CS265
Advanced Programming Techniques

C Pointers and Strings

C is a low-level programming language

- C is close to the OS and its resources
- Pointers exemplify this
- Pointers allow the C program to manipulate the memory of the computer

Pointer Variables

- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes***, with each byte capable of storing eight bits of information:

0	**1**	**0**	**1**	**0**	**0**	**1**	**1**

1 byte = 8 bits

Pointer Variables

- Each byte has an address
- If there are *n* bytes in memory, we can think of addresses as numbers that range from 0 to *n* – 1:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
n-1	01000011

Each byte has its own address!

Main Memory

- The CPU registers stores memory addresses, which is how the processor accesses data from the RAM.

memory address

9278	…
9279	…
9280	
9281	00010101
9282	
9283	
9284	
9285	
	…

Value at memory address

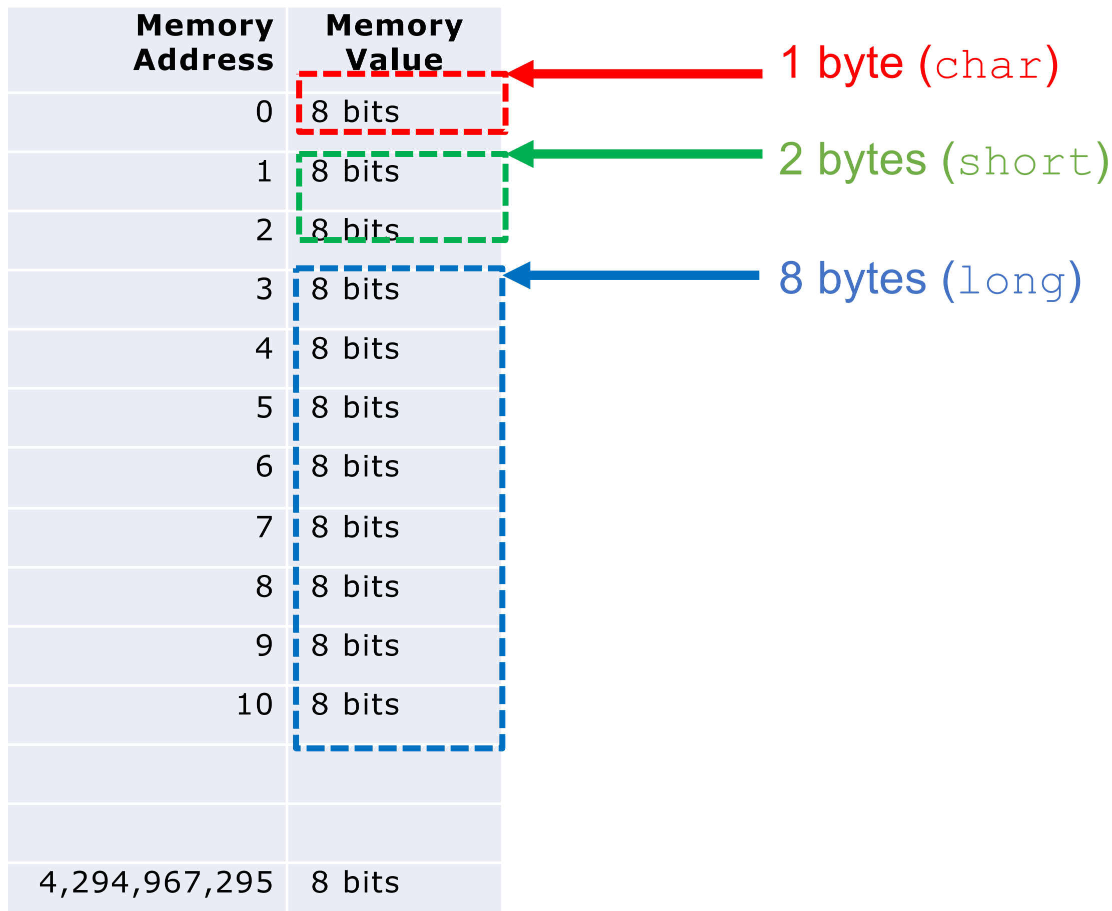
Each memory location stores

1 byte = 8 bits
= 2**8 different values
= e.g. values 0 to 255

Each byte has can have a value 0-255

What if the data type is larger than a byte?

