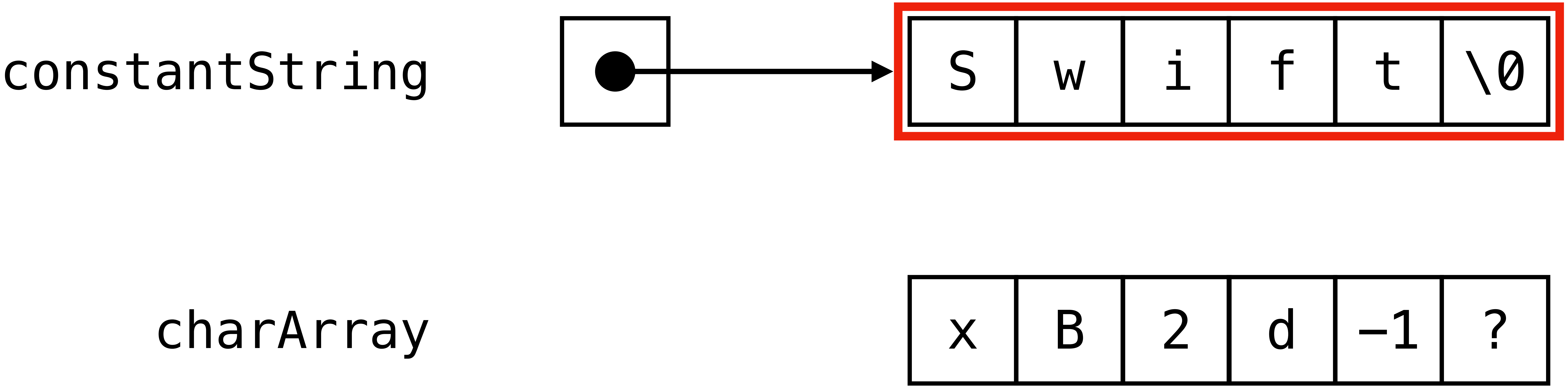
4. Using the strcpy function defined in string.h, what actions are allowed?

a. copy a constant string into a char array



```
char *strcpy(char *dest, const char *src) {
    char *tmp = dest;

    while ((*dest++ = *src++) != '\0')
        /* nothing */;
    return tmp;
}
```

Linus Torvalds' implementation in the Linux kernel, might differ on your platform



constantString

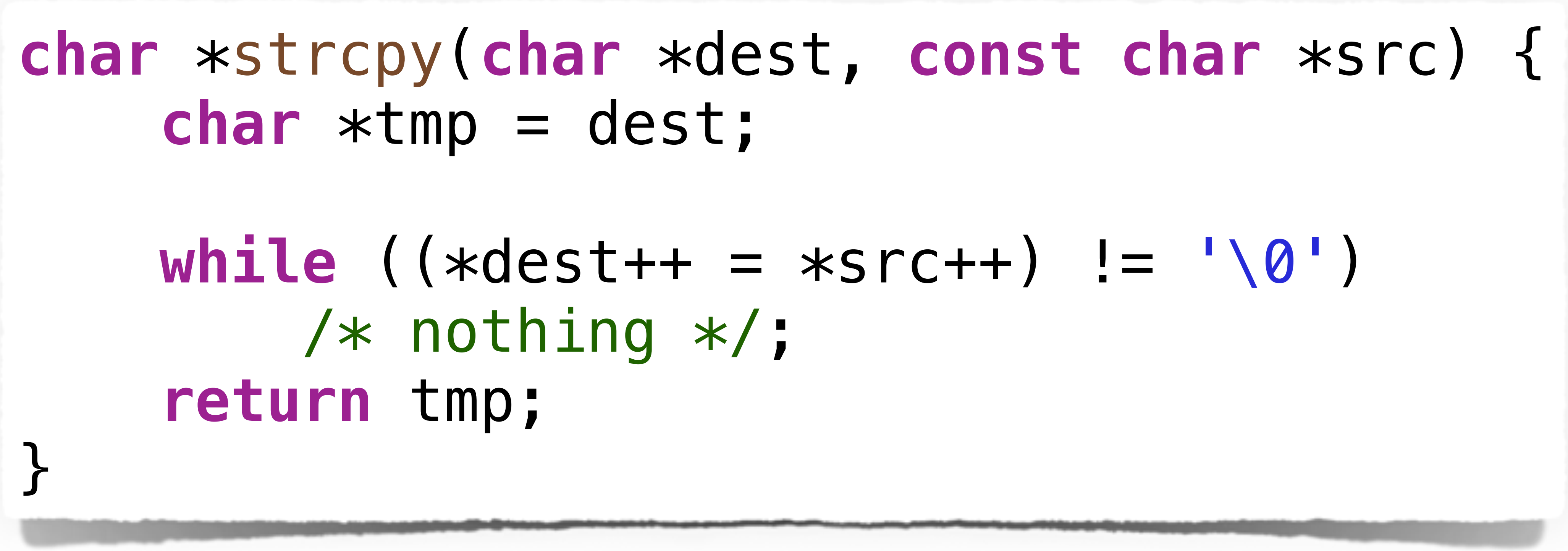
S	w	i	f	t	\0

charArray

x	B	2	d	-1	?

