

Accessing elements in an array
in C/C++ be like



Bring laptops tomorrow

- We will have an in-class exercise!

TODO

- Assignment 1 due September 24

Overview for today

- More arrays
- Memory layout

Array to pointer decay

- We can treat arrays as pointers in C
- In fact, any kind of arithmetic on an array causes it to decay into a pointer
- Note: array subscript syntax works on pointers

Pointer arithmetic

- Treat the pointer like a number and add to get the index
- `int a[5]; // 0x0c`
- `int *p = a; // p == 0x0c`
- `p = p + 1; // p == 0x0d 0x10`
- **C automatically scales arithmetic by size of the type being pointed to (int is 4 bytes)**

Equivalence

- Note:
- `int a[5];`
- `&a[b] == a + b`

More C syntactic sugar

- `a[b]` is syntactic sugar for `*(a + b)`
- Note also: like integer arithmetic, pointer addition is commutative: `*(a + b) == *(b + a)`

So we have...

- $\&a[b] == a + b$
- Because: $\&a[b] == \&*(a + b) == a + b$
- Note: $\&\&\&*x == x$
- (A “ $\&*$ ” cancels itself out.)

More C syntactic sugar

- $a[b]$ is syntactic sugar for $*(a + b)$
- Note also: like integer arithmetic, pointer addition is commutative: $*(a + b) == *(b + a)$
- Therefore, $a[b] == b[a]$

Arrays Exercise

```
int a[5];
```

```
int *p;
```

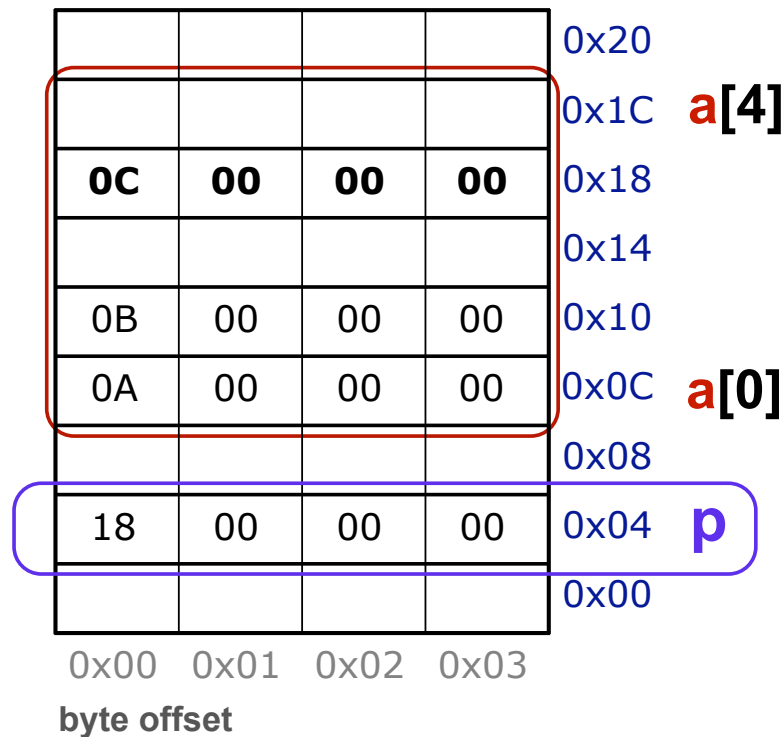
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

```
*p = 1[a] + 1;
```



C strings

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

```
int main(void) {  
    char s[] = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x00};  
    printf("%s\n", s);  
    return 0;  
}
```

Revisiting strings

- C strings are just char arrays
- C strings are null-terminated
- How would we get a string's length?
- In C: `strlen()`

strlen(3)

