Strings

Steven R. Bagley

Recap

- Programs are a series of statements
- Defined in functions
- Functions, loops and conditionals can alter program flow
- Data stored in variables or arrays
- Or pointed at by pointers

Strings

- C Strings are sequence of `chars` terminated by a null char `'\0'`
- Accessed by a pointer to the first character
- Create space for a new string by using a char array
- Access the characters either with pointers or as an array

There are other ways -- we'll see that later

Defining Strings

- Two ways of defining a string in C
  - As a *pointer* to the string
    `char *str = “Hello World”;`
  - As an array initialised with the string
    `char str[] = “Hello World”;`
- These are different — can modify the contents of the latter, but not the former

First defines a pointer to a string in the program code — unchangeable…
Go demo the difference…

String Processing

- Most string routines will iterate over every character in the string
- Using a loop
- And stop when they hit the `\0` character
- Classic example would be a string length routine

Strings as Arrays

- Need a string and an integer to hold the offset

```
char *string = “Hello World”;
int i=0;
```

- Loop conditional

```
while(string[i] != ‘\0’)
```

- Don’t forget to increment `i`

`i++;` or `i = i + 1;`

Go implement strlen…

Strings using Pointers

- Need a string
  `char *string = “Hello World”;`
- Also a pointer to a character
  `char *p = string;`
- Loop conditional
  `while(*p != ‘\0’)`
- Advance pointer to next character
  `p = p + 1;` or `p++;`

Set pointer to point to the first character

Speed Daemon

- Both methods will have the same effect
- But the array version has to do twice as much work
- Array access has to take the pointer to the base of the array and add the offset
- As well as incrementing `i`
- In the pointer version, we just move the pointer on by one

And in most cases, the CPU can automatically increment the pointer for us

Caesar Cipher

- Write an encryption program that uses the Caesar Cipher
- Caesar Cipher rotates each letter along the alphabet by a certain number
- So using the classic rotate 13, `A` becomes `N`, `B` becomes `O` etc.
- Wraps around so `M` is `Z`, `N` is `A`

Program Logic

- Visit every character in the string
- If it is a letter add 13 to its ASCII value
- If greater than `'Z'` wrap it back to `'A'`
- Store value back in string at same position
- Print it out
- Can use modular arithmetic to wrap things

Go program it up

Reading a string

- Could use `scanf()` to read text
- But this only reads one word
- There's another routine `gets()` which returns a complete line

gets

- `char *gets(char *s)`
  Takes a pointer to a string to read the line into
- Must be enough space to store the string; `gets()` won't check, that's your job
- Returns a pointer to the buffer, or `NULL` if an end of file occurs
- Newline character is *not* returned

gets

- However, there's a problem with `gets()`
- It doesn't check how large the buffer is
- If the user types in too many characters it will overflow the buffer
- Overwriting valid memory and causing the program to crash…

Getting a line

- `char *fgets(char *s, int n, FILE *stream)`
  Takes a pointer to a string to read the line into
- Reads at most `n` characters
- Need to tell it what `stream` to read from, we use `stdin`
- *Will* return the newline character

if it gets to the end of the line...

NULL pointers

- The `NULL` pointer is used to say this doesn't point at anything
- If you try and access it, you'll almost certainly crash the machine
- But you can always test for it first
- To speed things up, `NULL` is defined to be `0`
- We'll see this in more detail later…

Case-sensitivity

- Need to handle upper case and lower case separately
- Currently handle it manually
- Can use `isupper()` and `islower()` instead
- Then adjust accordingly
- But we can optimise our routine…

Changing the rotation

- Currently, we have a fixed rotation of 13
- Be nice if we could specify this
- One way to do this would be to allow us to put the rotation value on the command line
- So we pass it to the program as we call it

Command Line

- C passes the command line argument to `main` as an array of strings
- Generally, called `argv`
- Also passes a count of how many arguments there are, generally called `argc`
- First argument is always the name of the program

argv - argument vector

Command Line

- Can read the rotation value from here
- Provide a default of `13` if not specified (check how many arguments passed using `argc`)
- But need to convert string to an `int`
- Can use `scanf`, well, a special version for strings, `sscanf`

sscanf()

- Works identically to the normal scanf
- Reads characters from a string rather than the `stdin`
- First parameter is the string to read the characters from

```
int sscanf(char *s, char *format, …);
```

Character Type

- C library provides a variety of routines for testing characters
- All of the form `int is…(int c)`
- Returns true (non-zero) or false (zero) depending whether the character is or not
- Must `#include <ctype.h>`

Function	Tests
`isascii(int c)`	Between `0` and `127`
`isalpha(int c)`	is it an alphabetic character?
`isdigit(int c)`	is it a digit? '`0-9`'
`isnumber(int c)`	Any number character (depends on location)
`isalnum(int c)`	Is it a digit or alphabetic chracter?
`ishexnumber(int c)`	Is it a hex digit '`0-9A-F`'

Some of the more common examples

Function	Tests
`islower(int c)`	Is it lower case?
`isupper(int c)`	is it upper case?
`isspace(int c)`	is it a space character? Including tabs, newlines etc.
`isprint(int c)`	is the character printable?
`int tolower(int c)`	Converts character to lowercase if uppercase
`int toupper(int c)`	Converts character to uppercase if lowercase

Some of the more common examples

Optimized Implementation

- Often, these routines are written in an optimized fashion
- Don't use repeated comparisons
- An array of `longs` – one for each character
- Each bit of `long` means something if set
- Bit 8 is alphabetic character, Bit 12 is lower case, etc…

Optimized Implementation

- Bit 8 is alphabetic character
- Bit 10 is digit character
- Bit 12 is lower case
- Bit 15 is upper case and so on for other bits
- So character ‘A’ would have bit 8, and bit 15 set (at least)

Optimized Implementation

- Can test whether a bit is set using a bitwise-AND (the & operator in C)
- So `__runetype['A'+1] & 0x00008000L` would test if it is upper-case
- If the bit is set this will return, `0x8000L` a non-zero value, so true
- If not, returns zero so false

+1 is so you can cope with EOF which is -1
Try being that clever in Java...

Lookup Table

- This is called a ‘Lookup Table’
- Rather than calculating the result, it looks it up in the array
- Very common practice
- Sometimes used to get approximations for complex calculations
- Trading memory space for time

(e.g. sine/cosine in games, colour transformations etc)

Common String Routines

- Some string routines are used a lot
- String Length routine
- Copy a string
- Concatenate one string onto another
- Find the first occurrence of a character in a string

See implementation of these

Pointers into Strings

- Often use pointers to access the individual characters in a string
- This works very effectively
- Can also use it to ‘chop’ strings in half by manipulating the pointer that is passed to string routines
- Or the position of the null terminator

Printing a String

- Consider a function that prints a string
- Take a pointer to the first character
- Use the * operator to fetch the character
- If `'\0'` then end
- Use `putchar()` to output each character
- And loop

Go write it...

```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```

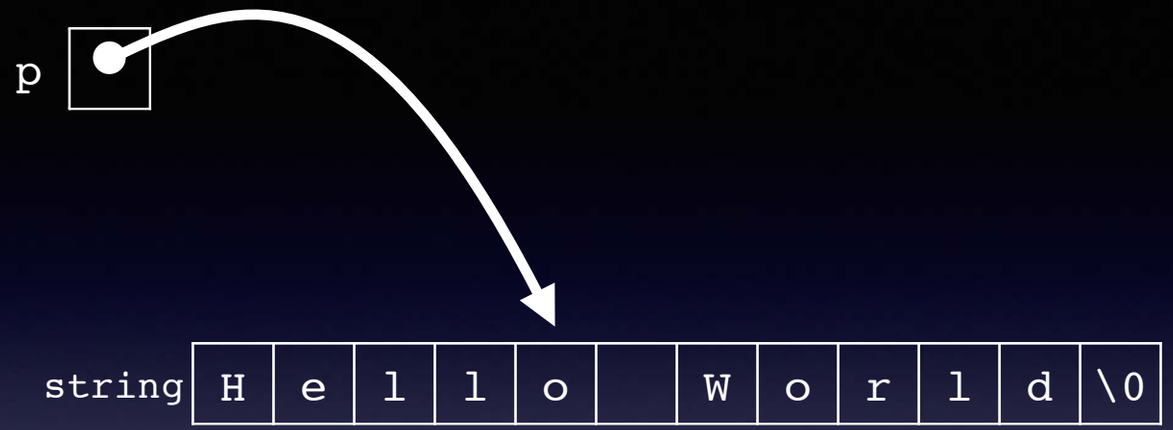

```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```

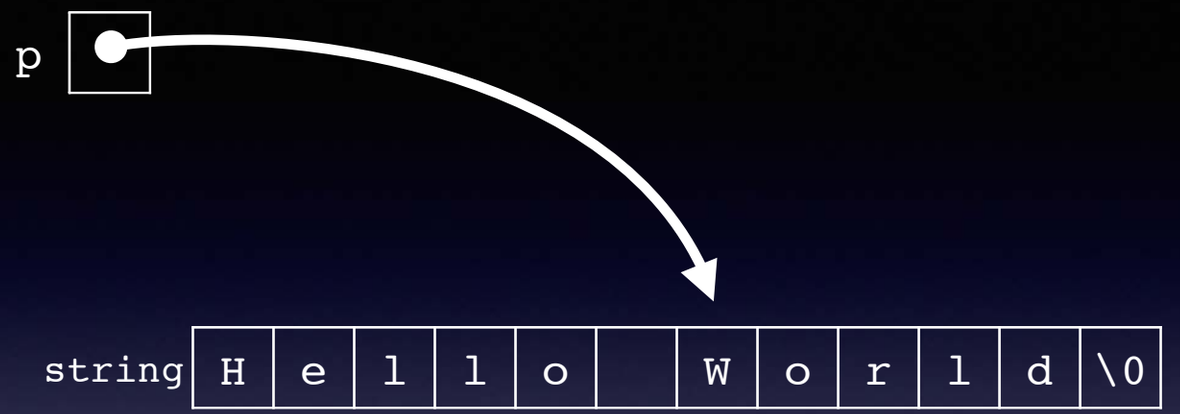



```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```

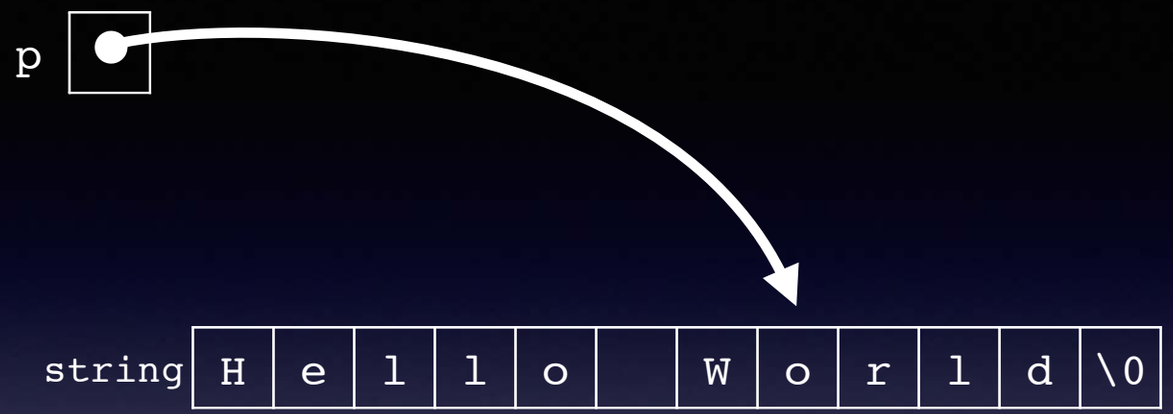

```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```

Moving Pointers

- Routine prints from the first character
- Since it is given a pointer to the *first character*
- But if we give it a pointer to the *sixth* character `(string + 6)`
- It will start printing from the sixth character



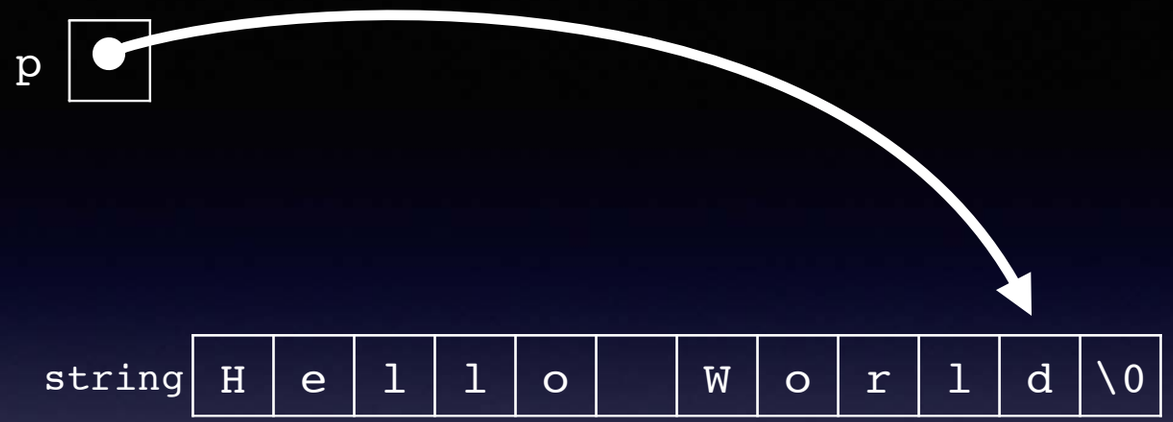

```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```




```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```




```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```