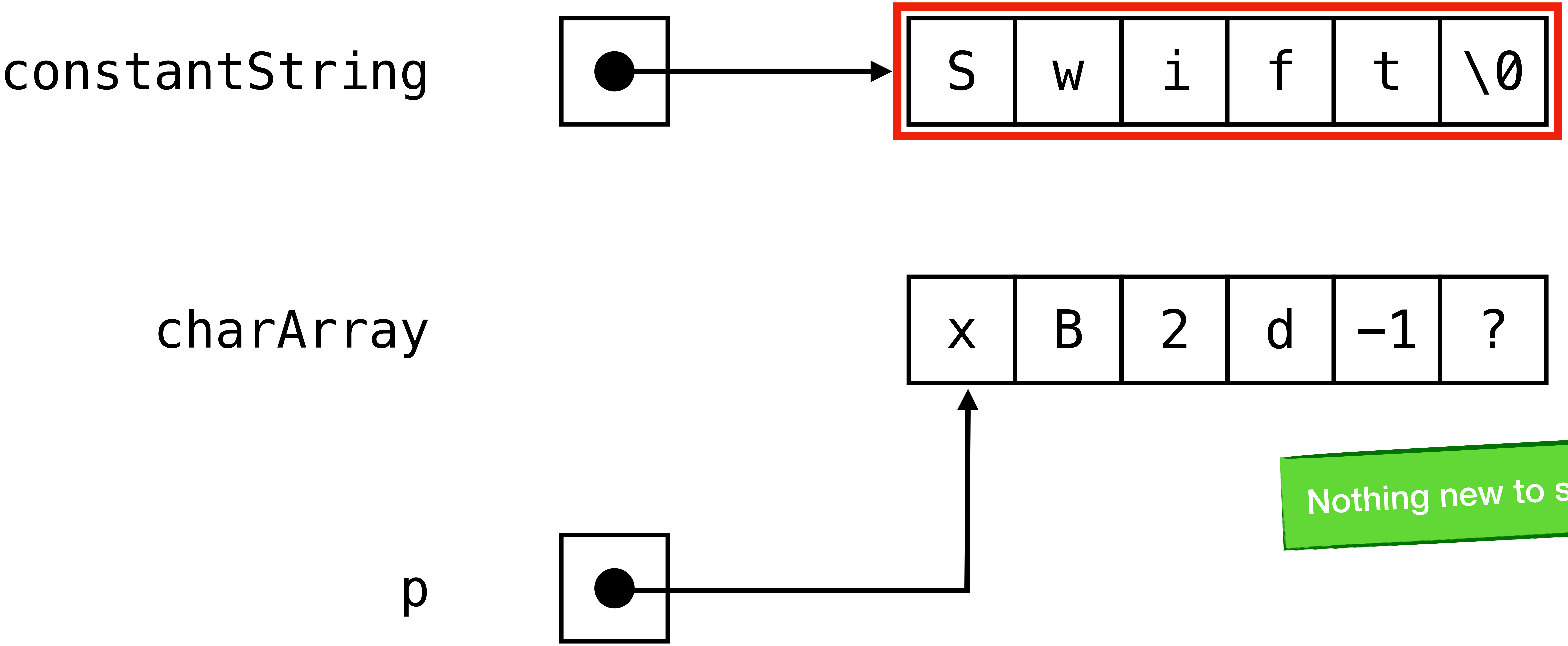
All good 👌

4. Using the `strcpy` function defined in `string.h`, what actions are allowed?



```
char *strcpy(char *dest, const char *src) {
    char *tmp = dest;

    while ((*dest++ = *src++) != '\0')
        /* nothing */;
    return tmp;
}
```

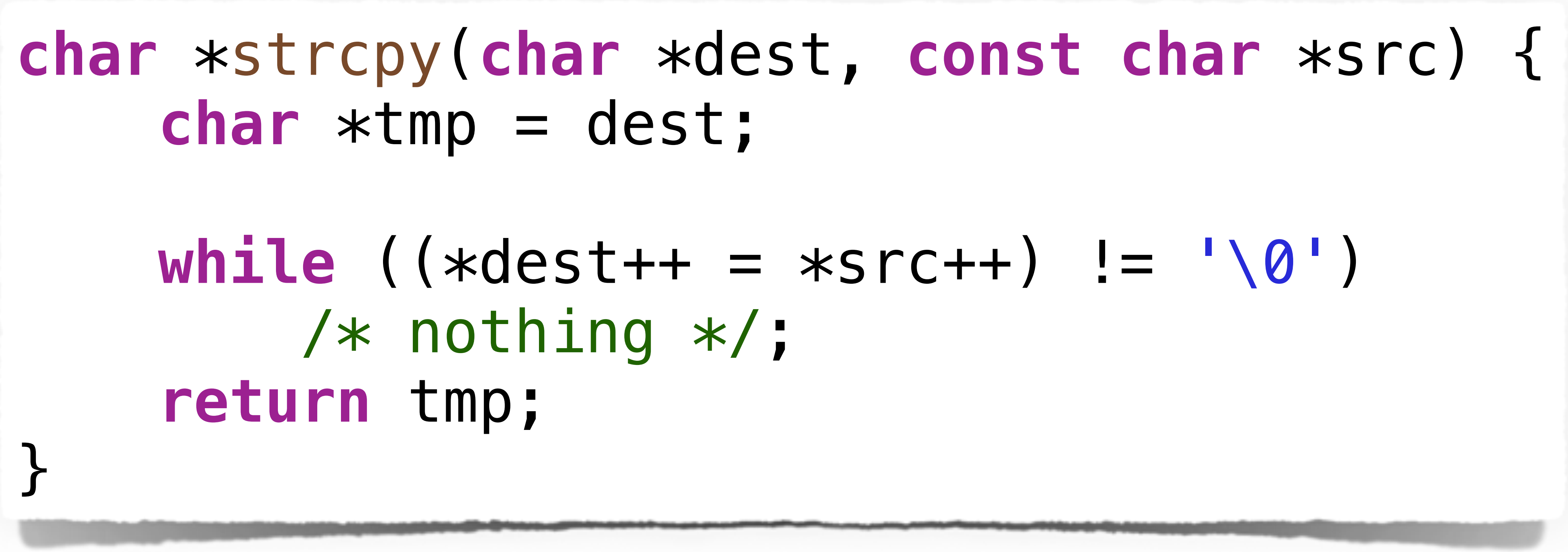
a. Bonus: copy a constant string into a pointer to a char array