- It is the data type of the variable that tells us how many bytes of memory we need



- The address of the first byte is said to be the address of the variable

How many bytes are there total?

- How many memory locations (bytes) are there?

Memory Address (32 bits)	Memory Value (8 bits)
0	8 bits
1	8 bits
2	8 bits
3	8 bits
????	8 bits

max value ?

How many bytes are there total?

- How many memory locations (bytes) are there?
- It depends!!!
- In a 32-bit architecture every CPU register has 32 bits (4 bytes) which can reference up to 2^{32} different locations

Memory Address (32 bits)	Memory Value (8 bits)
0	8 bits
1	8 bits
2	8 bits
3	8 bits
4,294,967,295	8 bits

max value

I have 32 bits to reference a memory location

2^{32} = 4,294,967,296 = 4GB

32-bit architecture

- Most computers made in the 1990s and early 2000s were 32-bit machines
- 32-bit computers can address a max of 4GB of memory
- Each CPU register in a 32-bit architectures is 4 bytes

4 bytes x 8 bits per byte = 32 bits

64-bit architecture

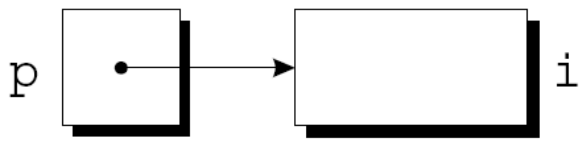
- Each CPU register in a 64-bit architecture is 8 bytes

8 bytes x 8 bits per byte = 64 bits

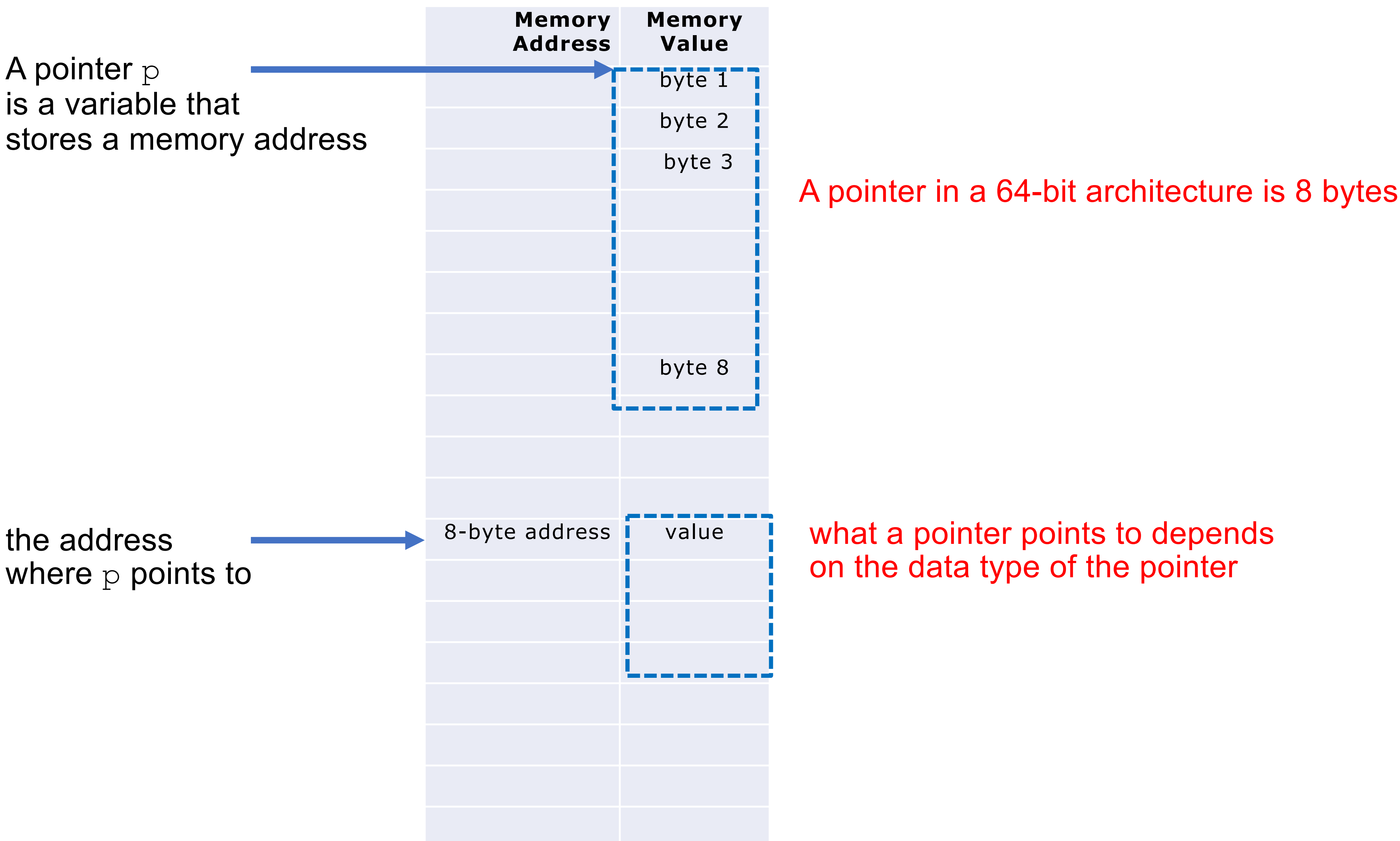
- A 64-bit register can theoretically reference 18,446,744,073,709,551,616 bytes or 17,179,869,184 GB (16 exabytes) of memory
- To put this in perspective in 2014 the size of the entire internet was 1M exabytes
- That’s several million more than an average computer would need
- If a computer has 8 GB of RAM, it better have a 64-bit processor. Otherwise, at least 4 GB of the memory will be inaccessible by the CPU.