```
void putstring(char *string)
{
    char *p = string;

    while(*p != '\0')
    {
       putchar(*p++);
    }
}
```

Common String Routines

- Some string routines are used a lot
  - String Length routine
  - Copy a string
  - Concatenate one string onto another
  - Find the first occurrence of a character in a string

See implementation of these

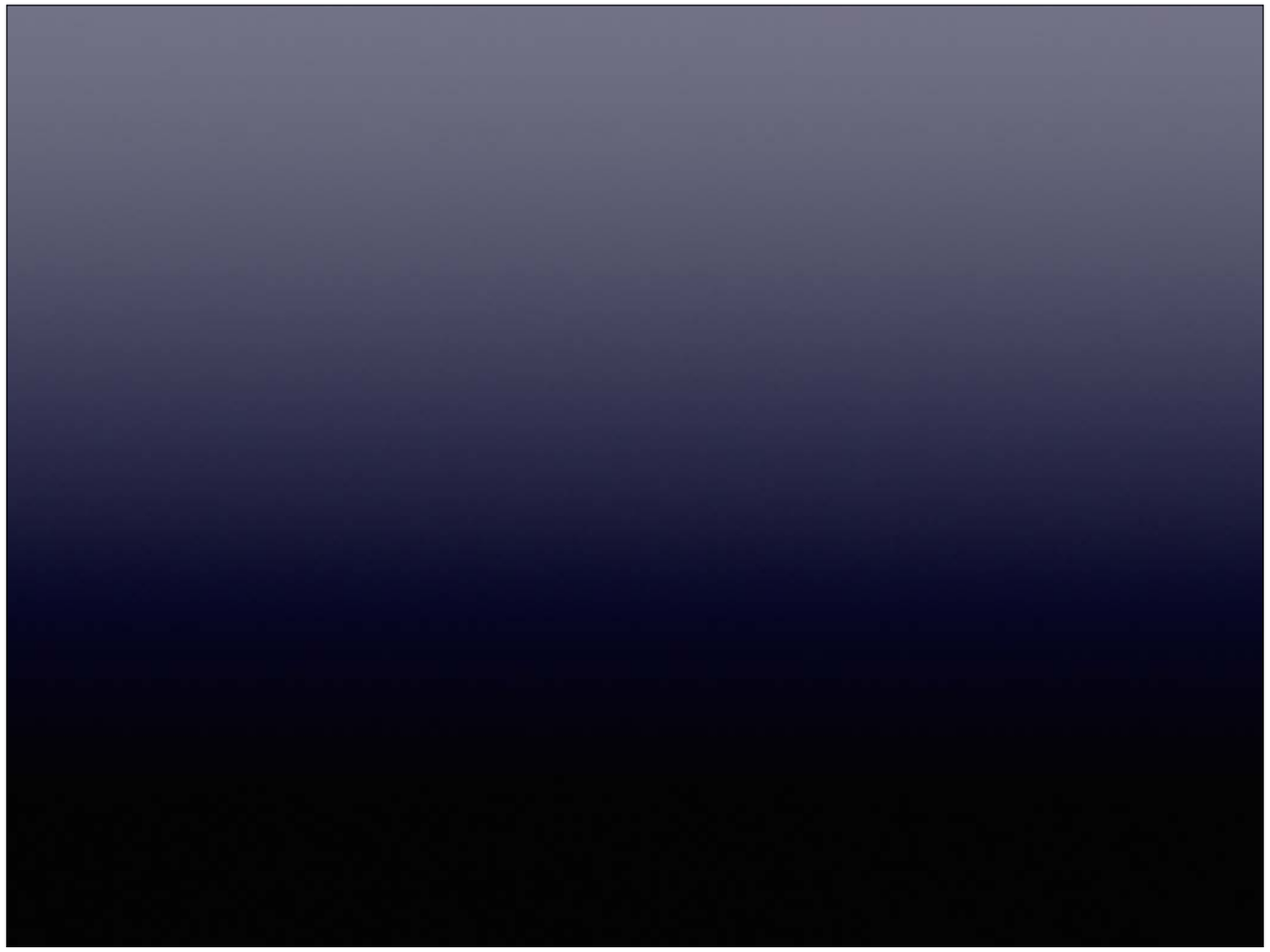
String Length

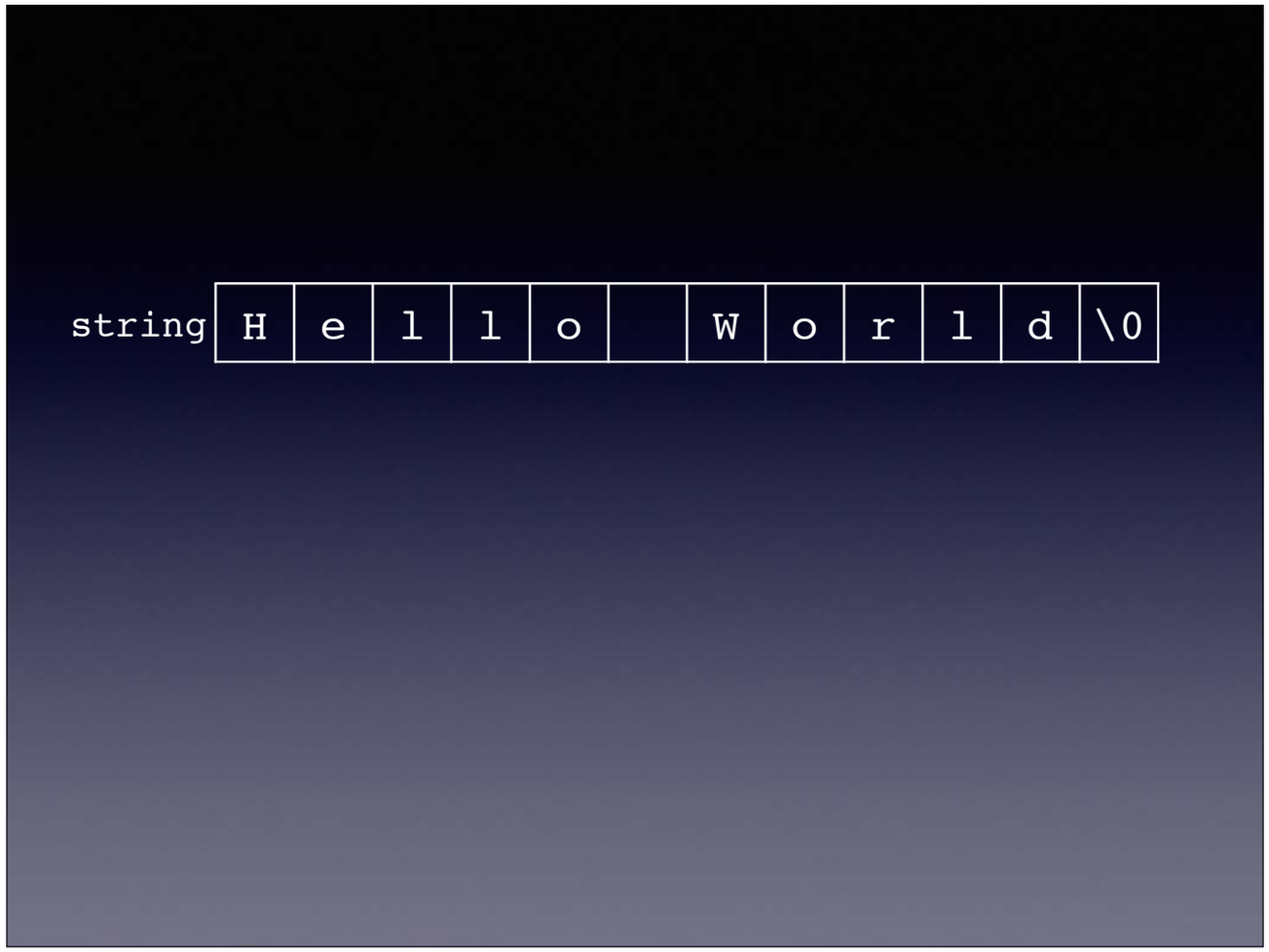
- Already saw this last lecture
- Step over every character
- Adding one to a counter
- When we hit the null character, we stop
- Counter contains the number of characters