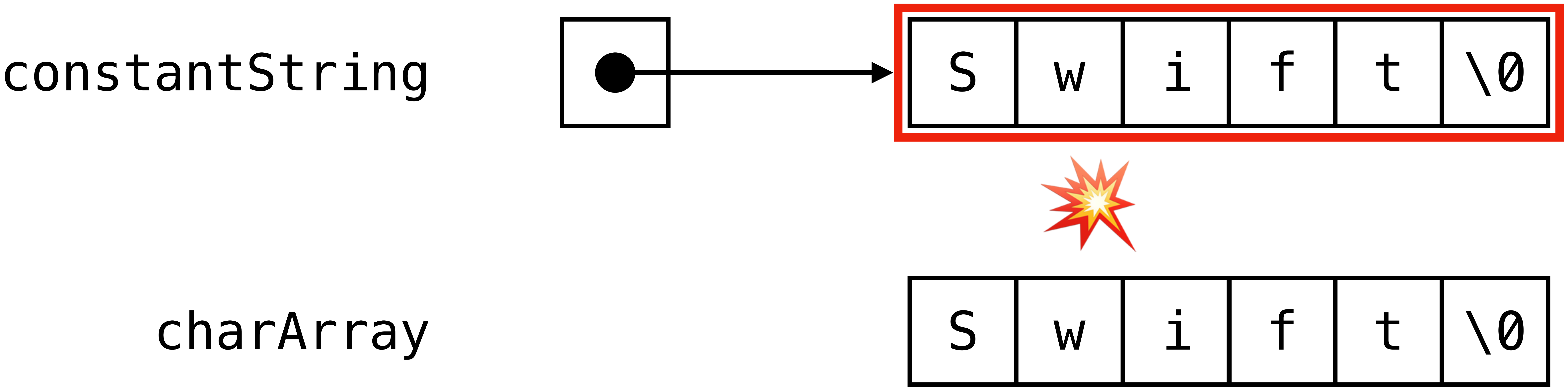
Nothing new to see here! 🤷‍♀️

4. Using the strcpy function defined in string.h, what actions are allowed?

b. copy a char array into a constant string

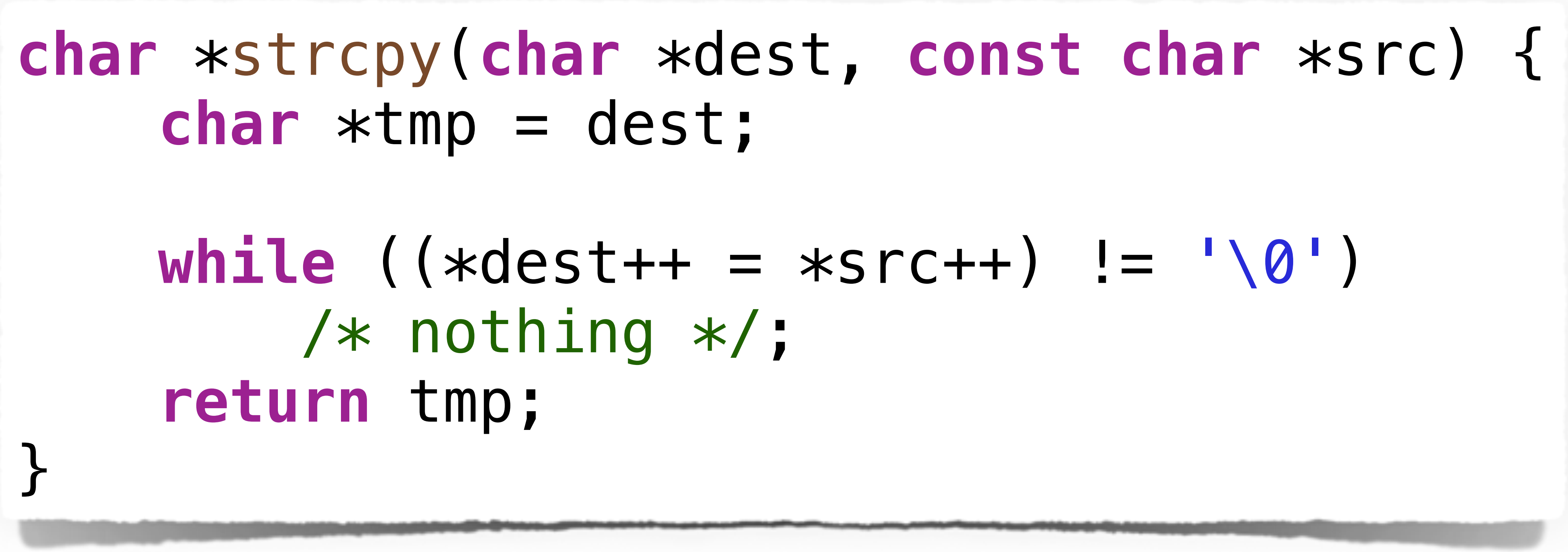


```
char *strcpy(char *dest, const char *src) {
    char *tmp = dest;

    while ((*dest++ = *src++) != '\0')
        /* nothing */;
    return tmp;
}
```