Library Functions Manual

strlen(3)**NAME**

strlen - calculate the length of a string

LIBRARYStandard C library (libc, -lc)**SYNOPSIS**

#include <string.h>

size_t strlen(const char *s);**DESCRIPTION**The **strlen()** function calculates the length of the string pointed to by s, excluding the terminating null byte ('\0').**RETURN VALUE**The **strlen()** function returns the number of bytes in the string pointed to by s.**ATTRIBUTES**For an explanation of the terms used in this section, see **attrib-
Manual page strlen(3) line 1** (press h for help or q to quit)

Version 1

```
unsigned int string_length(char *str) {  
    unsigned int len = 0;  
    while (str[len] != '\0') {  
        len++;  
    }  
    return len;  
}
```

Version 2

```
unsigned int string_length(char *str) {  
    unsigned int len = 0;  
    while (*(str + len) != '\0') {  
        len++;  
    }  
    return len;  
}
```

Version 3

```
unsigned int string_length(char *str) {  
    unsigned int len = 0;  
    char *ptr = str;  
    while (*ptr != '\0') {  
        ptr++;  
        len++;  
    }  
    return len;  
}
```

Version 4

```
unsigned int string_length(char *str) {  
    char *ptr = str;  
    while (*ptr != '\0') {  
        ptr++;  
    }  
    return ptr - str;  
}
```

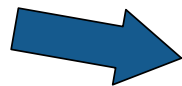
Which is faster?

- Let's count arithmetic operations per loop
- Version 1: 1 addition, 1 increment
- Version 2: 1 addition, 1 increment
- Version 3: 2 increments
- Version 4: 1 increment

Memory layout

- A.k.a. *memory map*
- A program's memory is carved into *segments*
- Let's take a look at what those segments are laid out

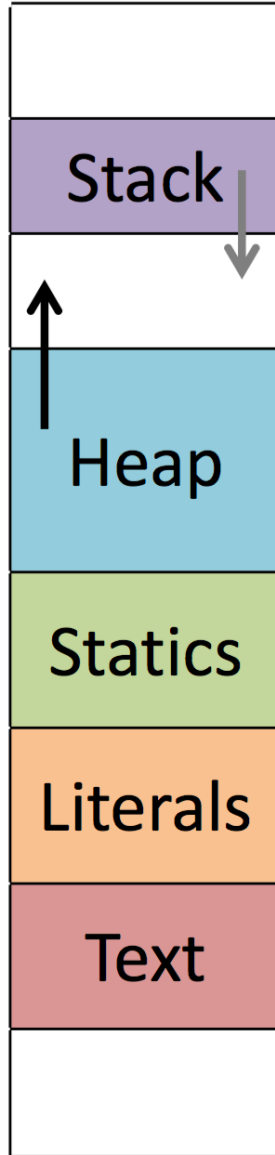
"Top" of stack
(lowest address)



2^N-1

Addresses

0



Stack

Function context, local variables
(**compiler-managed**)

Heap

Persistent data structures
(**programmer-managed**)

Statics

Global variables

Literals

String literals

Text

Program code (instructions)

Breakdown

- Stack – automatic allocation
 - E.g., declaring a variable in a function body
- Heap – dynamic allocation
 - E.g., `new Blob()`, `malloc()`
- Statics – static allocation
 - E.g., global variables
- Literals – read-only static allocation
 - E.g., string constants and other array constants
- Text
 - Program machine code

Stack (automatic allocation)

```
int next = 1;
```

```
void next_ticket(void) {  
    int n = next++;  
  
    printf("Ticket %d\n", n);  
}
```

Statics (static allocation)

```
int next = 1;
```

```
void next_ticket(void) {  
    int n = next++;  
  
    printf("Ticket %d\n", n);  
}
```

Literals (read-only static allocation)

```
int next = 1;
```

```
void next_ticket(void) {  
    int n = next++;  
  
    printf("Ticket %d\n", n);  
}
```

Text

```
int next = 1;
```

```
void next_ticket(void) {  
    int n = next++;  
    printf("Ticket %d\n", n);  
}
```

Heap?

- We will see heap / dynamic allocation soon!