Pointer Variables

- Addresses can be stored in special pointer variables
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.

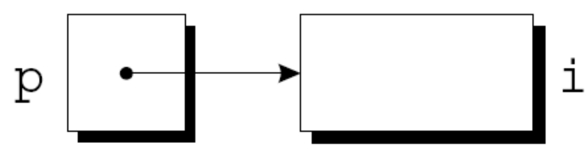


Pointers in a 64-bit architecture are 64 bits = 8 bytes long



Address and Redirection Operators

- C provides a pair of operators that designed specifically for pointers
- To find the address of a variable, we use the `&` (address) operator.
- To gain access to the object that a pointer points to, we use the `*` (***indirection***) operator.

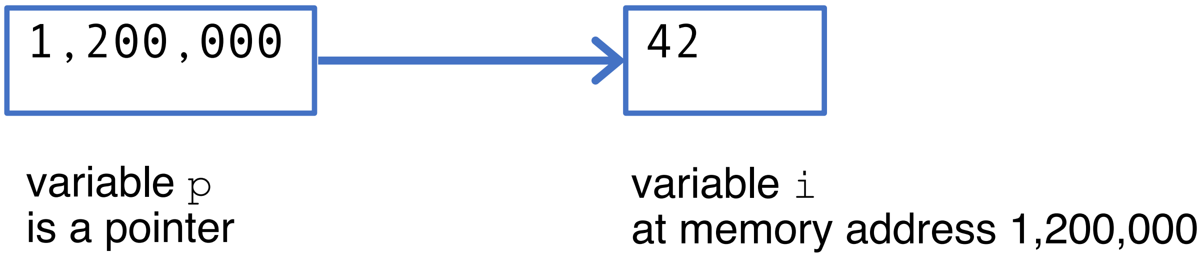


`*p` **unary operator `*` is the value of what the pointer `p` points to**

`&i` **unary operator `&` is the address of variable `v`**

Pointers - Example

- Pointers in C are variables that store the address of another variable



p =

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

```
p=1,200,000

i=
```

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

```
p=1,200,000
```

`i=42`

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

```
p=1,200,000

i=42

*p=
```

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

`p=1,200,000`

`i=42`

`*p=42` unary operator `*` is what the pointer `p` points to

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

`p=1,200,000`

`i=42`

`*p=42` unary operator `*` is what the pointer `p` points to

`&i=`

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

`p=1,200,000`

`i=42`

`*p=42` unary operator `*` is what the pointer `p` points to

`&i=1,200,000` unary operator `&` is the address of variable `i`

Pointers - Example

- Pointers in C are variables that store the address of another variable

variable `p`
is a pointer

variable `i`
at memory address 1,200,000

`p= 1,200,000`

`i=42`

`*p=42` unary operator `*` is what the pointer `p` points to

`&i= 1,200,000` unary operator `&` is the address of variable `i`

Is this true or not?