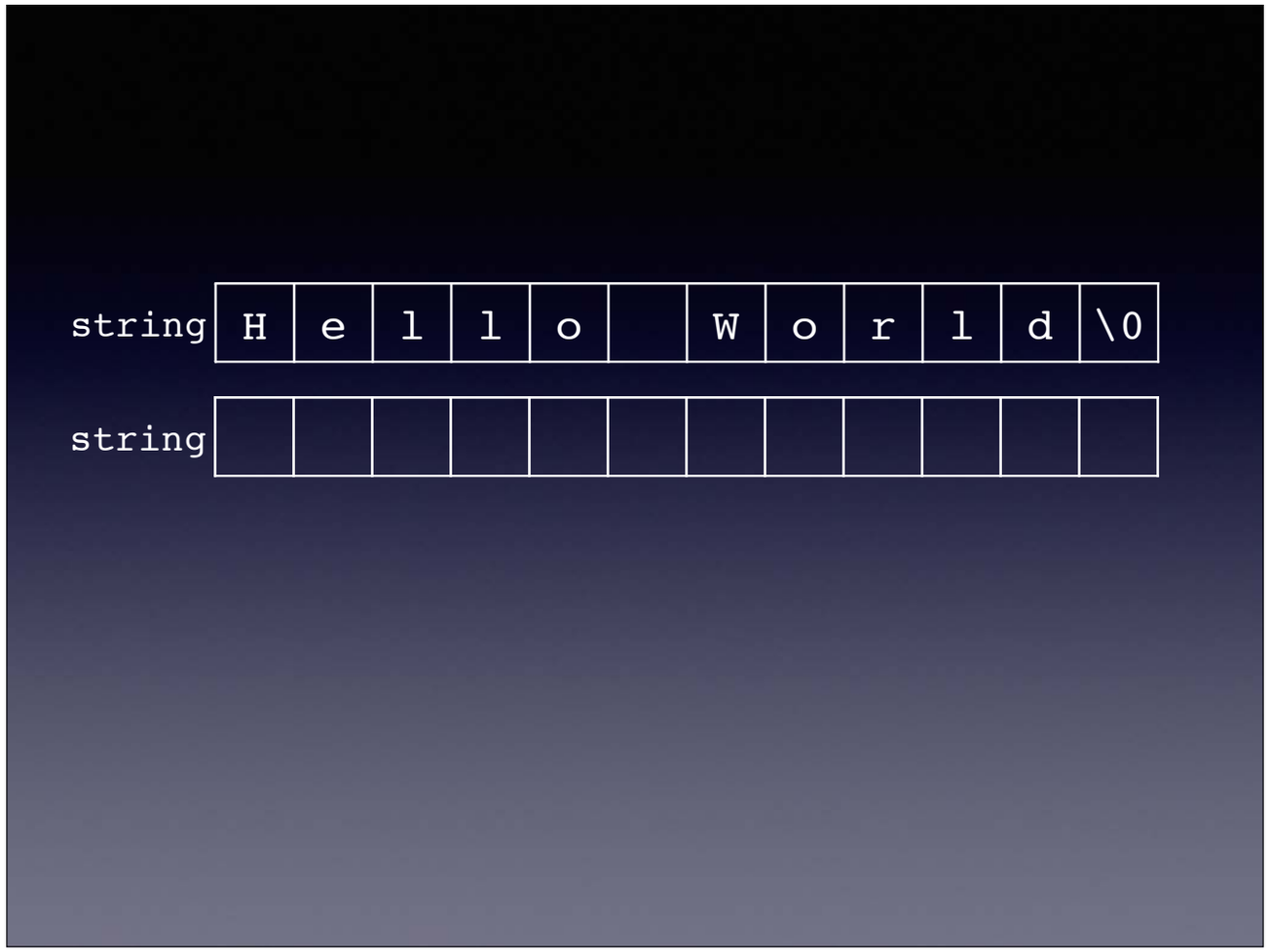
Again though we only count the number of characters from the pointer passed to the routine

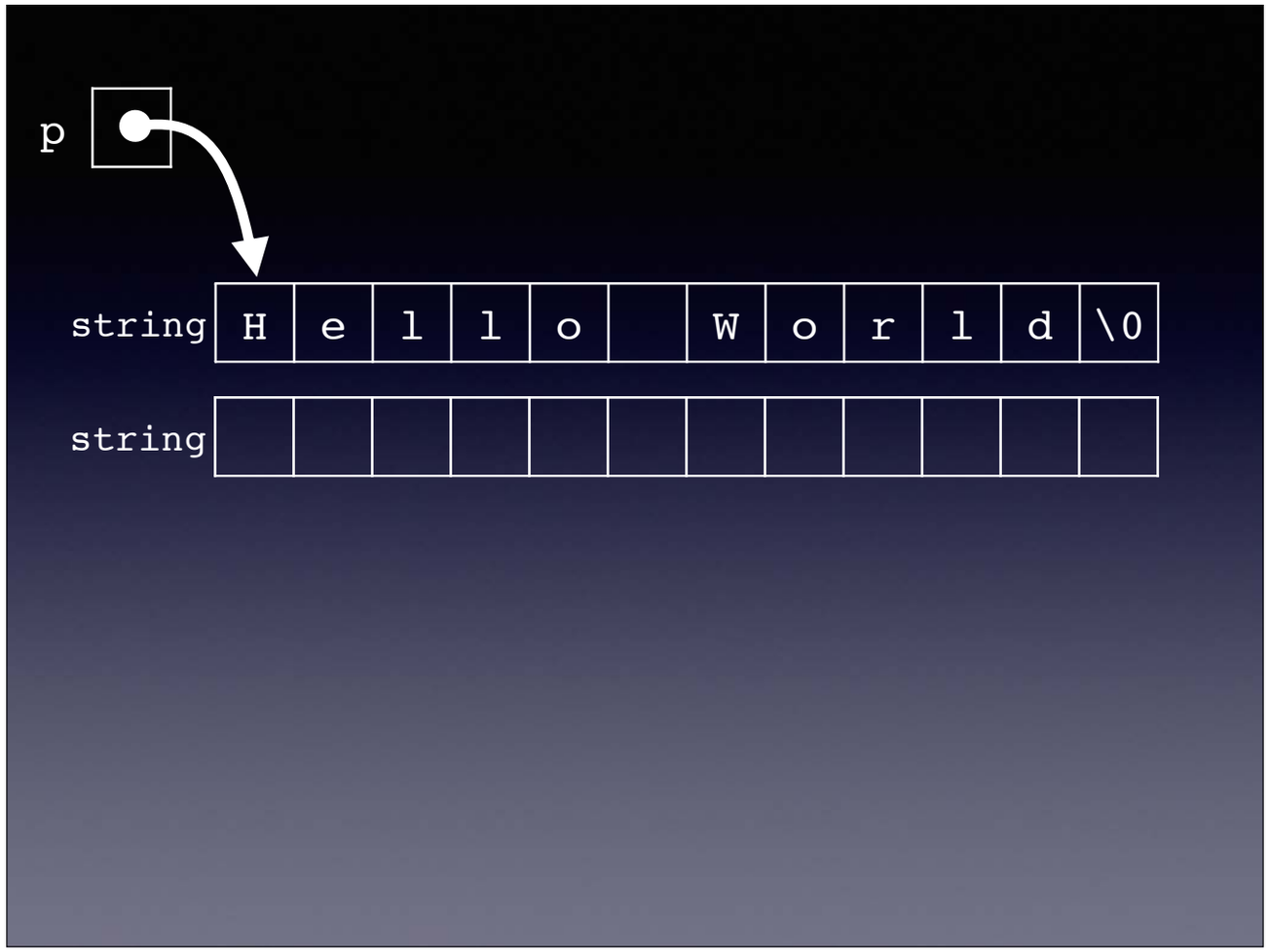
String Copy

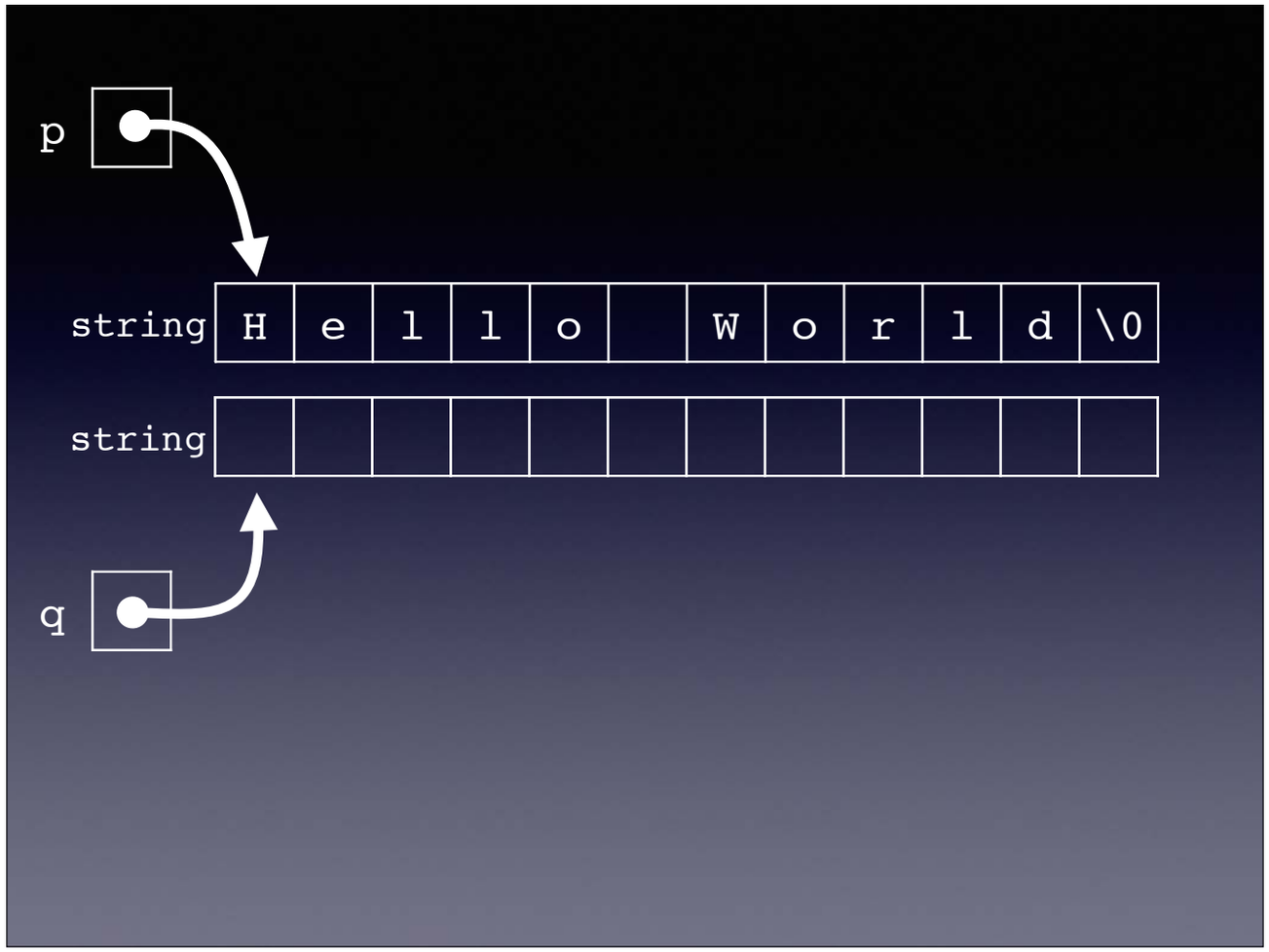
- Easy to do
- One pointer points at the source string
- Other points at the destination
- Copy value from source pointer to destination pointer
- Till we reach null character

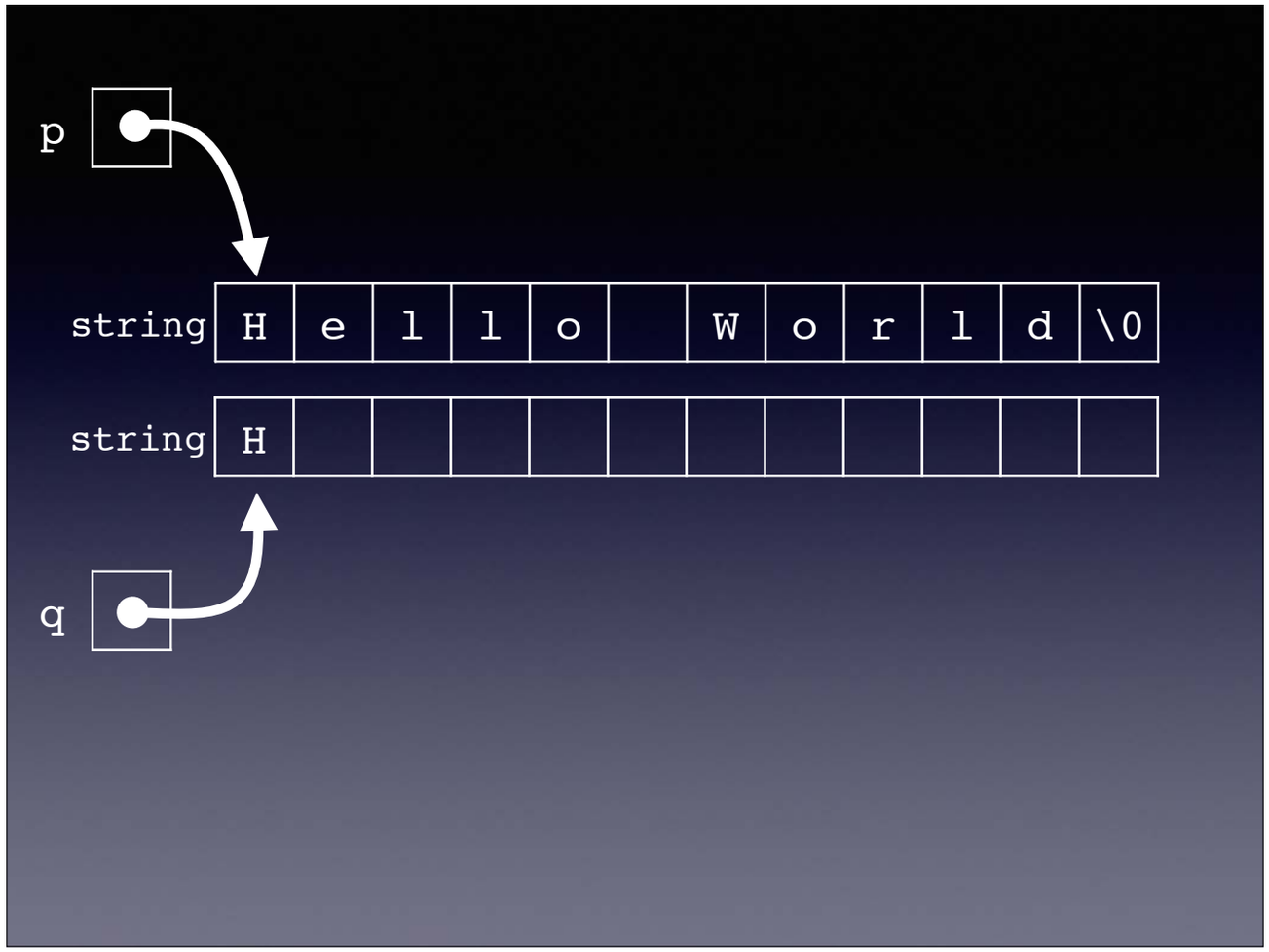


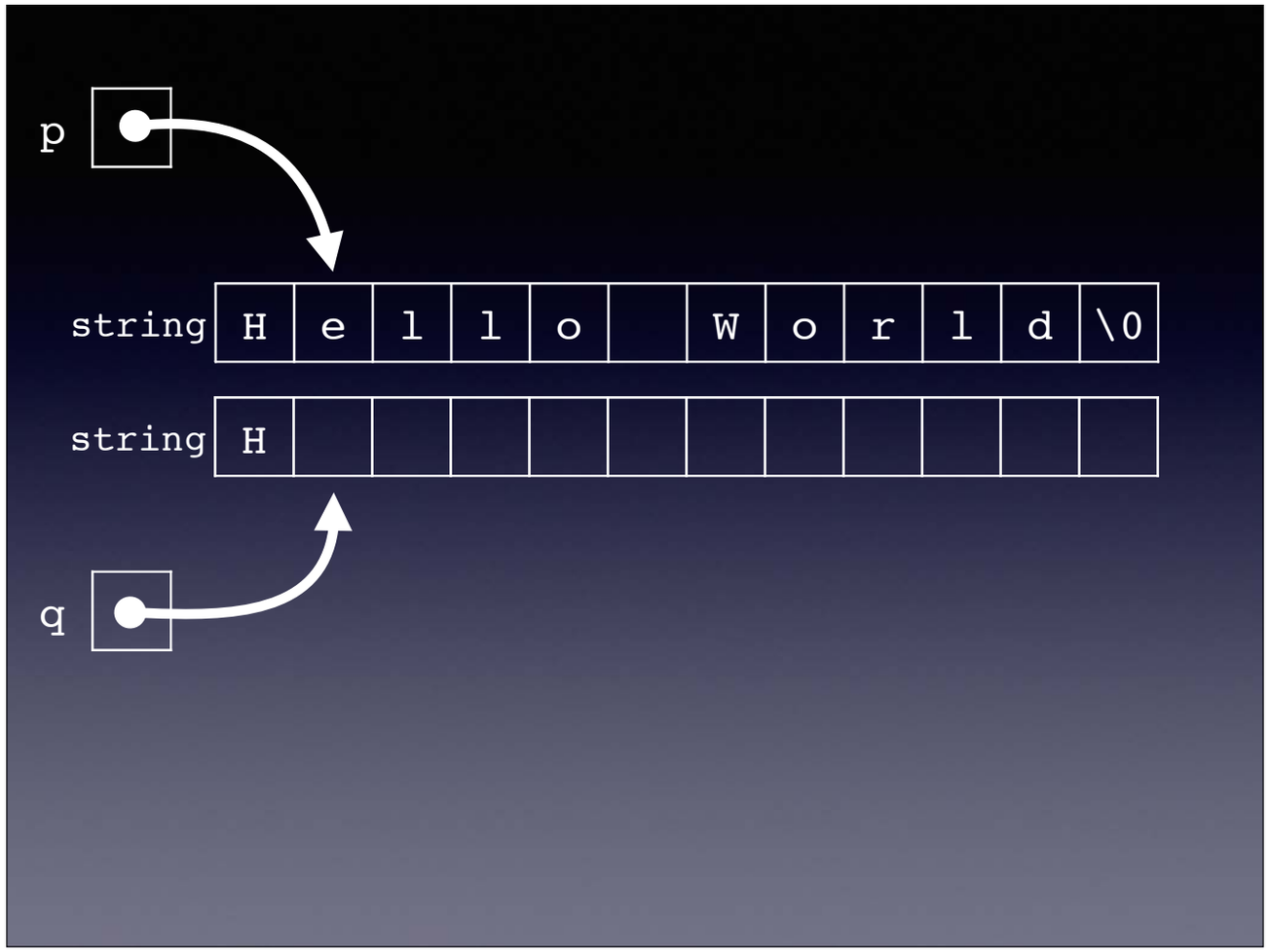

string
H
e
l
l
o
W
o
r
l
d
\0

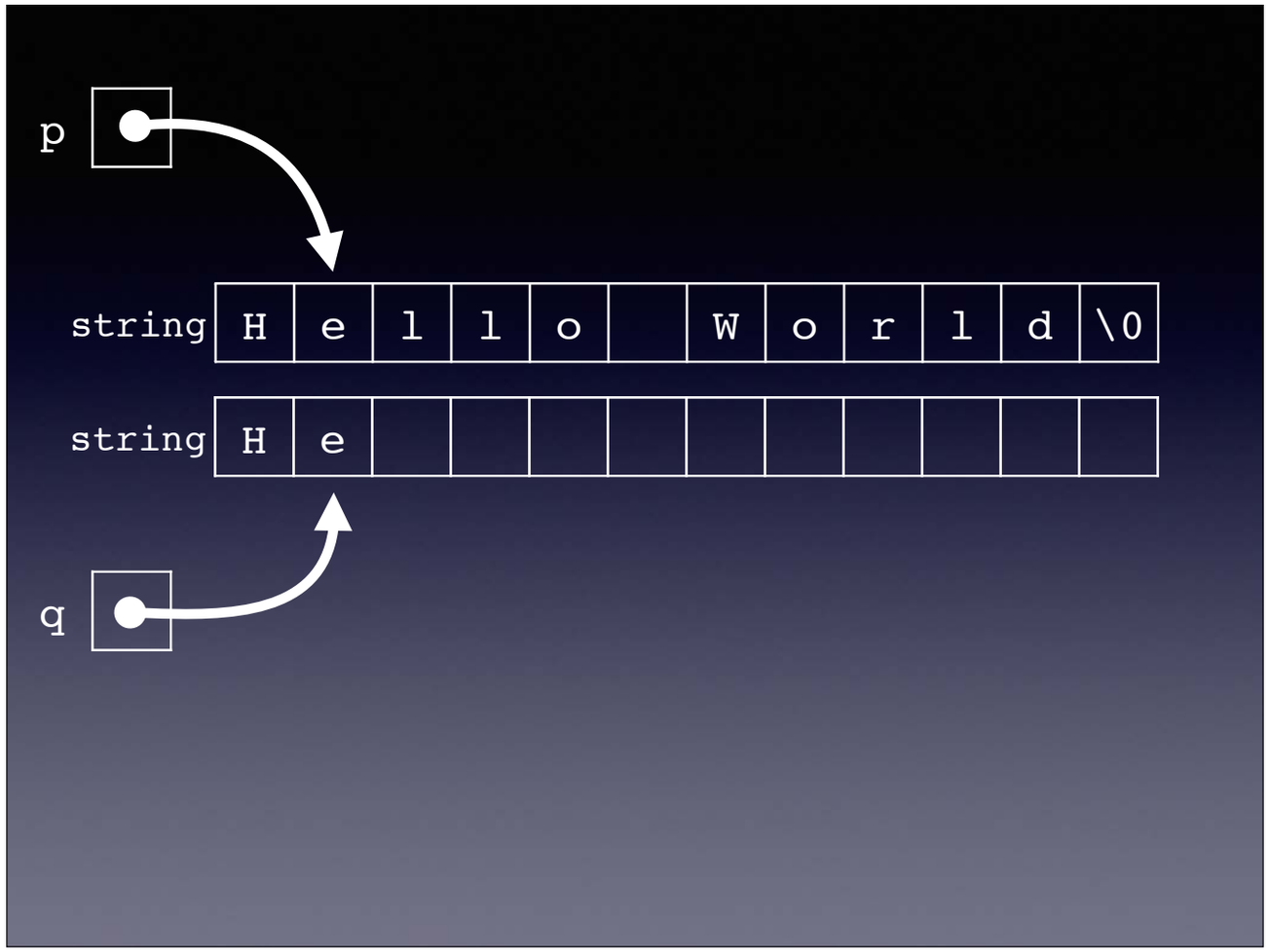

string
H e l l o W o r l d \0
string

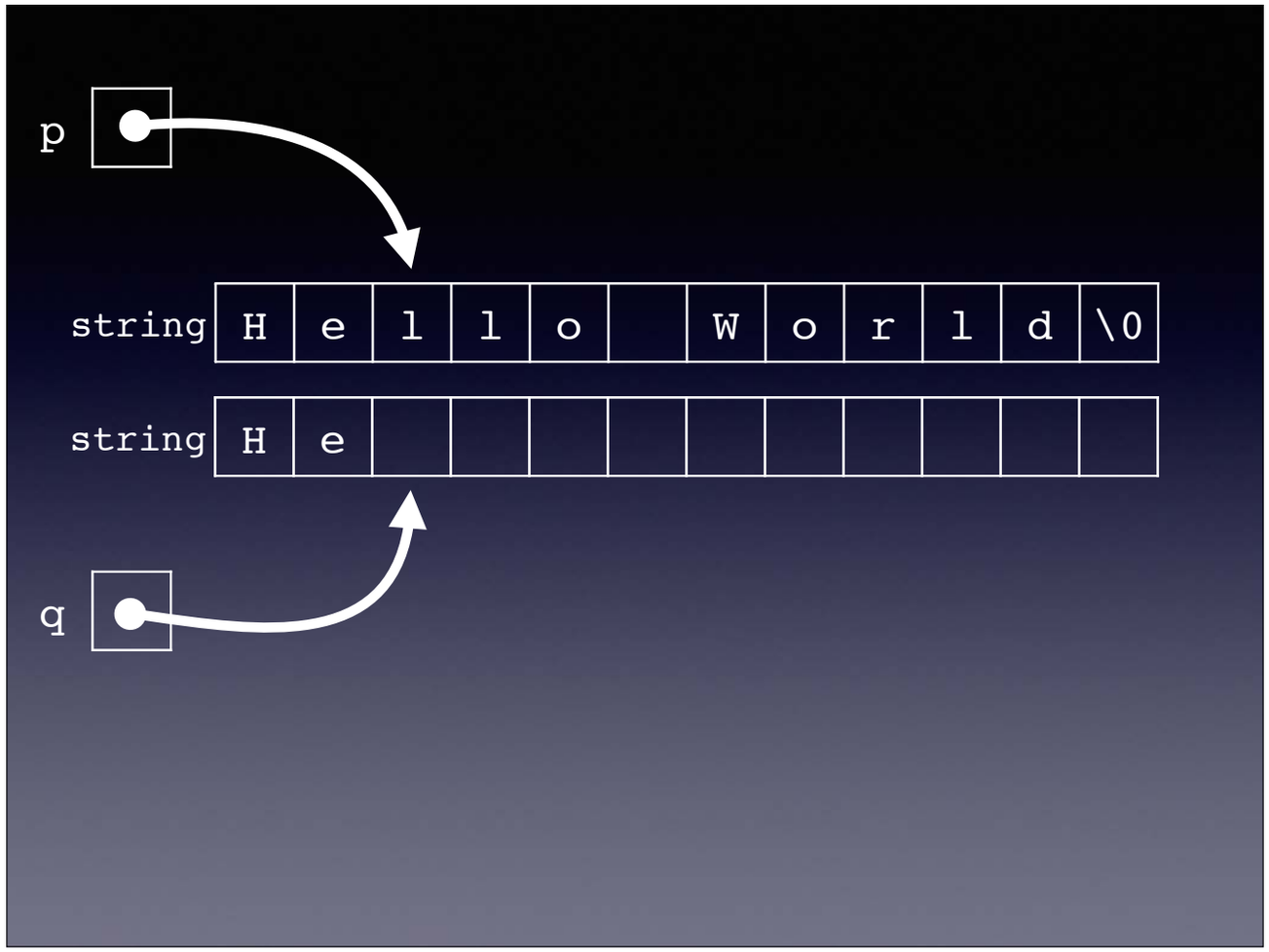

p
string