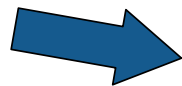
Changing size

- Statics/literals/text do not change size with program execution.
- Stack/heap can grow and shrink with program execution.
- Stacks grow the “deeper” you get into function calls (the C compiler generates code to do this)
- Heaps grow when you write code to allocate memory

Growing

- Heaps grow upward
- Stacks grow downward!?
 - The “top” of the stack is actually the lowest stack address 🙄
- Why? Out of memory is easy to define: when the heap and stack collide

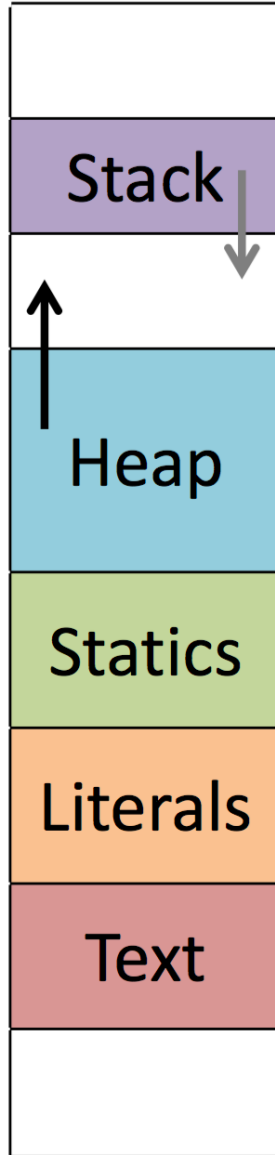
"Top" of stack
(lowest address)



2^N-1

Addresses

0



Stack

Function context, local variables
(**compiler-managed**)

Heap

Persistent data structures
(**programmer-managed**)

Statics

Global variables

Literals

String literals

Text

Program code (instructions)

Shrinking

- Stacks shrink when your functions return
 - The automatically allocated variables are automatically deallocated
 - The C compiler generates code to do this
- Heaps shrink when you write code to free memory

Allocation responsibility

- Statics, literals, stacks: C compiler automates it
- Heaps: Programmer manages it

But why would you want manual?

- Automatic memory is deallocated when the function returns
- What if you want to generate data that can be used after the function returns? 🤔

How do we dynamically allocate?

- Java: `new ClassName()`
- C: `malloc()`

How do we dynamically deallocate?

- Java: garbage collection
- C: `free()`

Function definitions

- `void *malloc(size_t size);`
- `void free(void *ptr);`

Function definitions

- `void *malloc(size_t size);`
- `void free(void *ptr);`
- `void pointer`: a pointer that can point to anything

Void pointers

- So to allocate a single int:

```
int *ptr = malloc(4); // !!!  
free(ptr);
```

- We can store malloc's return value into ptr
- We can pass ptr to free()

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument
- How would we malloc() memory for 1 long?
- `long *ptr = malloc(???)`;

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument
- How would we malloc() memory for 1 long?
- `long *ptr = malloc(sizeof(long));`

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument
- How would we malloc() memory for an array of 3 longs?
- `long *ptr = malloc(???)`;

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument
- How would we malloc() memory for an array of 3 longs?
- `long *ptr = malloc(sizeof(long) * 3);`

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

- `sizeof()` can take a type as an argument
- How would we `malloc()` memory for 1 **int pointer**? (Note: **not** for 1 int!)
- `int **ptr = malloc(???)`;

Data sizes not guaranteed

- We can use `sizeof()` operator:

```
int *ptr = malloc(sizeof(int));  
free(ptr);
```