`if p==&i, then *p=i`

Pointers – how to declare

Variables

`int *p;`	a pointer to a memory location for an integer
`char *p;`	a pointer to a memory location for a char
`void *p;`	a generic pointer
	i.e., a pointer to any location
	i.e., a pointer to an undefined data type

Note

`int *p;`	as a declaration it is a mnemonic for pointer
	it says that `*p` (where p points to) is an `int`
`*p`	anywhere else `*p` it is the value of where `p` points to

Declaring Pointer Variables

```
int *p;
```

- `p` is a pointer variable capable of pointing to ***objects*** of type `int`
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable

The address operator &

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

- It's crucial to initialize `p` before we use it.
- One way to initialize a pointer variable is to assign it the address of a variable:

```
int i, *p;
p = &i;
```

or

```
int i;
int *p = &i;
```

The indirection operator *

- Once a pointer variable points to an object, we can use the `*` (indirection) operator to access what's stored in the object.

- If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

- Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

```
j = *&i;   /* same as j = i; */
```

As long as `p` points to `i`, `*p` is an alias for `i`

The pointer must be initialized first

- Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;
printf("%p", *p);    /*** WRONG ***/
```

- Assigning a value to `*p` is particularly dangerous:

```
int *p;
*p = 1;    /*** WRONG ***/
```

- Example of correct pointer assignment

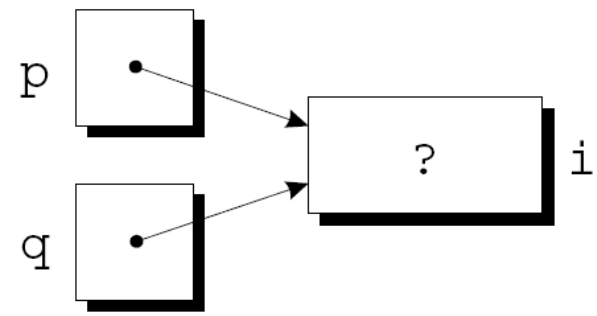
```
int i, *p, *q;
p = &i;    /*** Correct ***/
```

- As well as

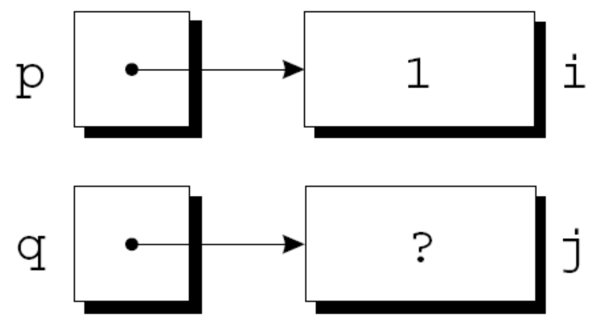
```
q = p;     /** CORRECT ***/
```

Do not confuse `q=p` with `*q=*p`

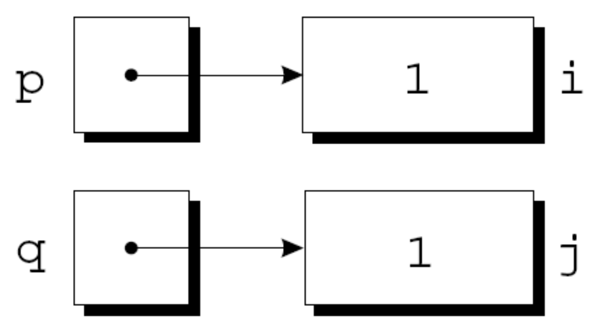
```
q = p;
```