H e l l o W o r l d \0
string

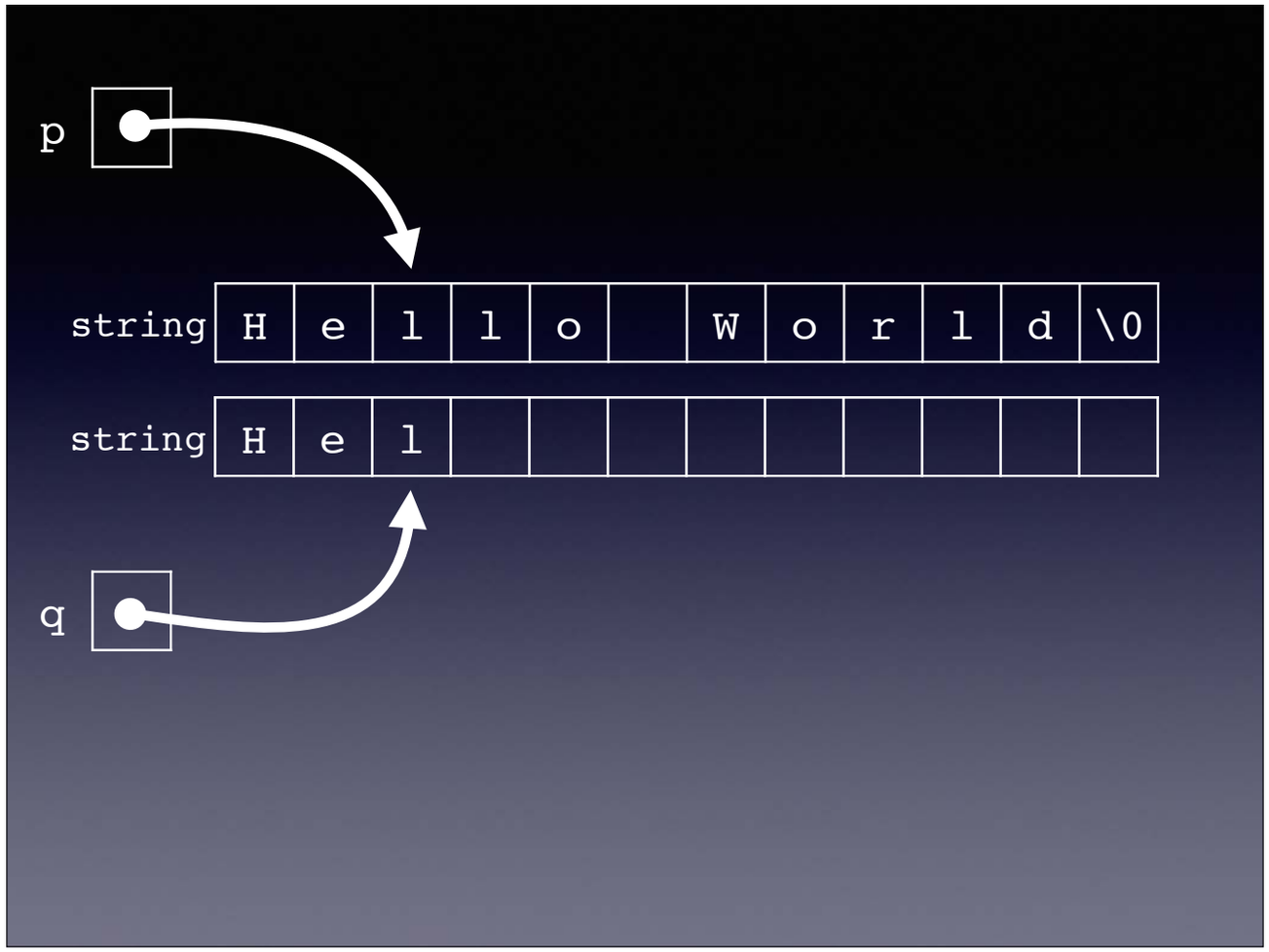

p
string
H e l l o W o r l d \0
string
q

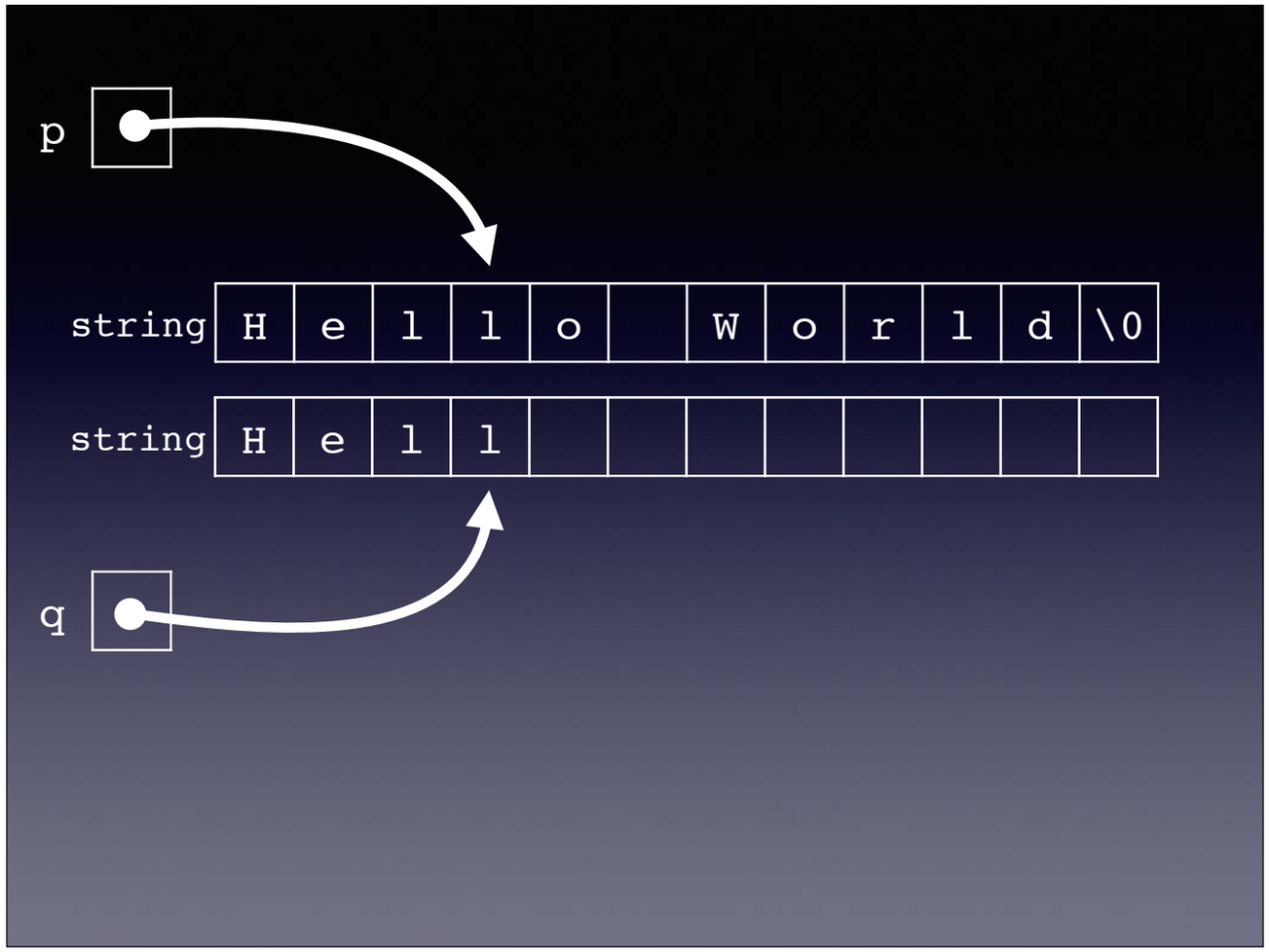

p
string
H
e
l
l
o
W
o
r
l
d
\0
string
H
q

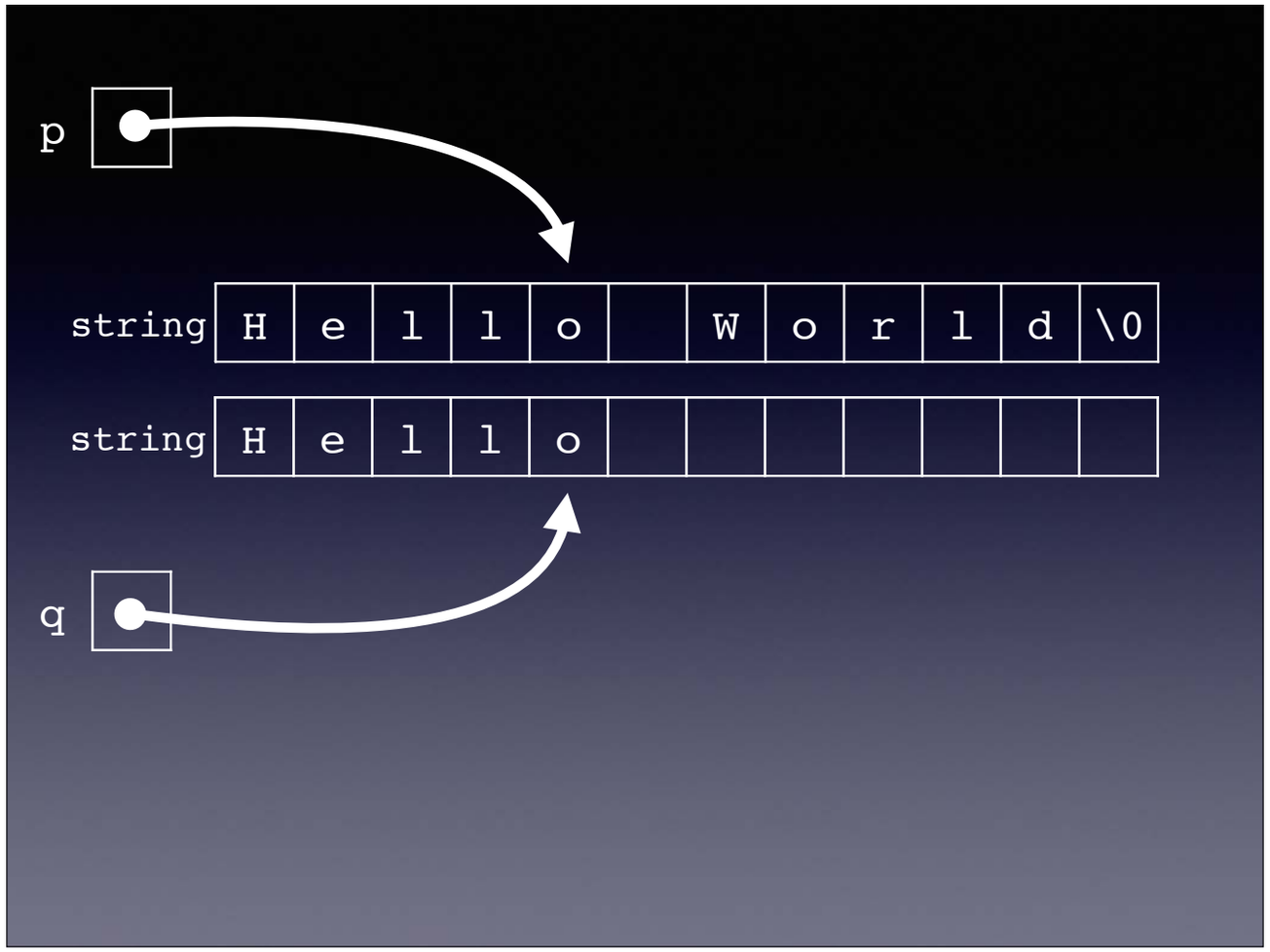

p
string
H e l l o W o r l d \0
string
H
q

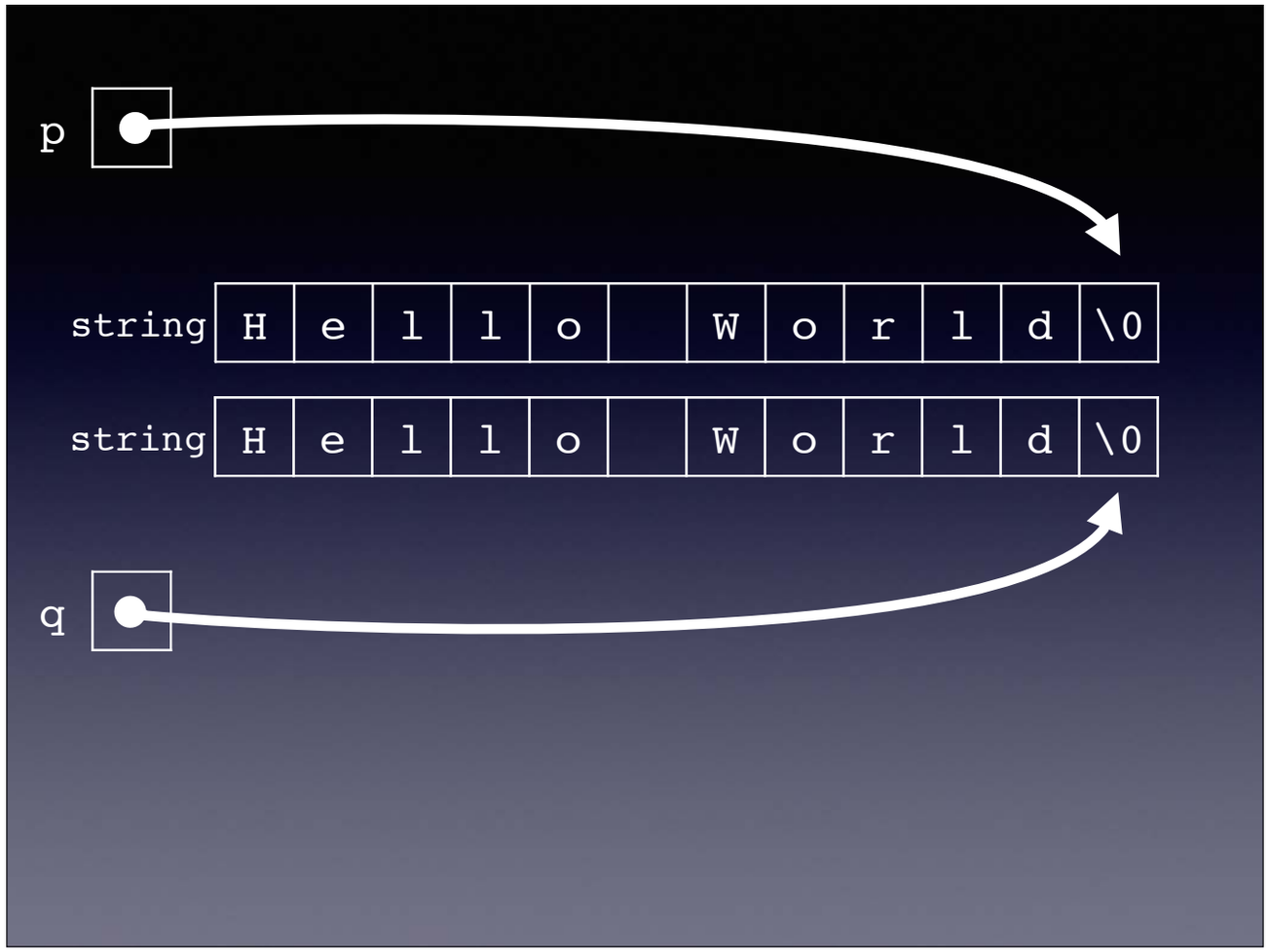

p
string
H
e
l
l
o
W
o
r
l
d
\0
string
H
e
q


p
string
H
e
l
l
o
W
o
r
l
d
\0
string
H
e
q


p
string
H
e
l
l
o
W
o
r
l
d
\0
string
H
e
l
q

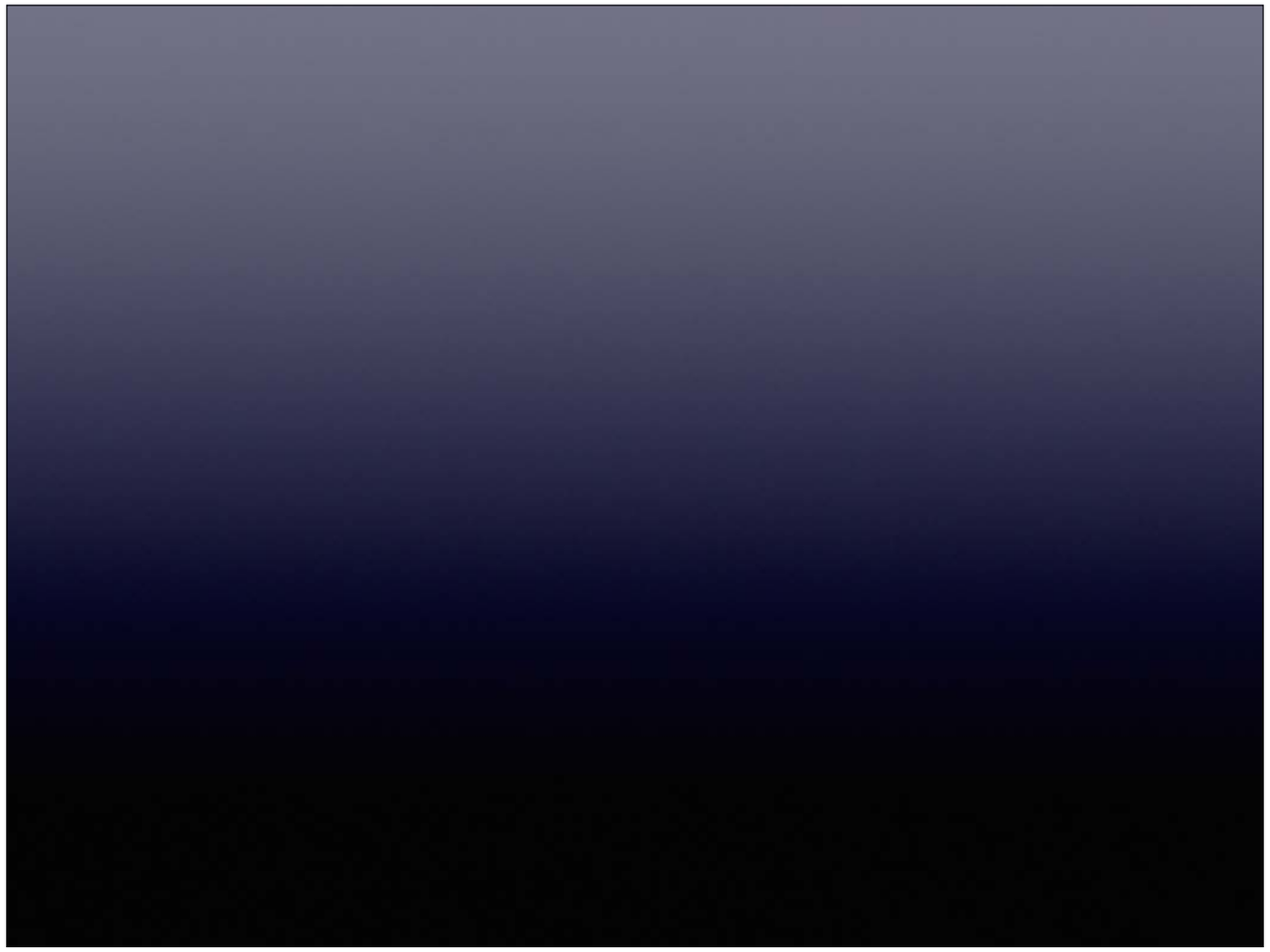

p
string
H
e
l
l
o
W
o
r
l
d
\0
string
H
e
l
l
q

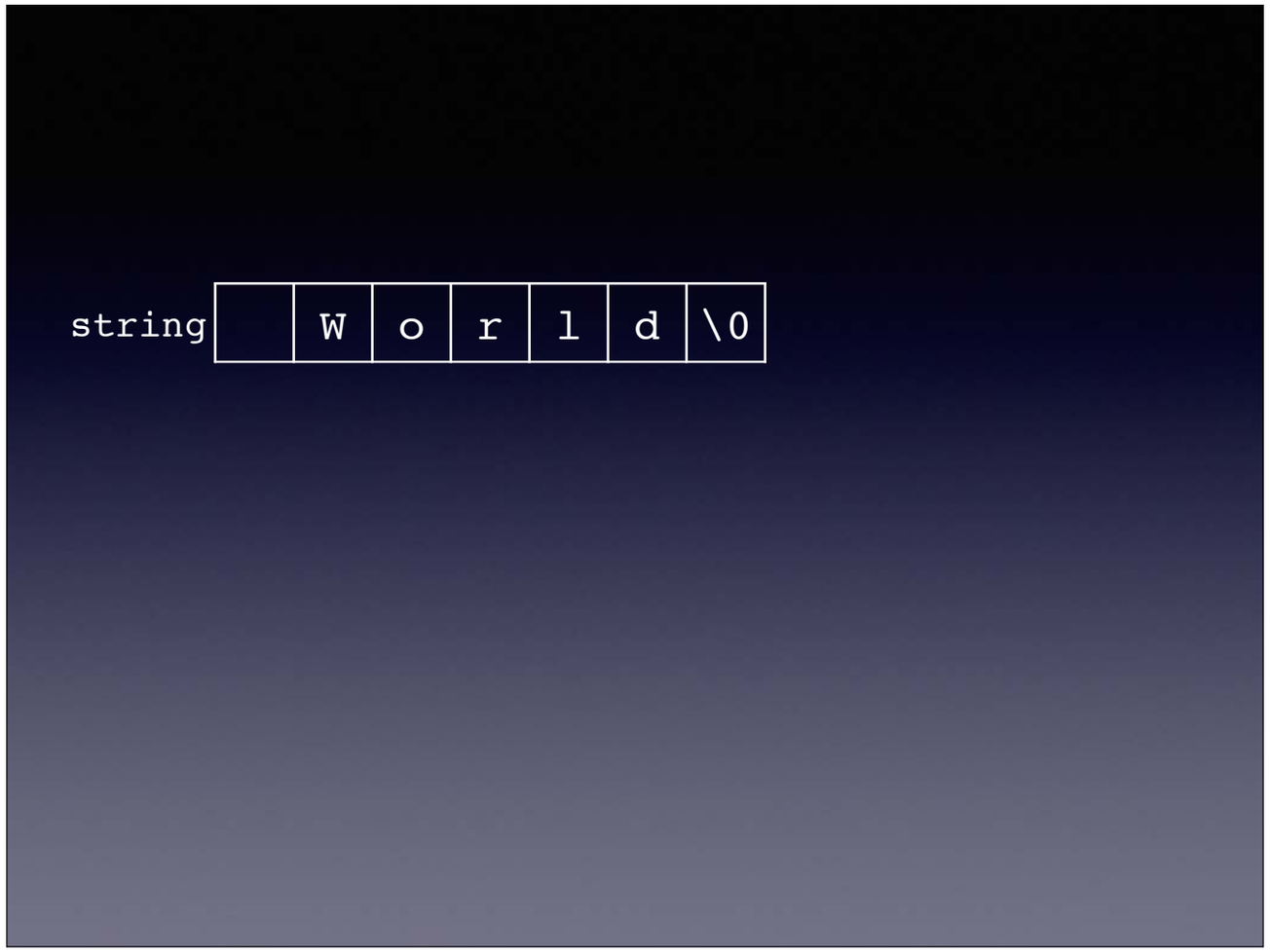

p
string
H e l l o W o r l d \0
string
H e l l o
