Arrays, Pointers and 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

Array

- Array is a simple data structure
- Holds a series of values of the same type
- C doesn't allow mixed arrays — you'd have to create your own if you want them
- Values accessed by a numeric index
- Values can be read or modified just like a variable

```
int main(int argc, char *argv[])
{
    int a[10];
    int i;


    printf(“Please enter 10 values\n”);
    for(i = 0; i < 10; i++)
    {
        scanf(“%d”, &a[i]);
    }

    printf(“You entered\n”);
    for(i = 0; i < 10; i++)
    {
         printf(“%d: %d\n”, i, a[i]);
    }
}
```

Pointers and Arrays

- Can also get a pointer to a value in an array
```
int *pa = &a[0];
```
- Can use this like any other pointer
```
int x = *pa; /* will get the value in a[0] */
```
- What's interesting is if we look at the addresses of all the values in the array

DEMO some of the pointer and array interaction

Show addresses of all values

```
i is 0, *pa is  0, pa points to address 0x7fff5fbff9f0
i is 1, *pa is 10, pa points to address 0x7fff5fbff9f4
i is 2, *pa is 20, pa points to address 0x7fff5fbff9f8
i is 3, *pa is 30, pa points to address 0x7fff5fbff9fc
i is 4, *pa is 40, pa points to address 0x7fff5fbffa00
i is 5, *pa is 50, pa points to address 0x7fff5fbffa04
i is 6, *pa is 60, pa points to address 0x7fff5fbffa08
i is 7, *pa is 70, pa points to address 0x7fff5fbffa0c
i is 8, *pa is 80, pa points to address 0x7fff5fbffa10
i is 9, *pa is 90, pa points to address 0x7fff5fbffa14
```

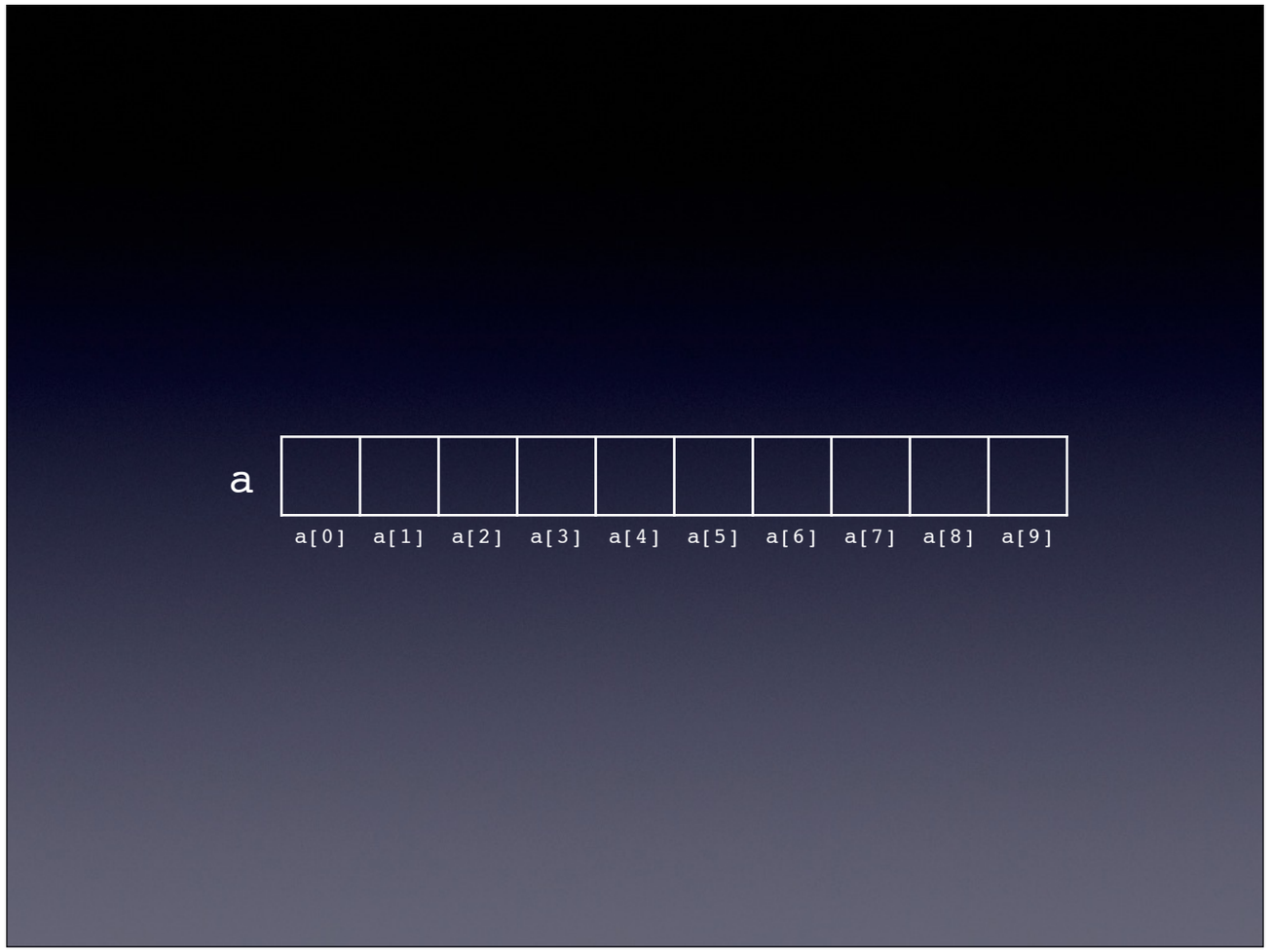
Actually address numbers aren’t interesting (will change every time you run the program)