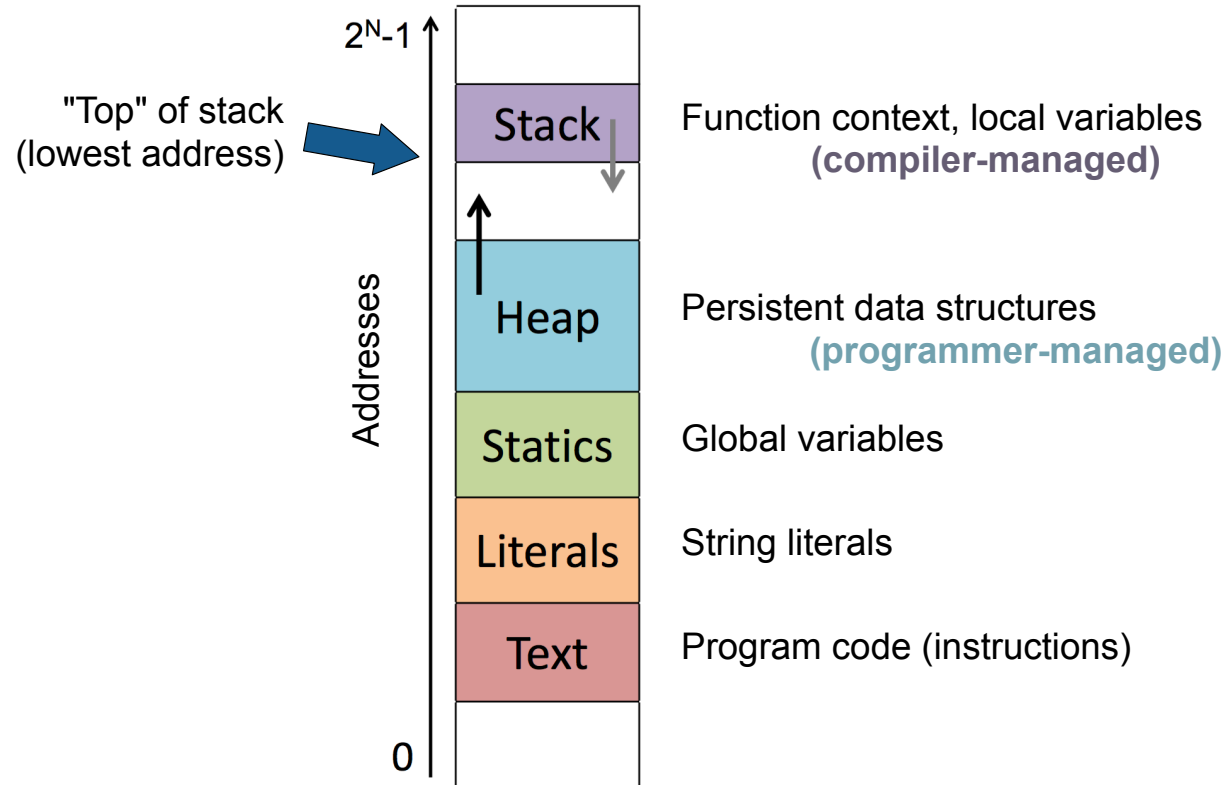
- `sizeof()` can take a type as an argument
- How would we malloc() memory for 1 **int pointer**? (Note: **not** for 1 int!)
- `int **ptr = malloc(sizeof(int *));`

sizeof()

- It's weird because it can take types as arguments
- A special operator that the compiler replaces with the corresponding integer during compilation

pointertest.c

- Let's experimentally confirm the memory layout



What happens when we are bad

- What happens if we forget to `free()` a `malloc()`?
- What happens if we access `free()`d memory?
- What happens if we `free()` `free()`d memory?

Forget to `free()` a `malloc()`?

- *Memory leak*
- If you do this in a loop, your program will use more memory until all memory is exhausted

Access free()d memory?

- *Use-after-free error*
- Undefined behavior
 - Immediate crash
 - Crash later
 - Weird, hard to debug behavior
 - Et cetera

free() freed memory?

- *Double-free error*
- Undefined behavior
 - Immediate crash
 - Crash later
 - Weird, hard to debug behavior
 - Et cetera

TODO

- Assignment 1 due September 24