q

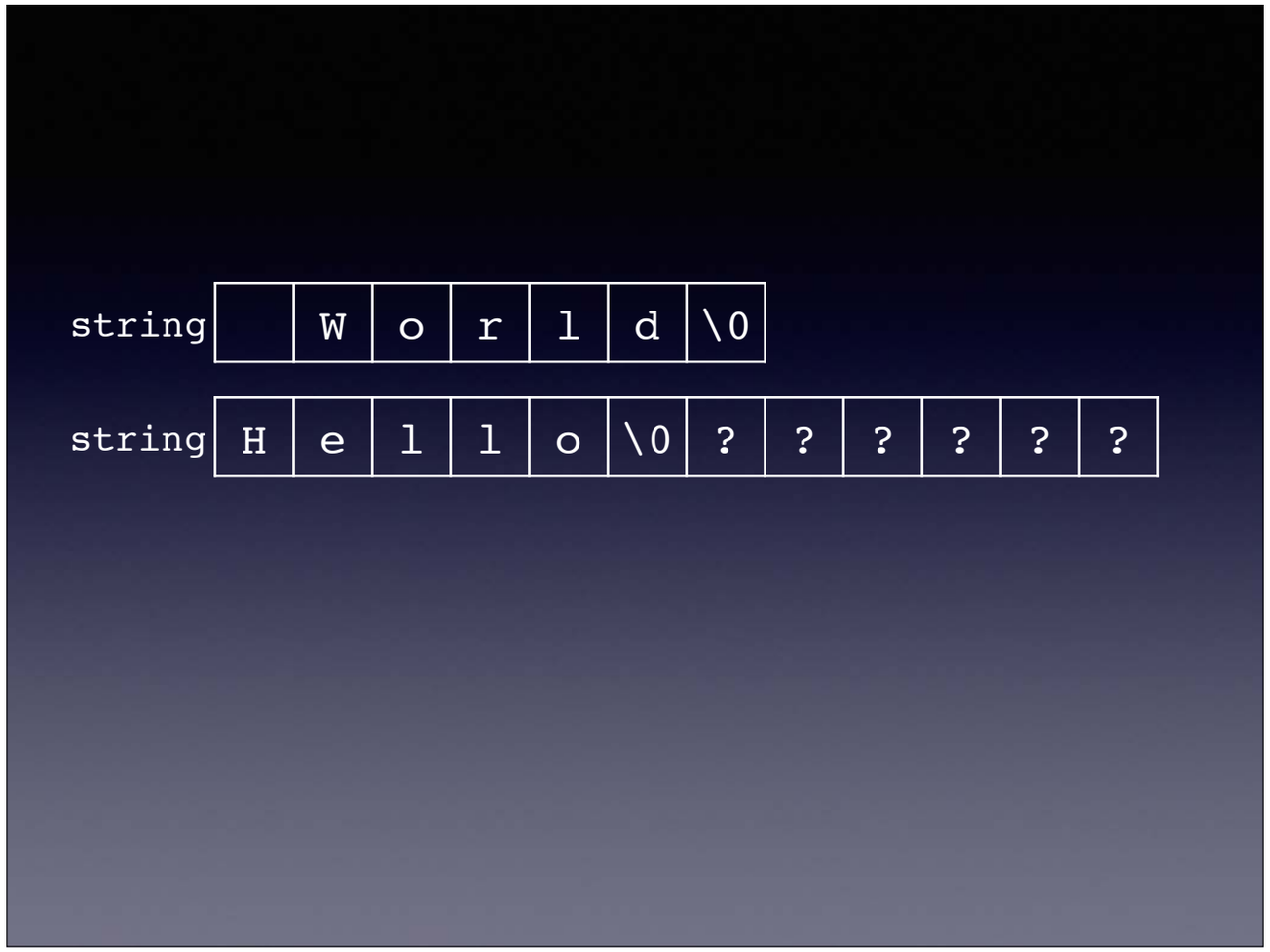

p
string H e l l o W o r l d \0
string H e l l o W o r l d \0
q

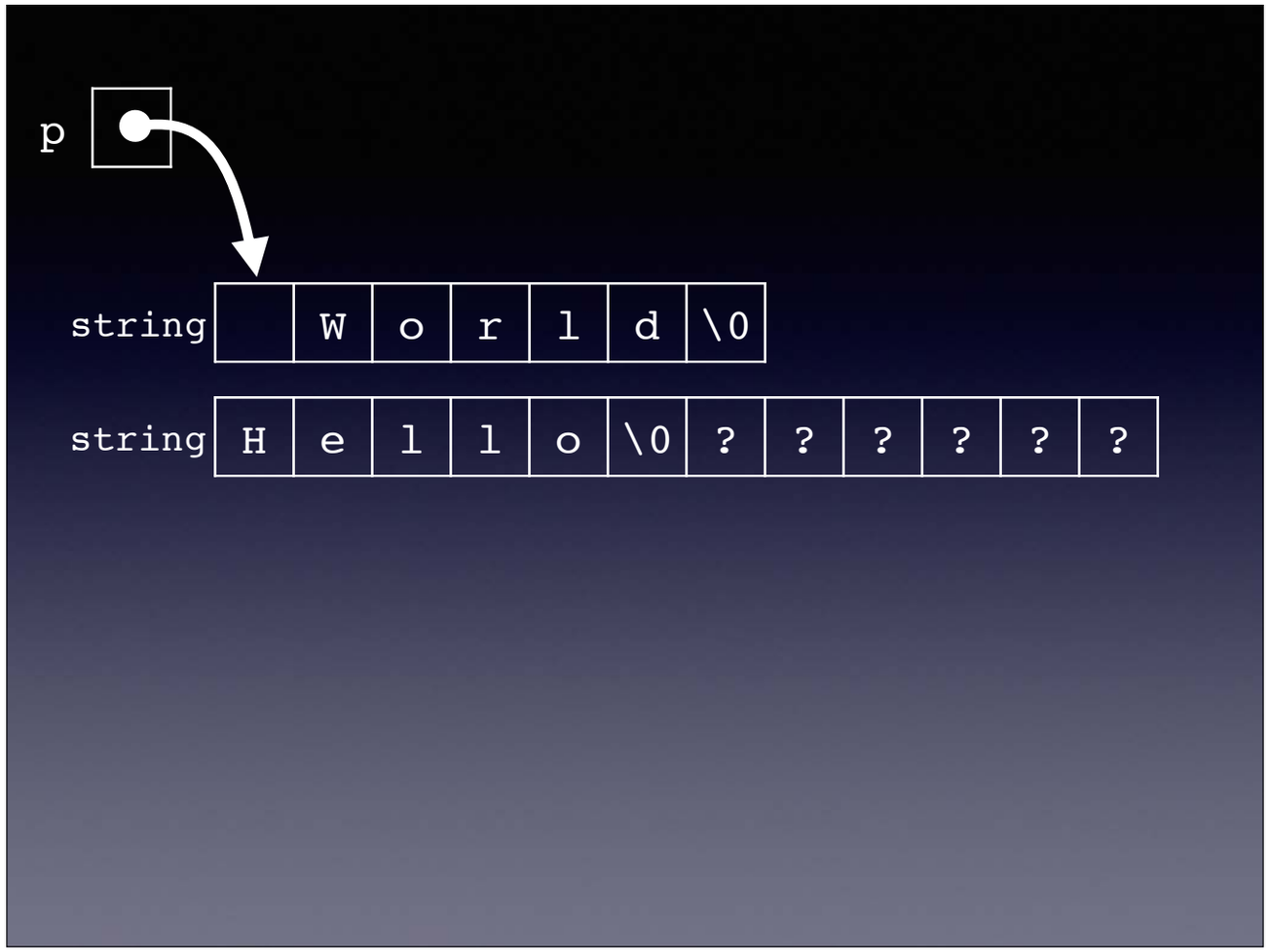
String Copy

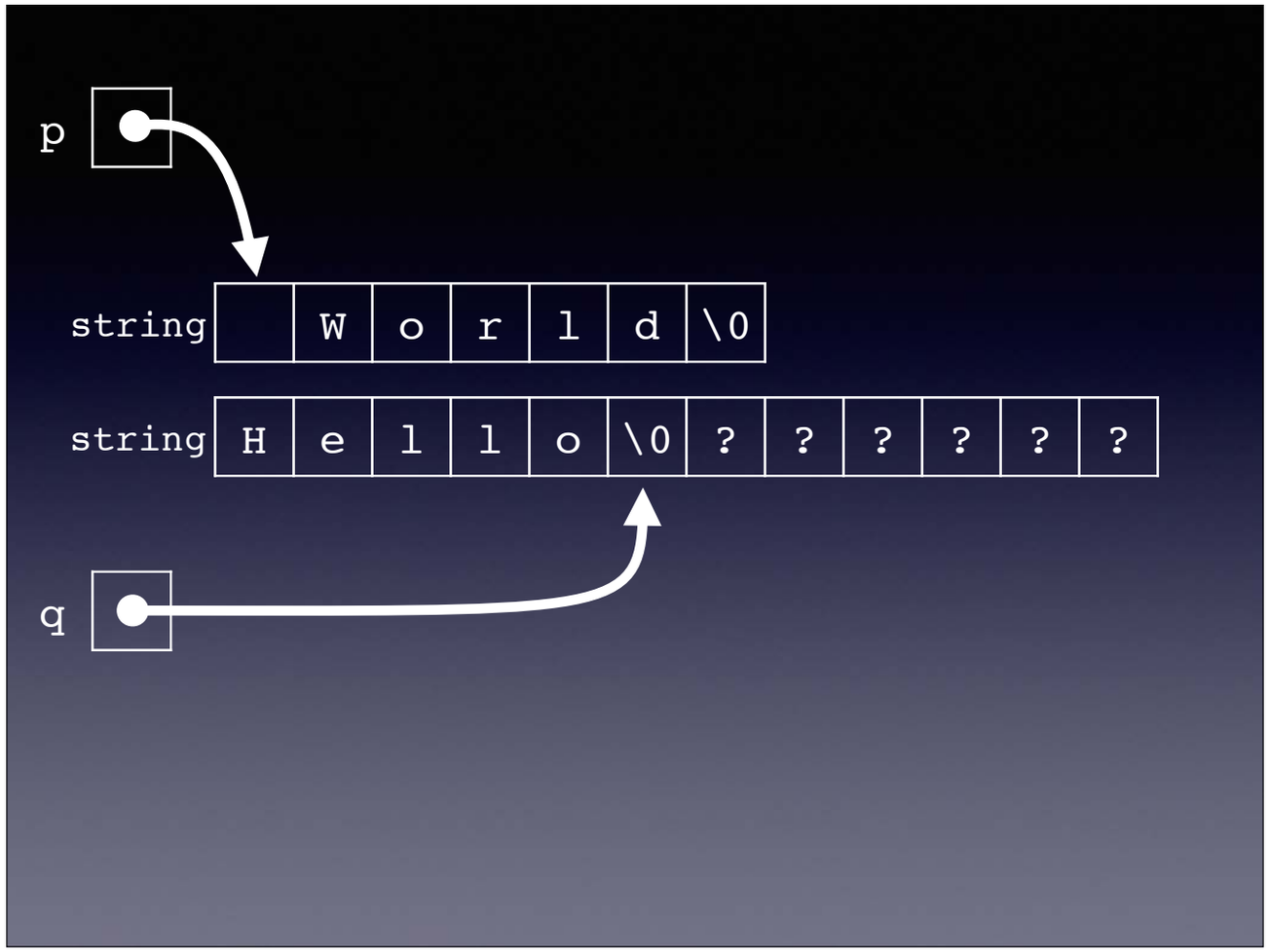
- Again, can copy from anywhere in the string
- Or to anywhere in the destination string
- If we manipulate the pointers
- If we pointer to the null terminator, we can concatenate two strings together

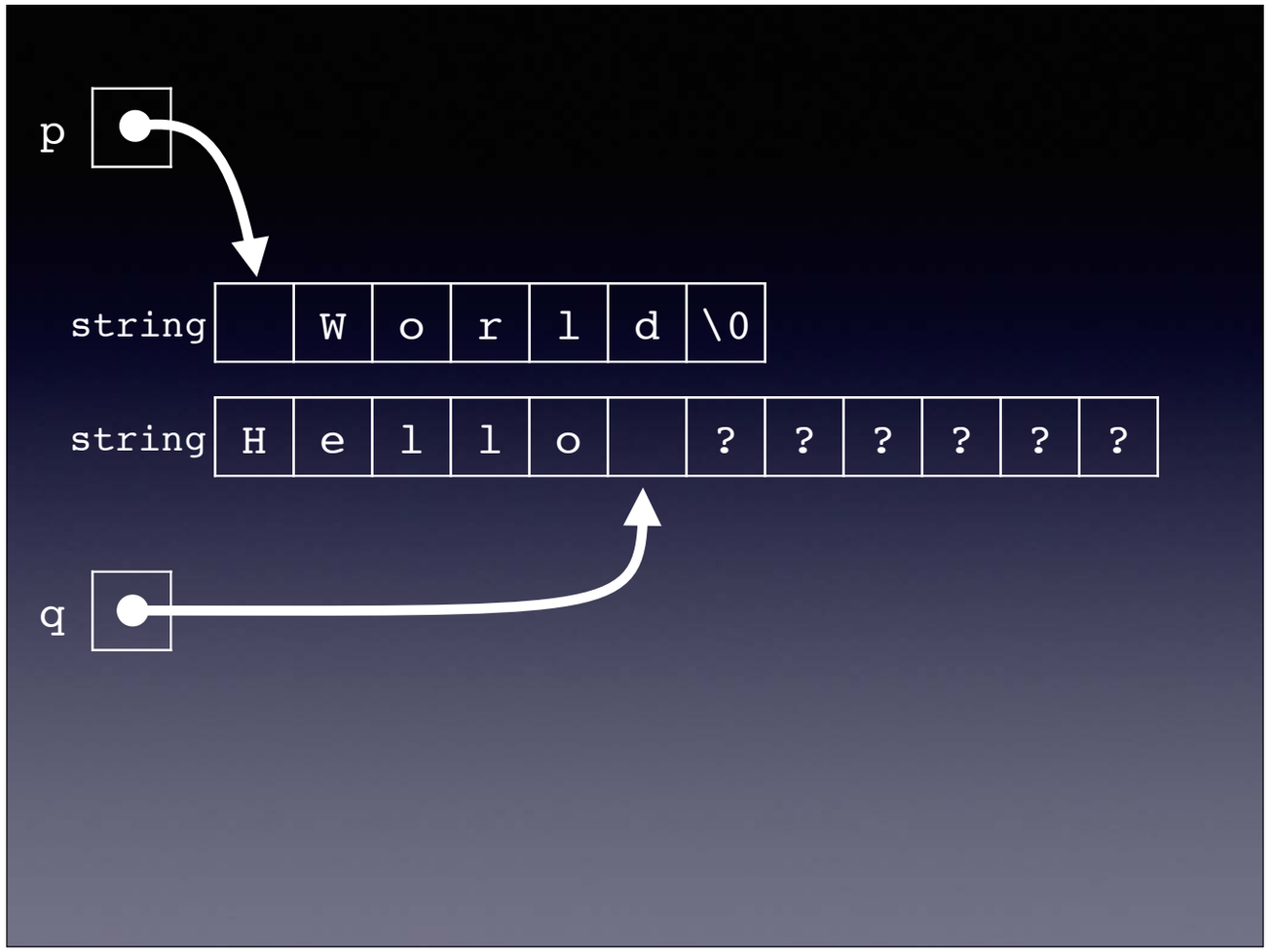


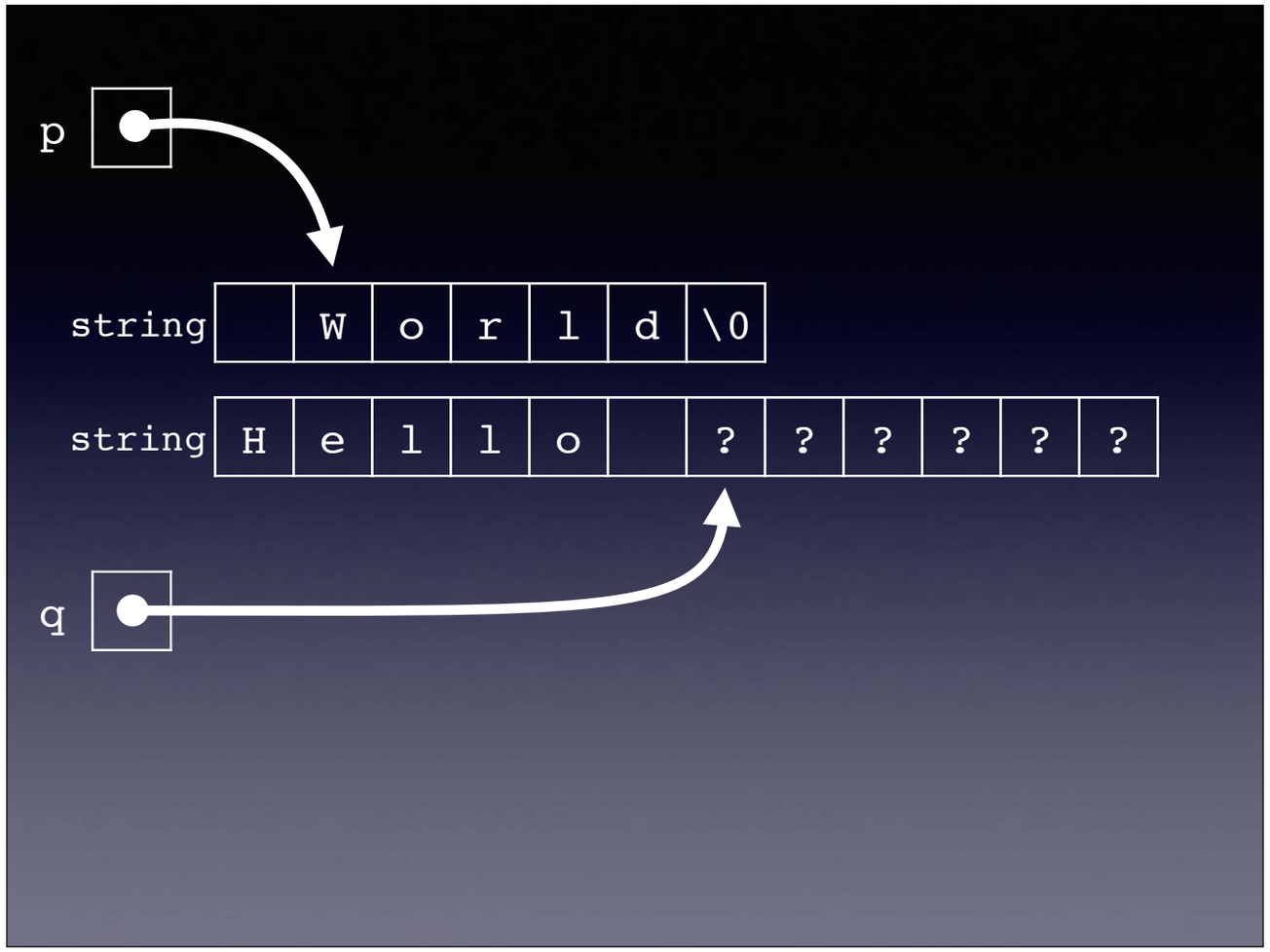

string
W
o
r
l
d
\0

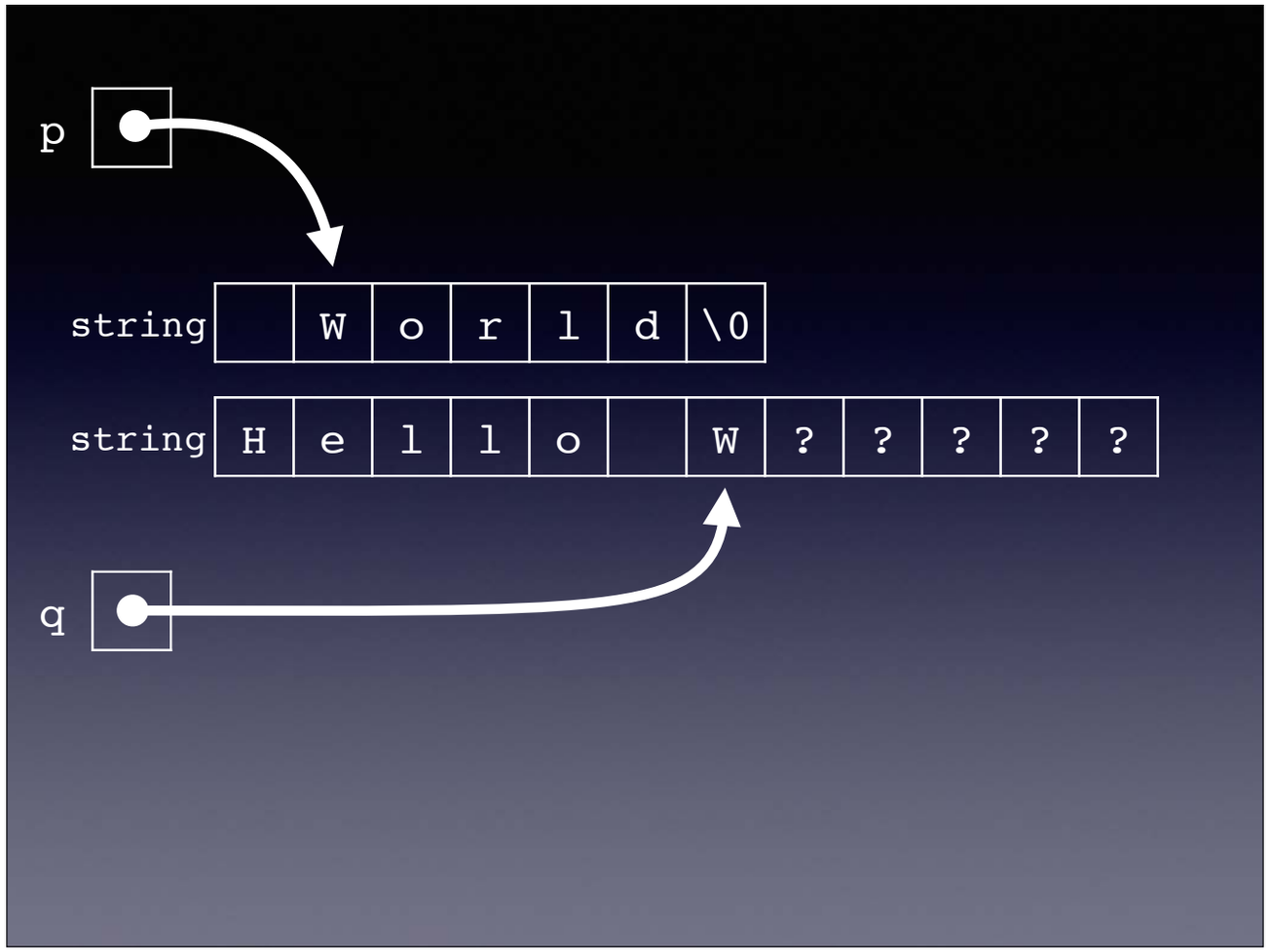

string
W
o
r
l
d
\0
string
H
e
l
l
o
\0
?
?
?
?
?
?

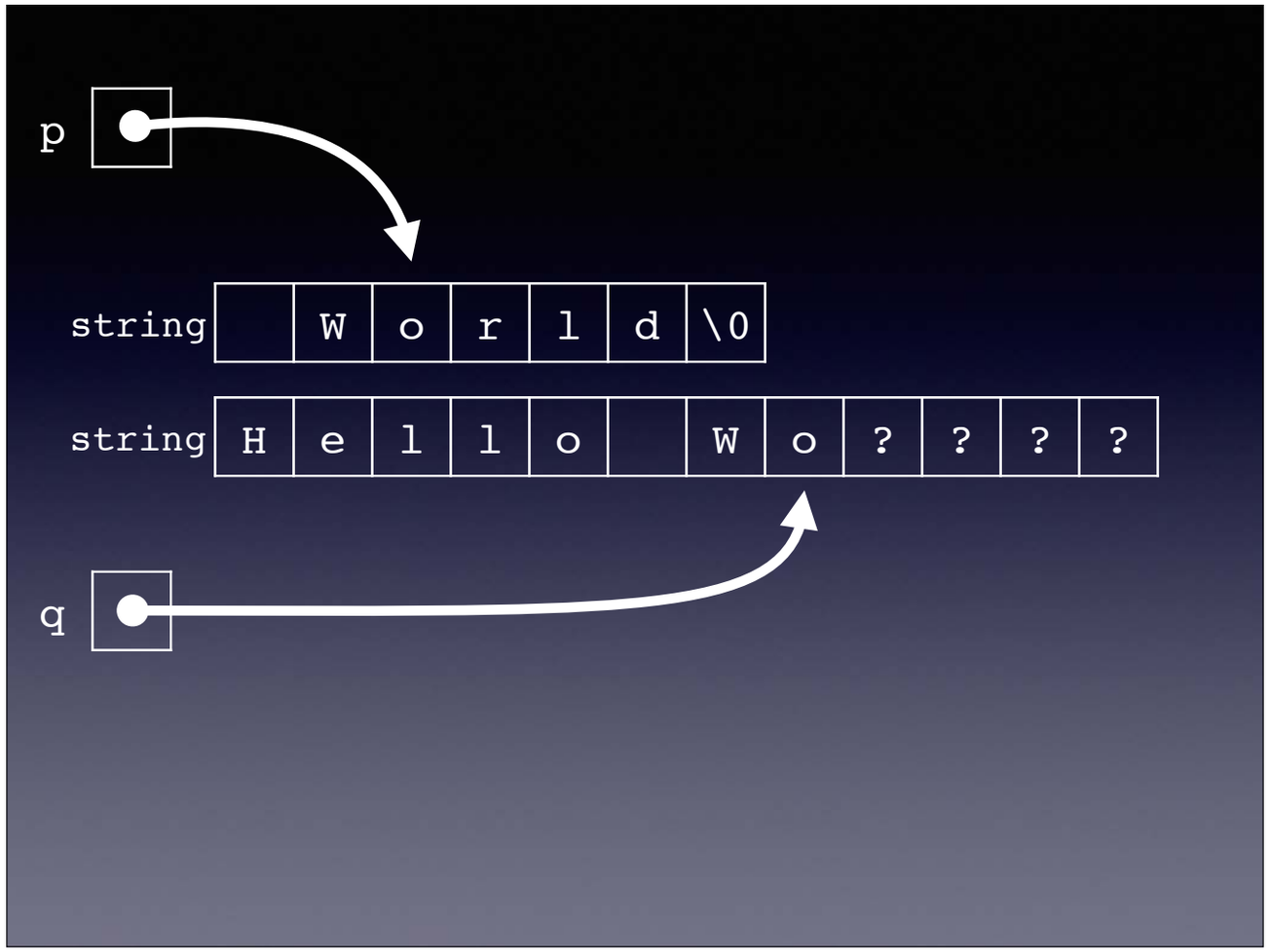