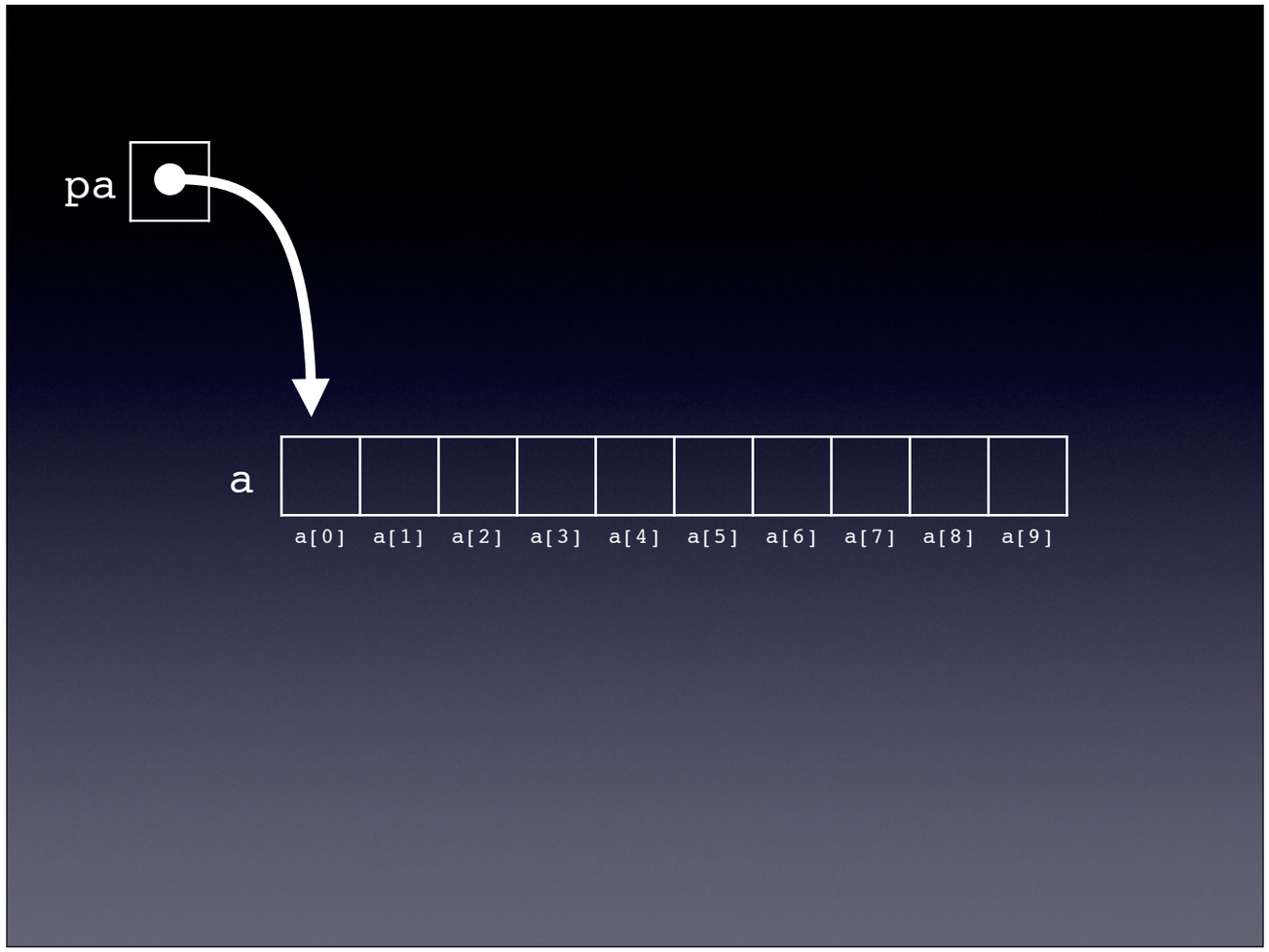
What is interesting is that the address increases by the same amount each time (4 in this case)