Constant strings cannot be changed 😱

4. Using the `strcpy` function defined in `string.h`, what actions are allowed?



```c
char *strcpy(char *dest, const char *src) {
    char *tmp = dest;

    while ((*dest++ = *src++) != '\0')
        /* nothing */;
    return tmp;
}
```

c. copy a char array into itself

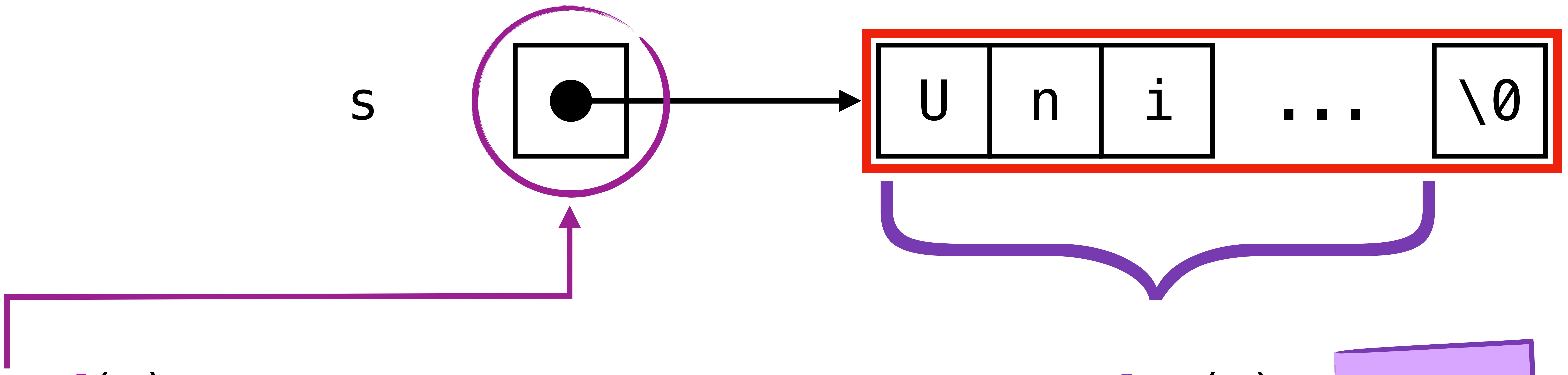
`charArray`	S	w	i	f	t	\0

All good, at least with this implementation!

Depending on how `strcpy` is implemented, a *parameter overlap* can provoke undefined behavior, or lead to a crash. Thus, apart from being pretty useless to copy a string into itself, it's not portable either.

5. Unicode Strings: Define a pointer to a constant string `"Université"`

```c
char * s = "Université";
```



`sizeof(s)`

Size of a pointer (typically 8 bytes)

`strlen(s)`

Here: 11

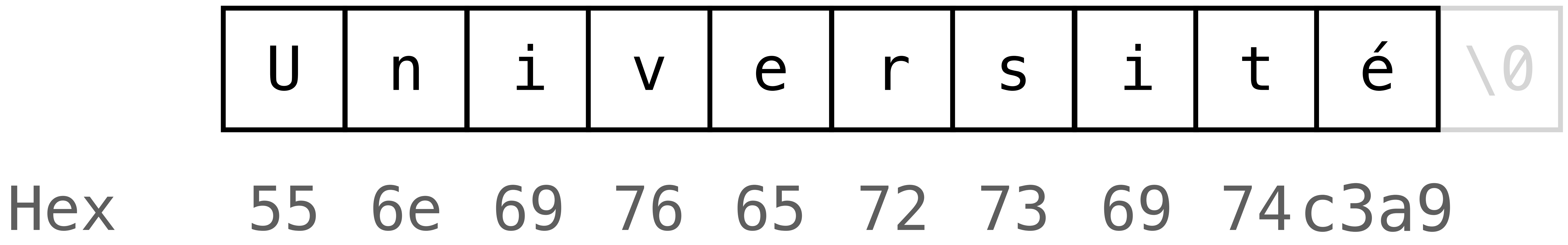
🤔 Wait a minute: 11 !?

U	n	i	v	e	r	s	i	t	é
1	2	3	4	5	6	7	8	9	10

5. Unicode Strings: Define a pointer to a constant string "Université"

```c
char * s = "Université";
```

```c
for(int i = 0; i < strlen(s); i++)
        printf("%02x ", s[i]);
```



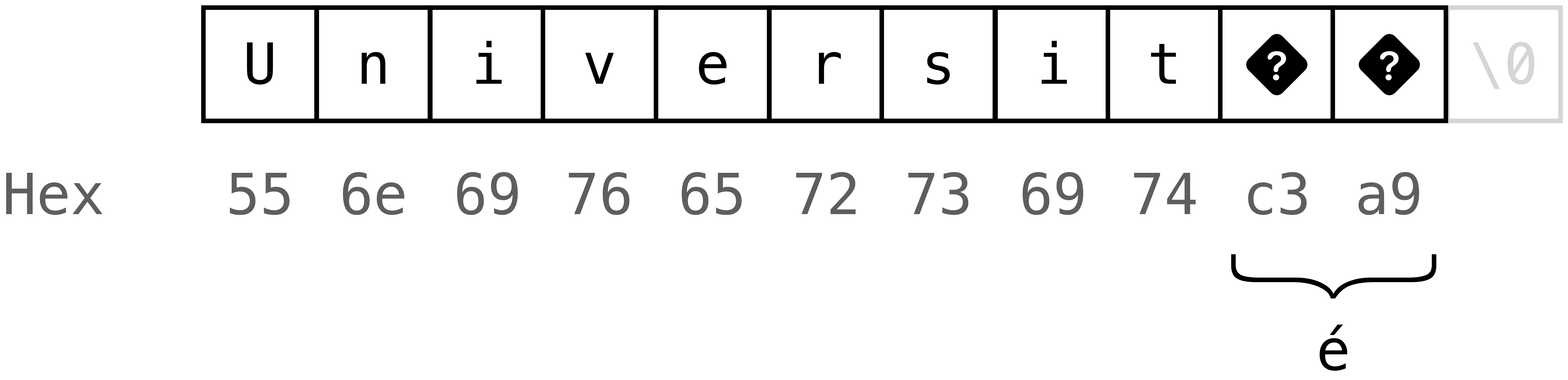
	U	n	i	v	e	r	s	i	t	é	\0
Hex	55	6e	69	76	65	72	73	69	74	c3a9	