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
\0
?
?
?
?
?
?

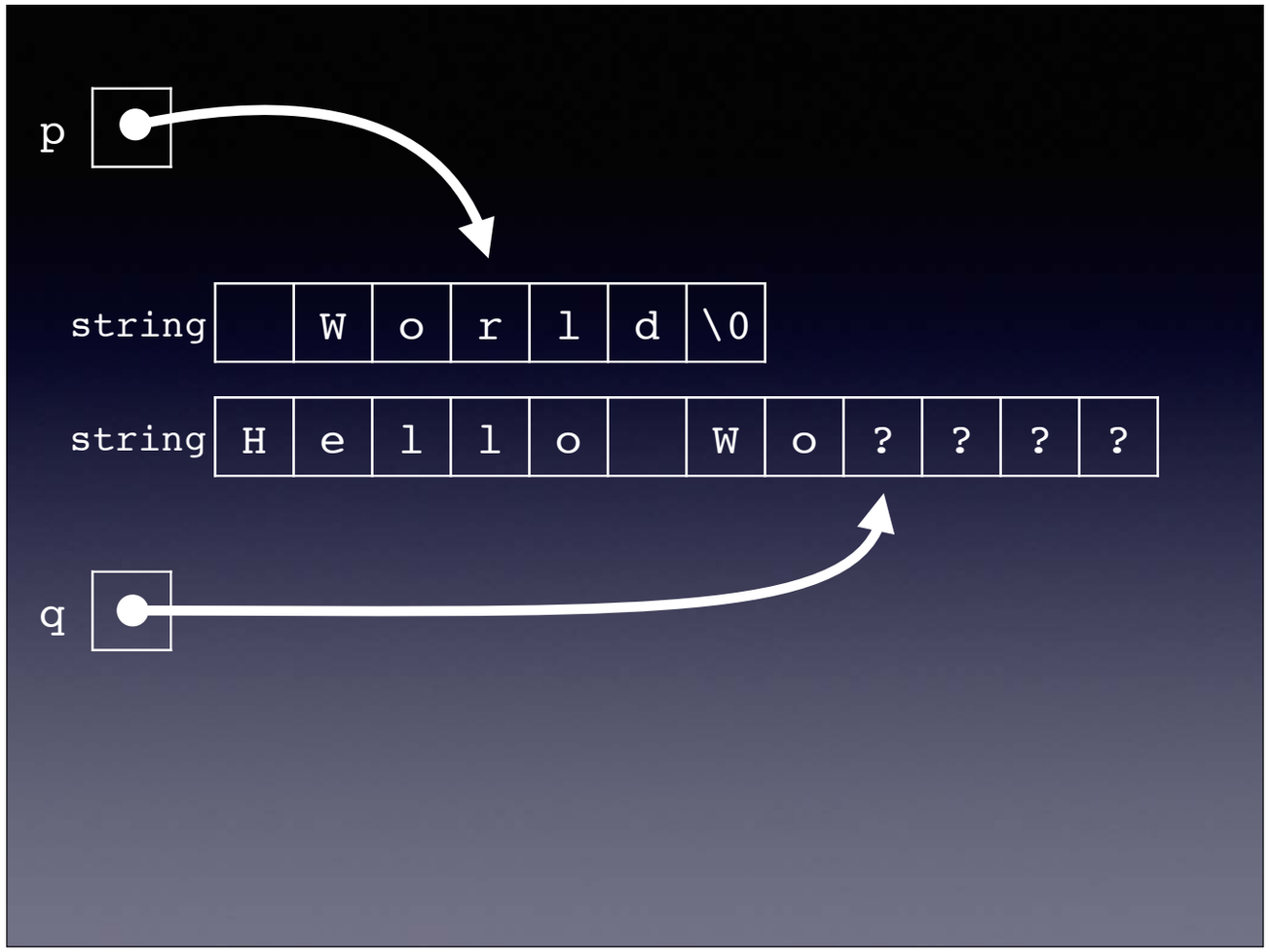

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
\0
?
?
?
?
?
?
q

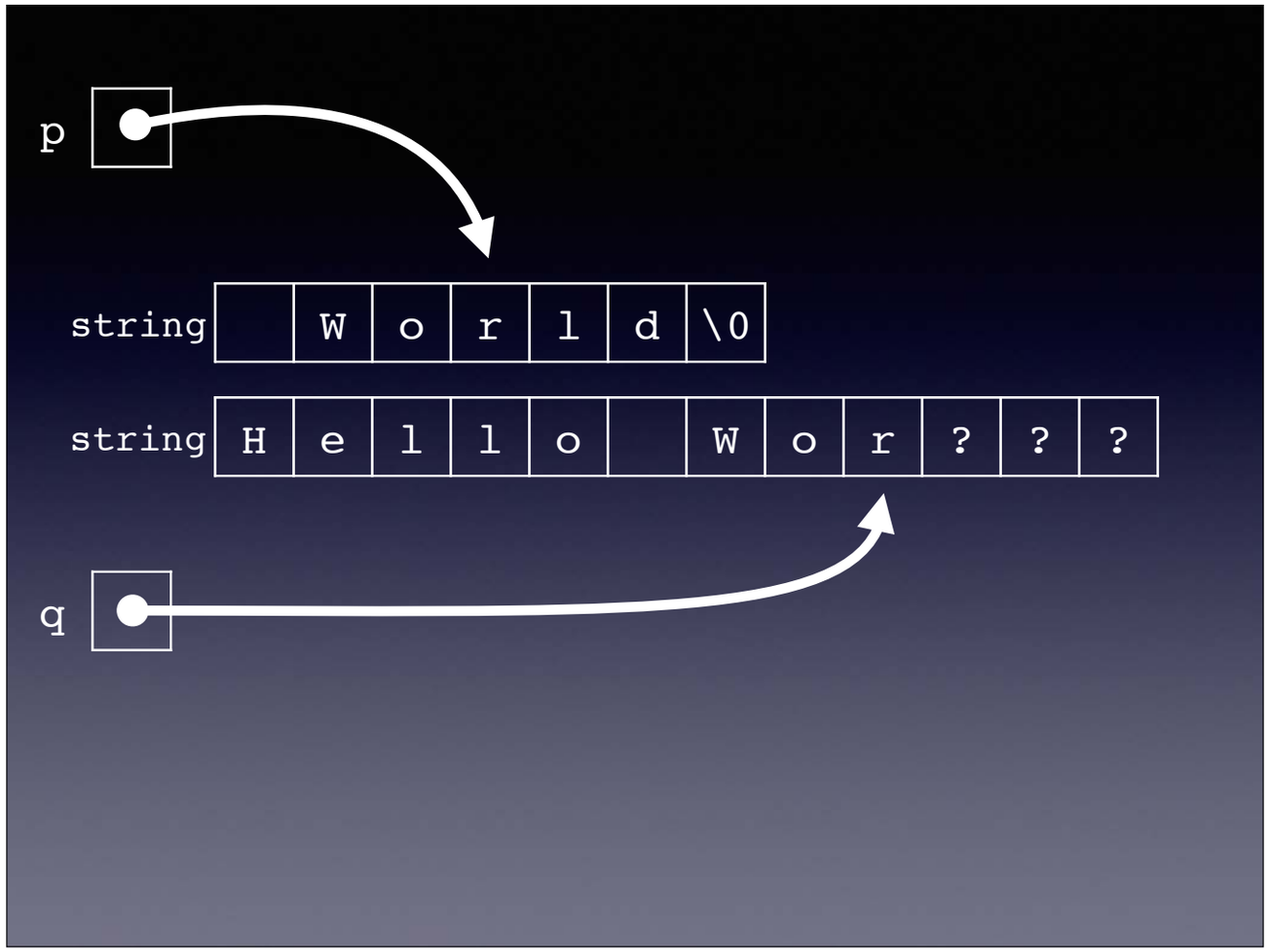

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
?
?
?
?
?
?
q

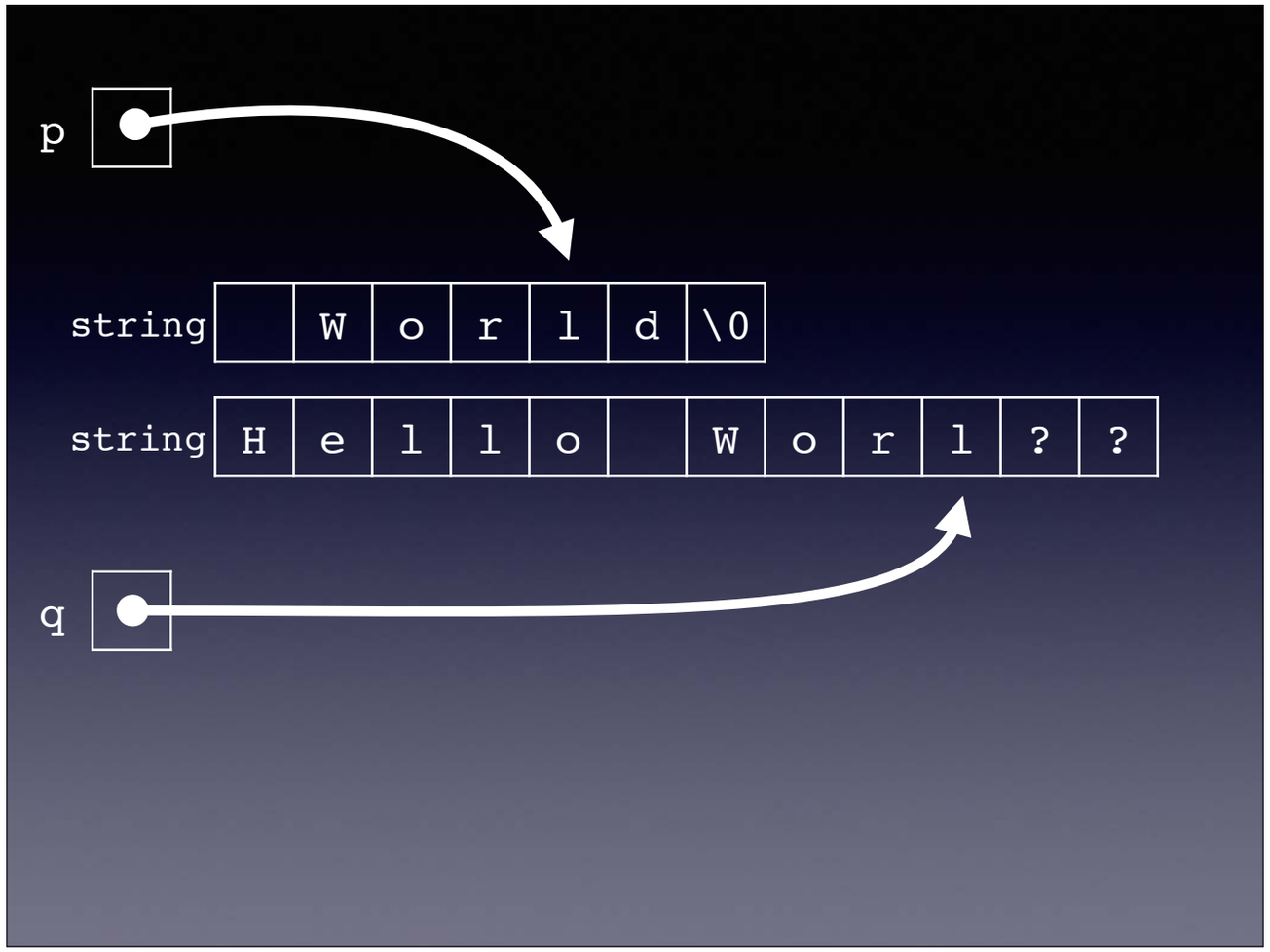

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
?
?
?
?
?
?
q

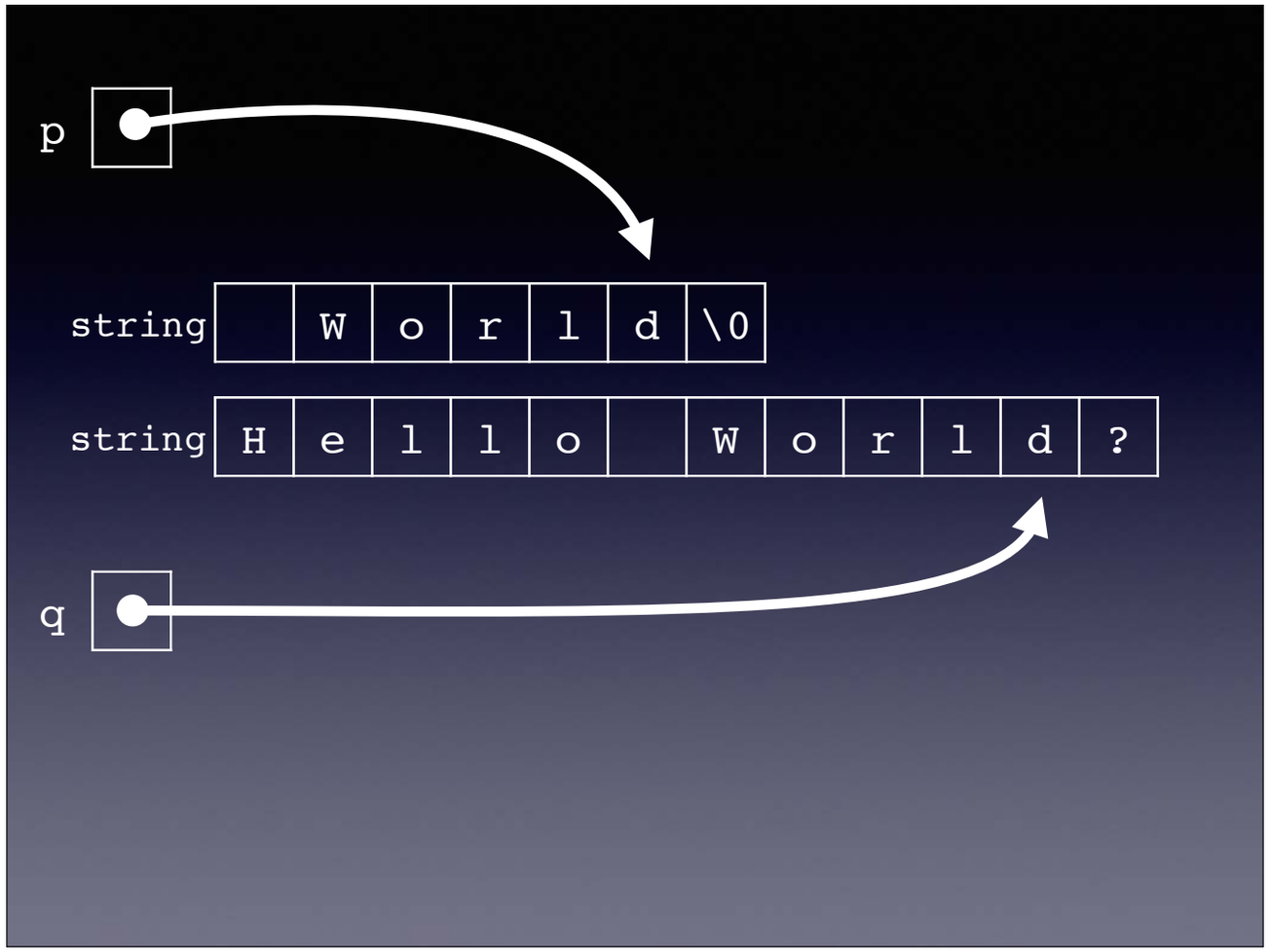

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
?
?
?
?
?
q

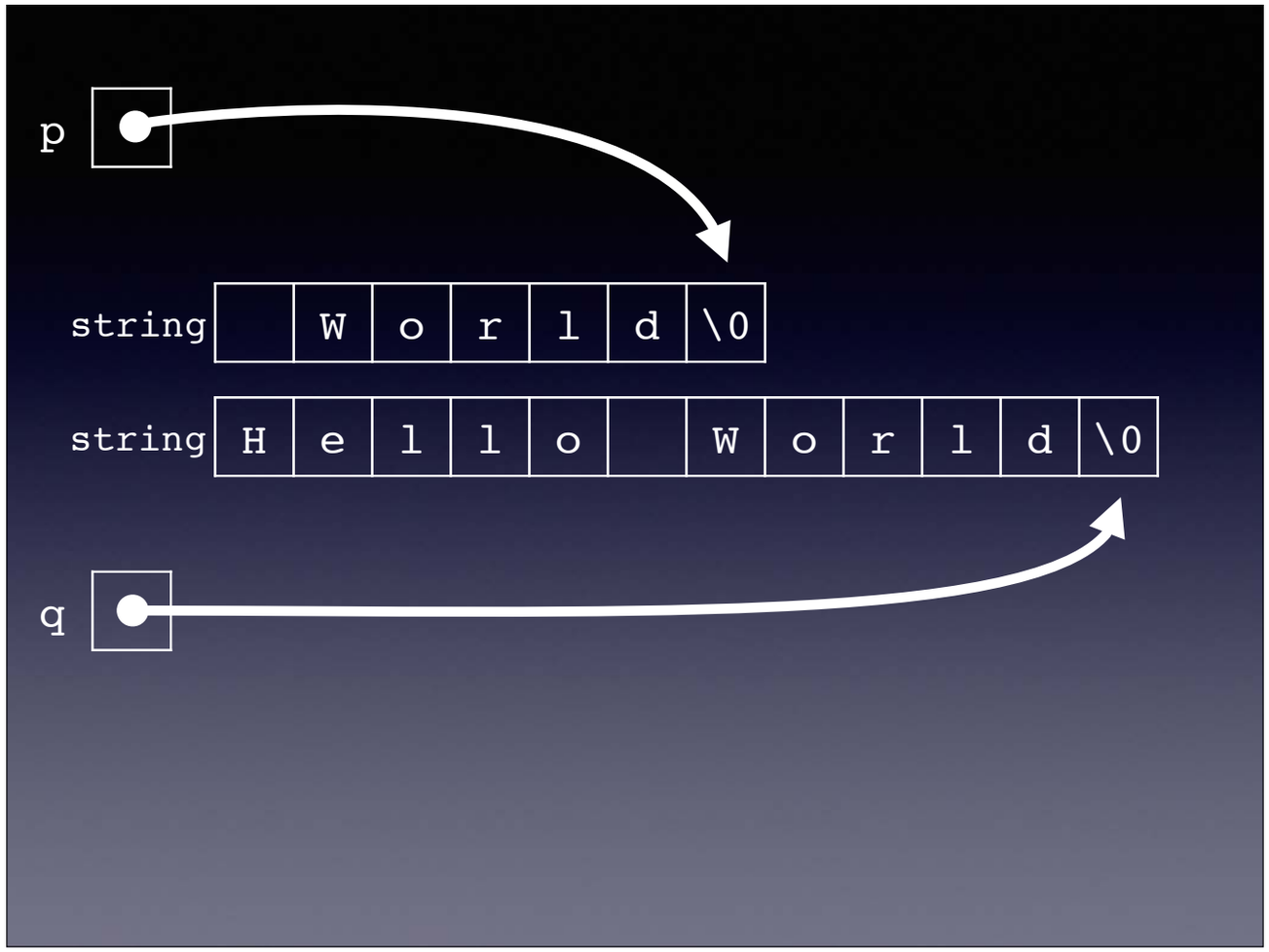

p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
o
?
?
?
?
q