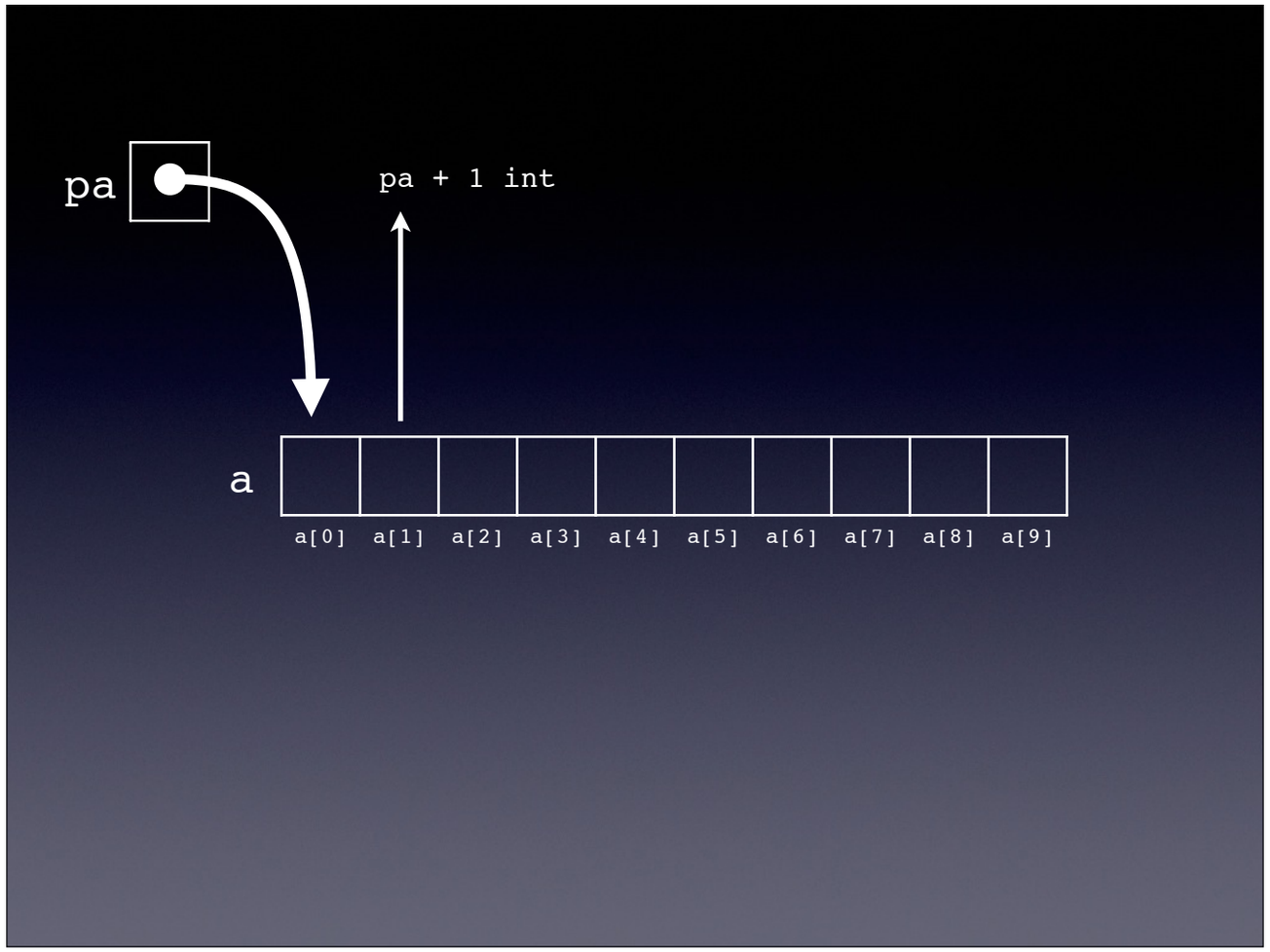
Arrays in Memory

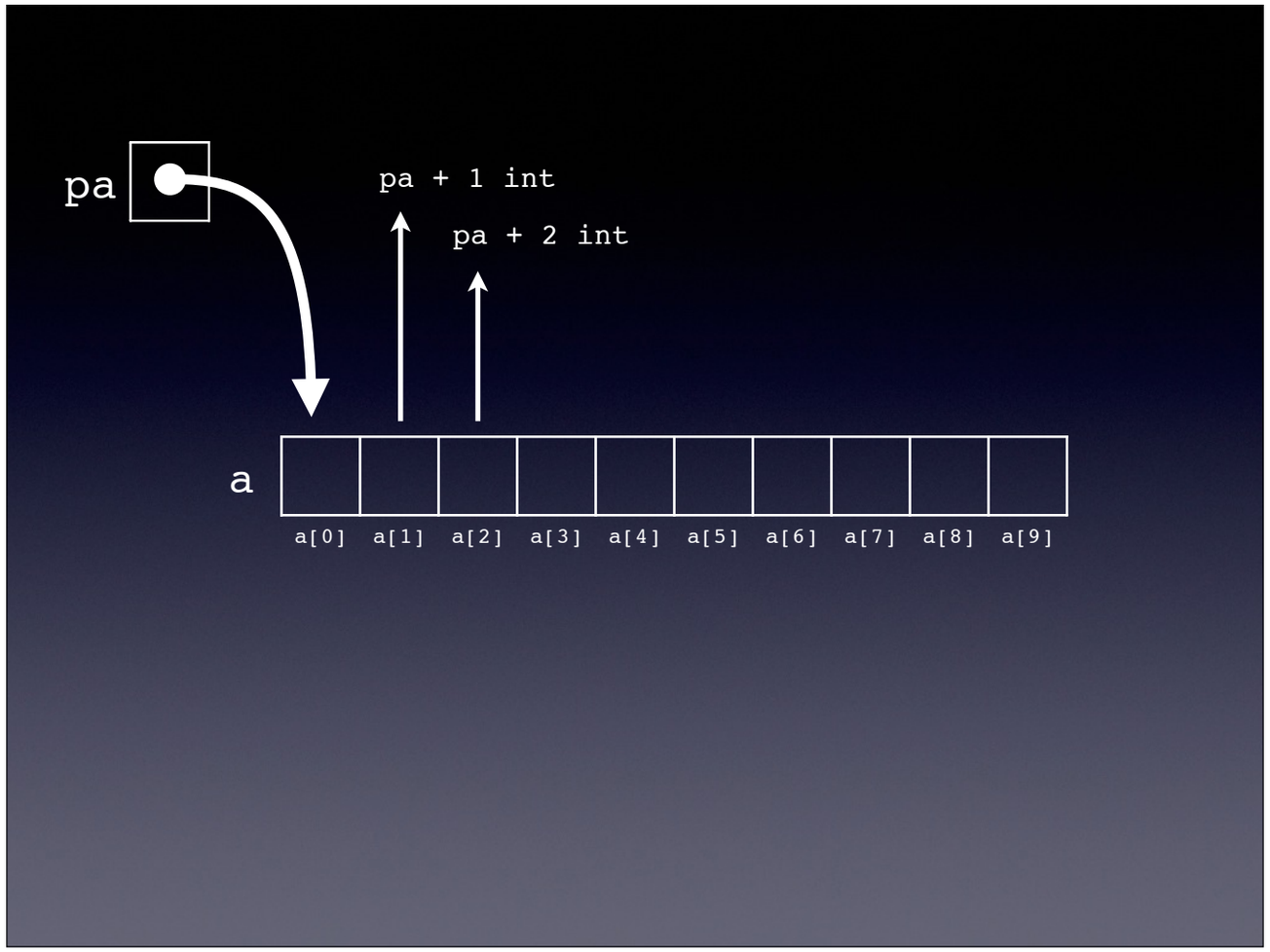
- Arrays are allocated in memory as a contiguous block
- Start with the first element (takes up 4 bytes)
- Then the next element (another 4 bytes)
- And so on