```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Using pointers in expressions

if `p` is a pointer to `x`, then `*p` can appear where `x` can appear

`x` is the same as `*p`

`x = x + 10` is the same as `*p = *p + 10`

`x++` is the same as `(*p)++`

Pointer arithmetic

The value of pointers are memory addresses, thus numbers, so we can do arithmetic. There are four arithmetic operators that can be used in pointers

++

--

+

-

- Move the pointer to the next object (move one byte, if `char *p,` two bytes if `short *p`)

```
p = p + 1;
```

- Same

```
p++;
```

- Advance the pointer by 4 “objects”

```
p = p + 4;
```

Functions – Call by value

swap

```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Before calling `swap(x,y)` the values are `x=1 y=2`

After calling `swap(x,y)` the values are `x=1, y=2`

Functions – call by reference

swap

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Before calling `swap(&x,&y)` the values are `x=1 y=2`

After calling `swap(&x,&y)` the values are `x=2 y=1`

Is python pass by reference or pass by value?

Pointers and Arrays

- In C there is a strong relationship between pointers and arrays
- Any operation that involves array subscripting can use pointers
- The declaration `int a[10]` defines and array

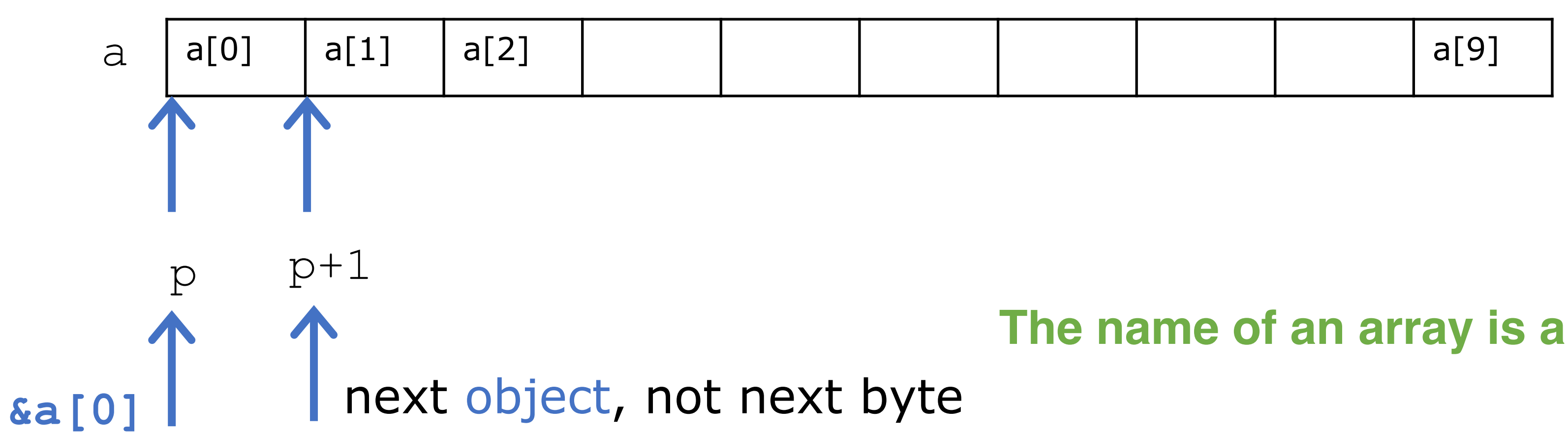
a	a[0]	a[1]	a[2]							a[9]

- If we declare a pointer that points to the beginning of the array

```
int *p;

p = &a[0];
```

- Then `a[i]` is the same as `*(p+i)` and `&a[i]` is the same as `p+i`



The name of an array is a pointer!

Strings

- Strings exemplify the relationship between arrays and pointers
- Strings can be defined as character arrays

```
char date[10];

char date[] = "June 14";
```

- Strings can be defined as pointers

```
char *date;

char *date = "June 14";
```

Thanks to the close relationship between arrays and pointers, either version can be used as a string

Character Arrays vs Character Pointers

```
char date[10];
```

```
char date[] = "June 14";
```

- allocates space for the array
- characters can be modified
- `date` is the name of an array - it cannot be made to point to other strings