p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
o
?
?
?
?
q


p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
o
r
?
?
?
q


p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
o
r
l
?
?
q


p
string
W
o
r
l
d
\0
string
H
e
l
l
o
W
o
r
l
d
?
q


p
string
W o r l d \0
string
H e l l o W o r l d \0
q

String Concatenation

- How do we find the null?
- Could use `strlen()` and add it onto the pointer
- But we could just advance the pointer in our function

```
while(*p != 0)
    p++;
```

Function	Effect
`strlen(char *s)`	Returns the length of string `s`.
`strcpy(char *s1, char *s2)`	Copy the string `s2` into `s1`. Returns `s1`.
`strcat(char *s1, char *s2)`	Concatenates `s2` onto `s1`. Doesn't check for enough room.
`strcmp(char *s1, char *s2)`	Compares `s1` to `s2`.
`stpcpy(char *s1, char *s2)`	As `strcpy`, but returns pointer to null character
`sprintf(char *s, char *format, …)`	As `printf`, but output written into string `s`

Standard C library routines you can use...
Don't let the strings overlap...
stpcpy isn't ANSI standard iirc

`strcmp`

- Comapres two strings
  `int strcmp(char *s1, char *s2);`
- Returns `0` if the two strings are equal
- Returns `<0` if `s1` comes *before* `s2` lexicographically
- Returns `>0` if `s1` comes *after* `s2` lexicographically