🤔 Non-ASCII character,
UTF-8 encoded over 2 bytes!

5. Unicode Strings: Define a pointer to a constant string "Université"

```c
char * s = "Université";
```

```c
for(int i = 0; i < strlen(s); i++)
        printf("%02x ", s[i]);
```



	U	n	i	v	e	r	s	i	t	�	�	\0
Hex	55	6e	69	76	65	72	73	69	74	c3	a9	

$\underbrace{\text{c3 a9}}_{\text{é}}$

5c. Redo this question by defining a (mutable) char array initialised with the constant string “Université"

```
char * s = “Université";
```

`strlen(s)` → **11**

`sizeof(s)` → **8** (size of a pointer)

```
char s[] = “Université";
```

`strlen(s)` → **11**

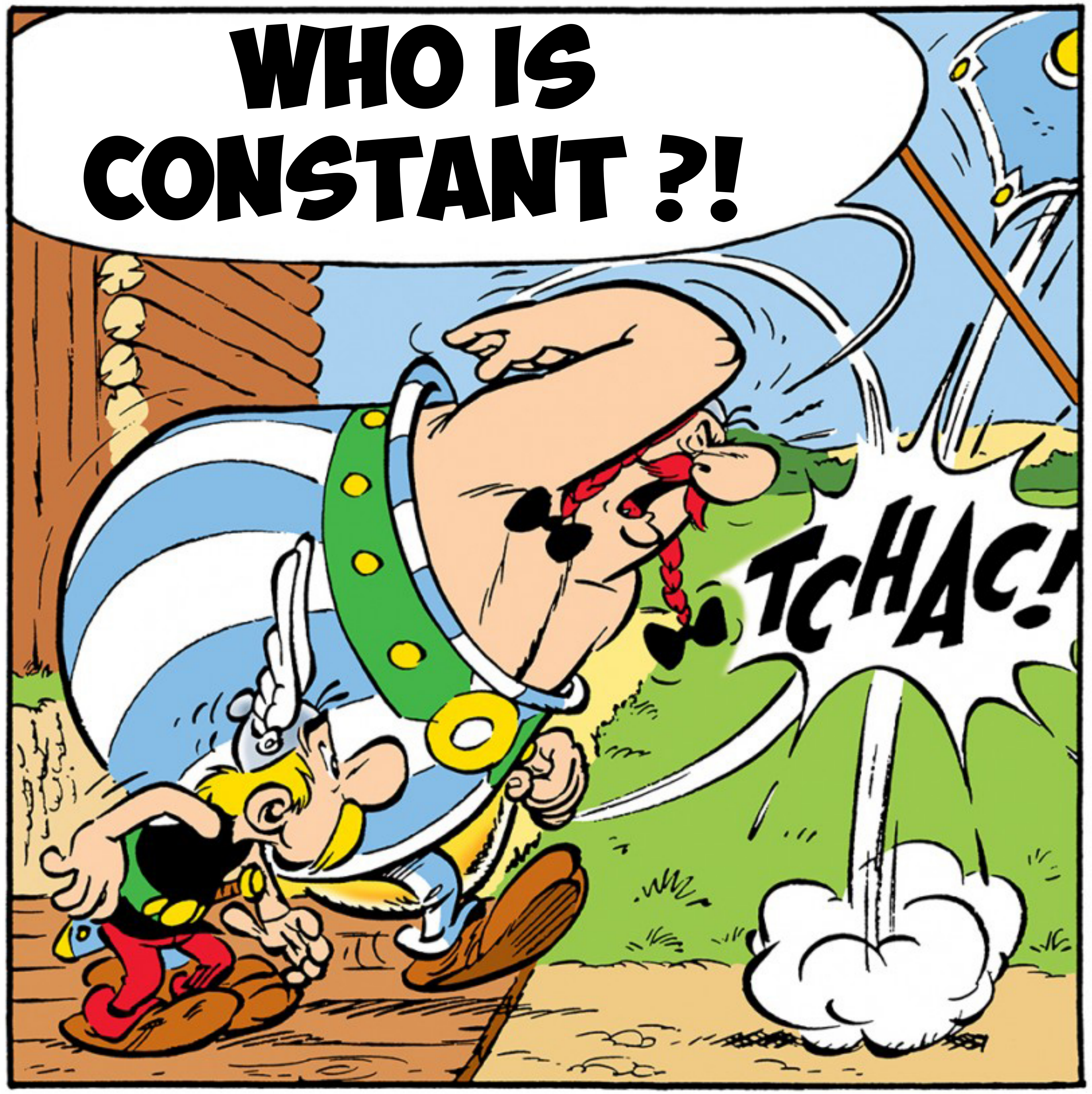
`sizeof(s)` → **12** (size of the array, including ‘\0’)

```
char * p = s;
```