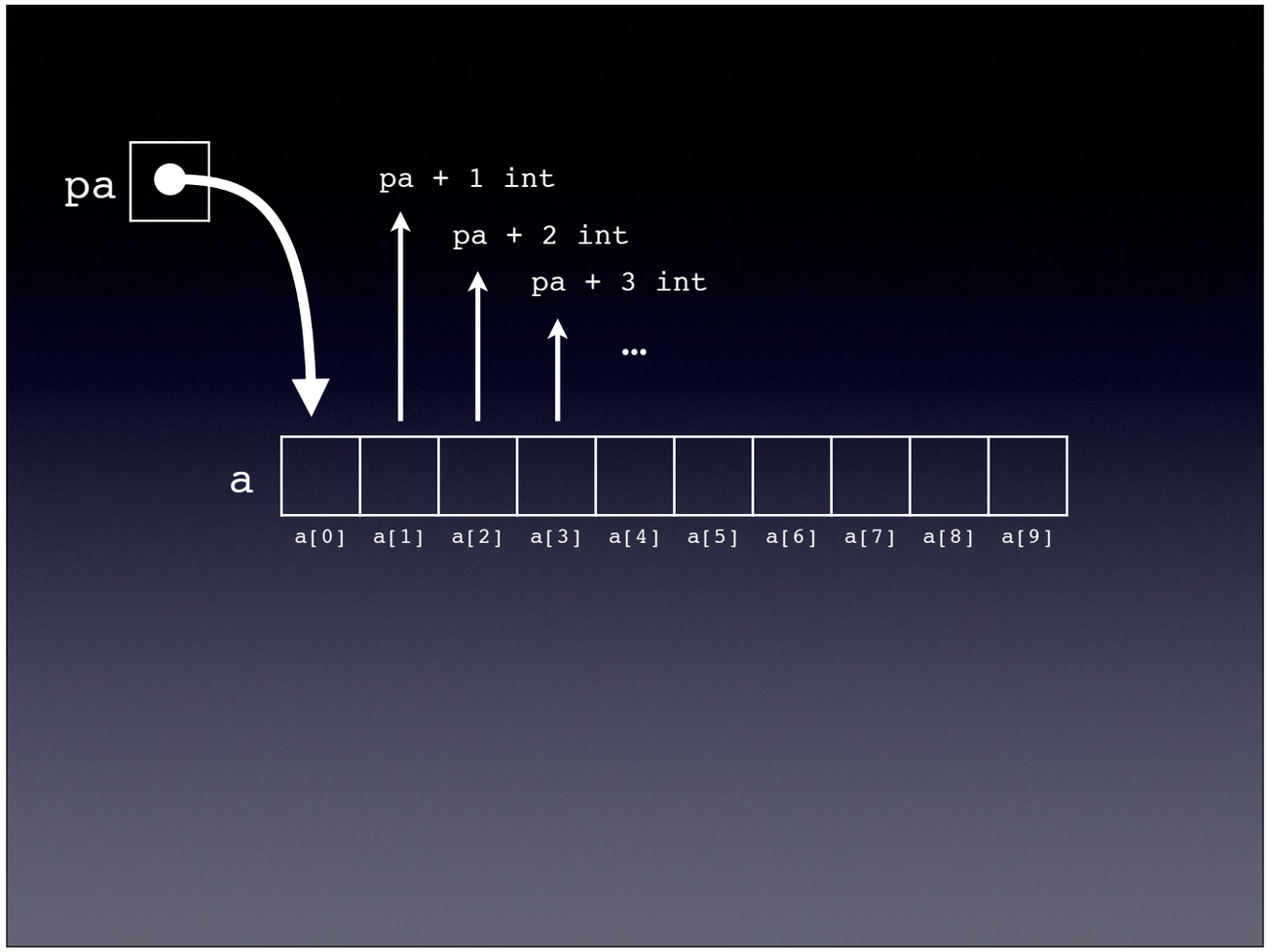
Above slide assumes we are talking about ints in C (on linux at least) an int represents 32-bits or 4 bytes.


a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]


pa
a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]


pa
pa + 1 int
a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]


pa
pa + 1 int
pa + 2 int
a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]


pa
pa + 1 int
pa + 2 int
pa + 3 int
...
a
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]

Arrays and Pointers

- If we have a pointer to a value in an array
- We can find the address of the next value by adding the size of one thing to it
- Or the next but one by adding the size of two things
- And so on…

If Amy lives in room 42 and you know Sarah is two doors down from Amy, you can find Amy's address by adding two to get room 44 -- normally. Computer memory is very regulated

Pointer Arithmetic

- Don't need to know the size of the thing
- In C, adding one to a pointer means move one object along
- So if it is a pointer to an `int`, C will advance the pointer to the next `int` along
- And so on…
- Can only add and subtract values from pointers (and value must be integer)

This can *definitely* catch you out!

Pointers into Arrays

- This means that if we have a pointer to the base of the array
- Then you can access any array value from it
- So to access the 3rd value, either use `a[3]` or `*(pa+3)` (note the brackets)
- Internally, C converts all `a[3]` into the `*(pa+3)` form as it compiles

Demo this

Arrays and Pointers

- Can also write:
  `int *pa = &a[0];`
- As:
  `int *pa = a;`
- In C, the name of an array is a synonym for the address of the first element
- But it is not a pointer, so can't be modified

```
int a[10];
int *pa;


pa = a; /* Valid, pa is a pointer */
pa++; /* Valid, moves pa to point to the next thing */

a = pa; /* Invalid, a cannot be modified */
a++; /* Again, invalid */
```

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