```
char *date;
```

- does not allocate space for what the pointer points to
- cannot be used unless first it is assigned to something (i.e., string variable)
- i.e., `date[0] = ‘a’ /*** WRONG ***/`

```
char *date = “June 14”;
```

- allocates space for the what the pointer points to
- characters cannot be modified (because the string is a literal)
- `date` is a pointer that can point to other strings

How to write a string to `stdout`

Two options

- Using `printf` will print a string without a newline

```
char str[15];
printf(“%s\n”, str);
```

- Using `puts` will print a string with a newline

```
char str[15];
puts(str);
```

How to read a string from `stdin`

Two options

- Using `scanf` will read a string skipping whitespace then reading characters and stopping when whitespace is encountered

`str` is a pointer, so we don't put `&` in front

```
char str[15];
scanf("%s", str);
```

- Using `fgets(str, n, stdin)` will read `n` characters of input without skipping whitespace from `stdin` into the variable `str`

```
char str[15];
fgets(str, 14, stdin);
```

Example

need 1 extra char for the '\0'

```
char sentence[SENTENCE_LENGTH+1];

printf("Enter a sentence:\n");
```

If the user enters the line

```
    To C, or not to C: that is the question.
```

- `scanf("%s", sentence);` will store the string `"To"` in `sentence`
- `fgets(sentence, SENTENCE_LENGTH, stdin);` will store the string `" To C, or not to C: that is the question."` in `sentence`.

Both `scanf` and `fgets` may store characters past the end of the array, causing undefined behavior.

C String Library

```
#include <string.h>
```

`strcpy(s1,s2)`	Copies string s2 into string s1, returns s1
`strncpy(s1,s2,n)`	Copies n characters from s2 into s1
`strlen(s1)`	Returns the length of s1, not include the '\0' character
`strcat(s1, s2)`	Appends s2 at the end of s1, returns s1
`strncat(s1,s2,n)`	Appends s2 to the end of s1 up to n characters long
`strcmp(s1,s2)`	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2
`strncmp(s1,s2,n)`	Compares at most the first n bytes of *s1* and *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 s2 in s1
`strtok(s1, delimiter)`	Breaks string *s1* into a series of tokens separated by *delimiter*

Example using `strtok()`

```
#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = ”time-honored-tradition";
    char *token;

    // Returns first token
    token = strtok(str, "-");

    // Keep printing tokens while one of the
    // delimiters present in str[].
    while (token != NULL) {
        printf("%s\n", token);
        token = strtok(NULL, "-");
    }

    return 0;
}
```

two calls to `strtok`
two different arguments

C Code to Read/Write to a file

```
FILE *in_file  = fopen("name_of_file", "r"); // read only
FILE *out_file = fopen("name_of_file", "w"); // write only

// test for files not existing.
if (in_file == NULL || out_file == NULL) {
    printf("Error! Could not open file\n");
    exit(-1); // must include stdlib.h
   }

// write to file vs write to screen
fprintf(out_file, "this is a test %d\n", integer); // write to file

fprintf(stdout, "this is a test %d\n", integer); // write to screen
printf(         "this is a test %d\n", integer); // write to screen

// read from file/keyboard. remember the ampersands!
fscanf(in_file, "%d %d", &int_var_1, &int_var_2);
fscanf(stdin, "%d %d", &int_var_1, &int_var_2);
scanf(        "%d %d", &int_var_1, &int_var_2);
```

C Code to Read One Line From a File

use `fgets` command

```
char line[100];

while ( fgets( line, 100, stdin ) != NULL )
  {
    printf("The line is: %s\n", line);
  }
```

Use the `getline` command

```
FILE *fp = fopen( "someTextFile" ) ;
char *buff = NULL ;
size_t len = 0;

while( getline( &buff, &len, fp ) != -1 )
{
    /* overwrite newline */
    buff[ strlen(buff)-1 ] = '\0' ;
    printf( "%zu chars: %s\n", len, buff ) ;
}
```

Lessons

- Lesson 1: Learn C to become a power programmer
- Lesson 2: C / C++ are the defacto systems programming languages

Resources

- These notes
- *C Programming: A modern Approach* by K.N. King, 2008
- Chapters 11, 13