`strlen(s)` → **11**

`sizeof(s)` → **8** (size of a pointer)

Bonus 1

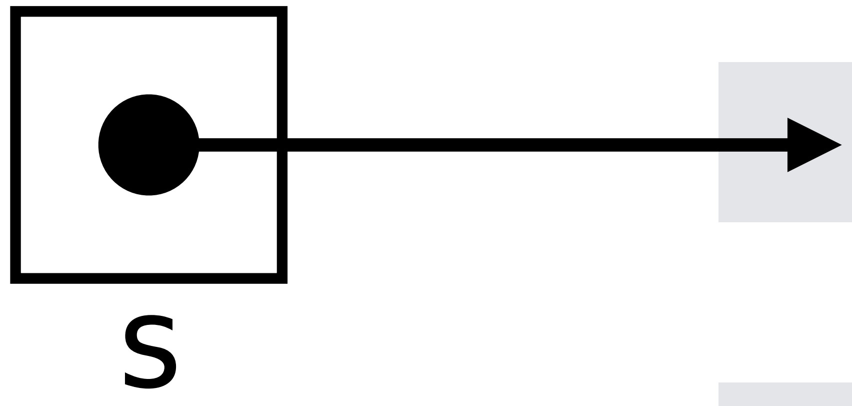


```
char * s = "Swift";
```

Mutable Pointer

Immutable String

```
char * s = "Swift";
```



Address	Value
...	...
100000F72	‘S’
100000F73	‘w’
100000F74	‘i’
...	
100000F77	‘\0’
...	...

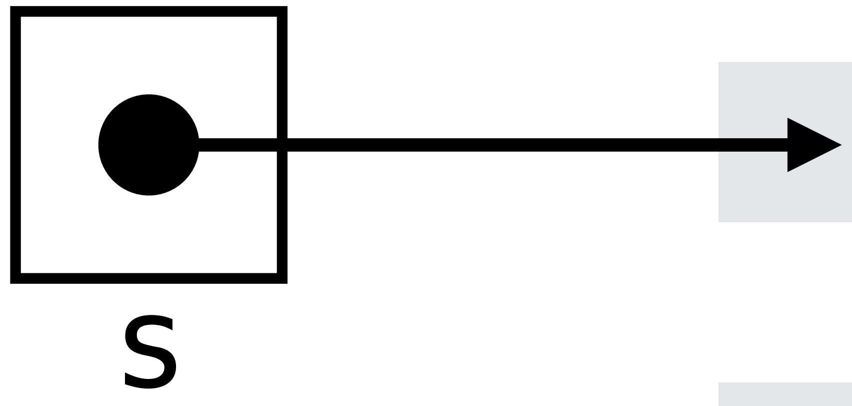
```
char * s = "Swift";
```

▶

```
char * s = “Python”;
```

Mutable Pointer

Changed reference 👌



Address	Value
...	...
100000F72	‘S’
100000F73	‘w’
100000F74	‘i’
...	
100000F77	‘\0’
...	...
100000F8F	‘P’
100000F90	‘y’
100000F91	‘t’
...	...
100000F95	‘\0’
...	...

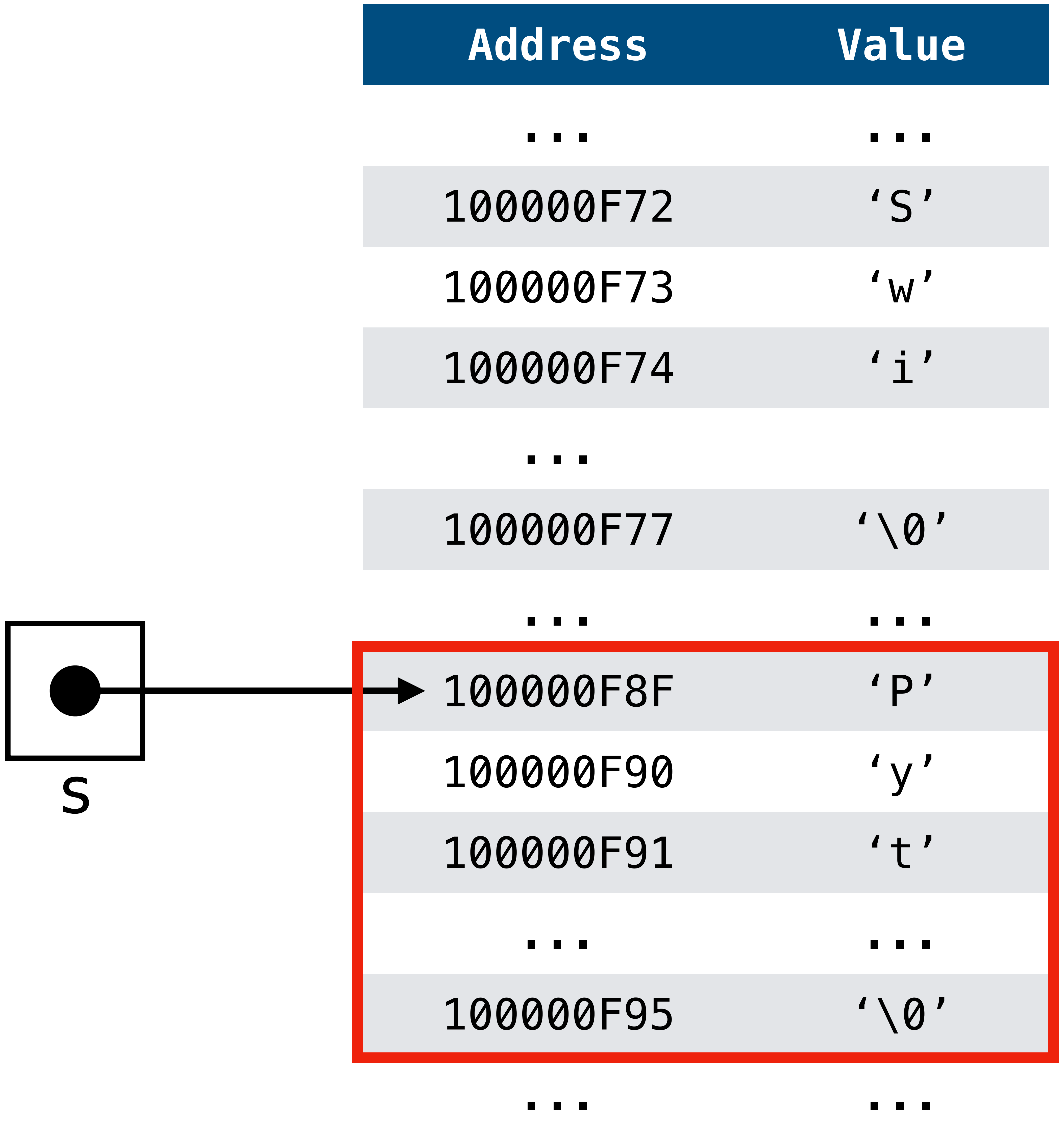
```
char * s = "Swift";
```

```
char * s = “Python”;
```

Immutable String

```
s[1] = ‘i’;
```

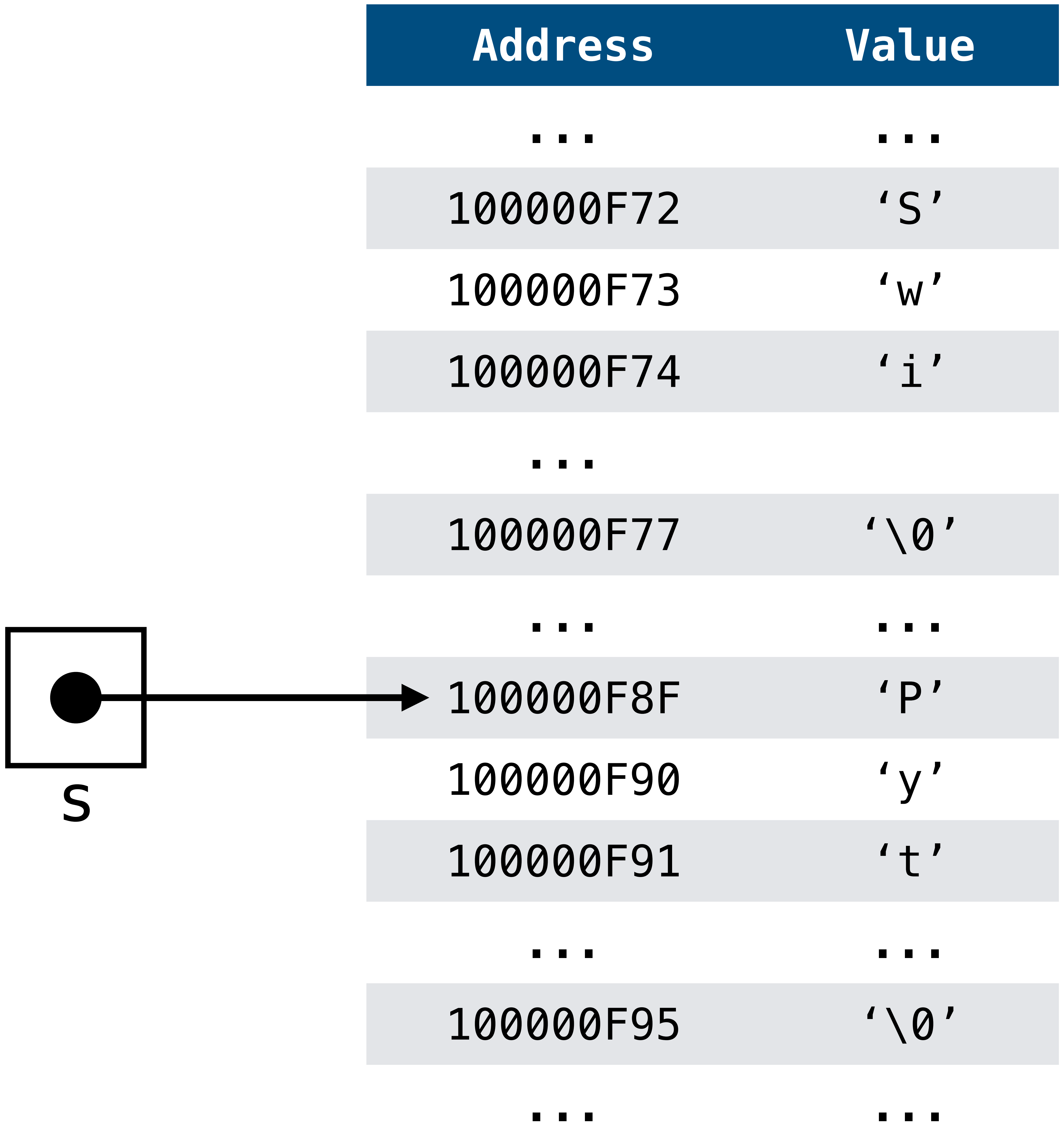
Constant strings cannot be changed 😱



Address	Value
...	...
100000F72	‘S’
100000F73	‘w’
100000F74	‘i’
...	
100000F77	‘\0’
...	...
100000F8F	‘P’
100000F90	‘y’
100000F91	‘t’
...	...
100000F95	‘\0’
...	...

```
char * s = “Python";

char * s = “Swift”;
```

Compiler Optimization: Reusing the read-only literal in the data segment



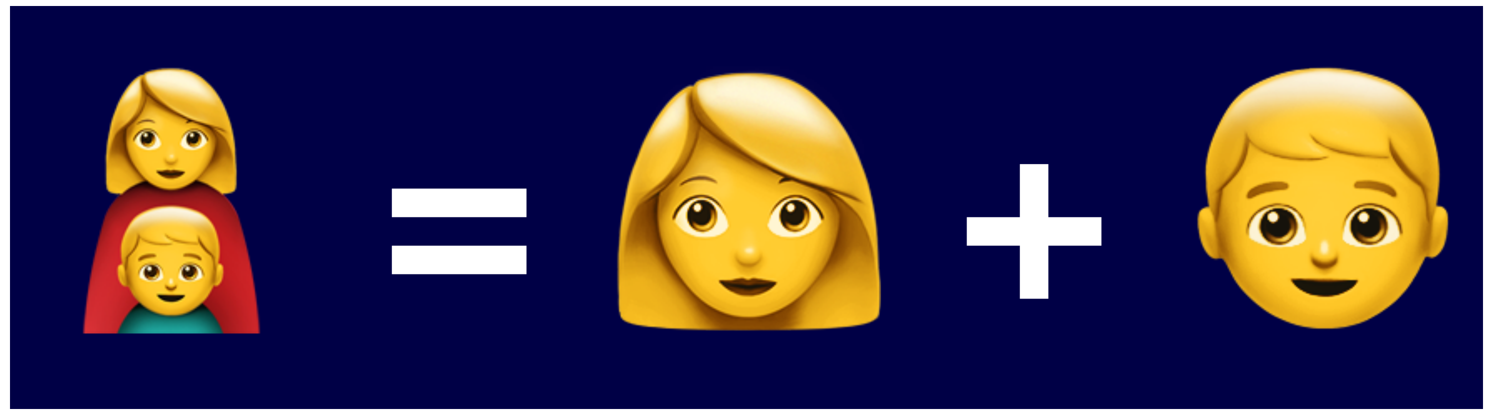
Address	Value
...	...
100000F72	‘S’
100000F73	‘w’
100000F74	‘i’
...	
100000F77	‘\0’
...	...
100000F8F	‘P’
100000F90	‘y’
100000F91	‘t’
...	...
100000F95	‘\0’
...	...

Bonus 2

```c
char emoji[] = "😂"; // f0 9f 98 82
*(emoji+3) += 11; // f0 9f 98 8d
printf("%s\n", emoji);
```

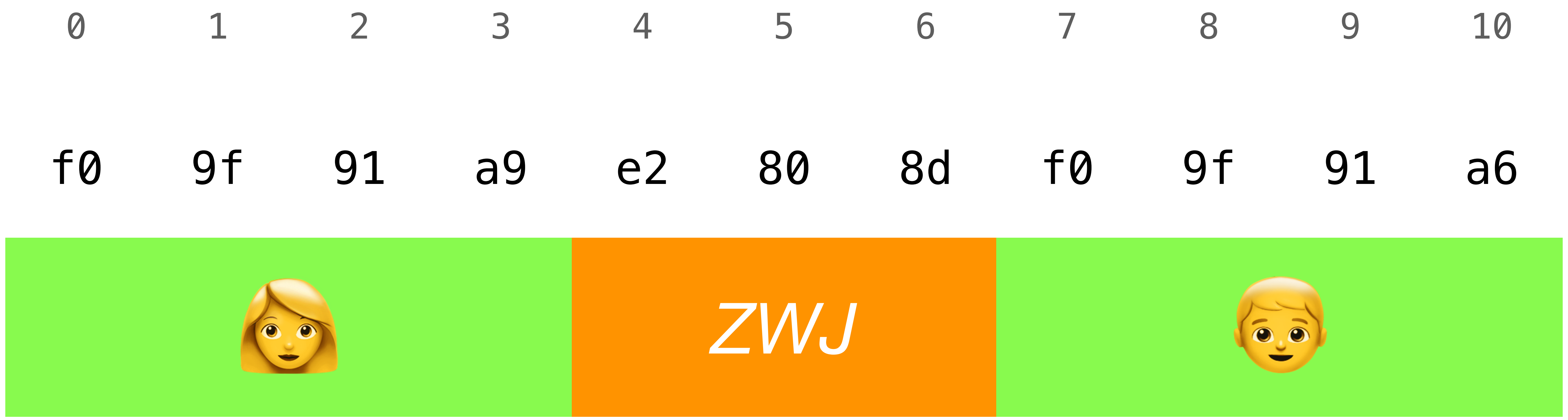
```
😍
```

```c
char * emoji = "👩‍👦";

char part1[5];
strncpy(part1, emoji, 4); // copy first 4 bytes
part1[4] = '\0';

char * part2 = emoji + 7;

printf("%s = %s + %s\n", emoji, part1, part2);
```



```
👩‍👦 = 👩 + 👦
```

About 👩‍👦



0	1	2	3	4	5	6	7	8	9	10
f0	9f	91	a9	e2	80	8d	f0	9f	91	a6

🤓 **Zero Width Joiner (“zwidge”):** Invisible character that joins several characters to create a new one. Used in Arabic/Indic scripts **… and emojis!**