

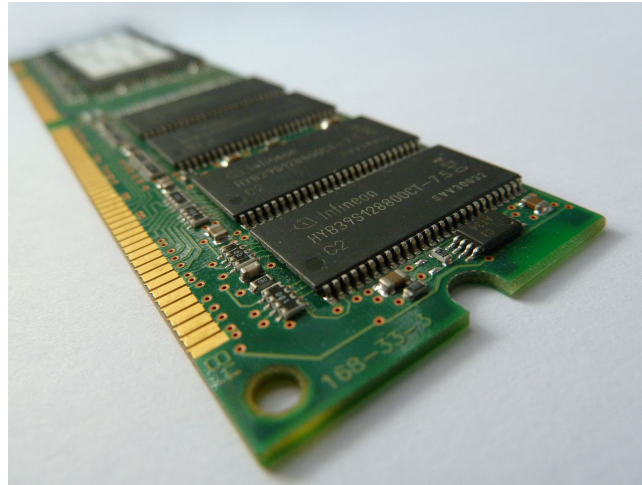
# Pointers and dynamic data structures

Kevin Wang

# Pointers

# Your computer needs memory to run programs

Variables must be loaded and stored from memory in order for things to happen



[https://upload.wikimedia.org/wikipedia/commons/b/b6/Ram\\_chip.jpg](https://upload.wikimedia.org/wikipedia/commons/b/b6/Ram_chip.jpg)

# Every object resides somewhere in memory

- Objects include variables, functions, and other things we haven't defined yet
  - Anything that's a “thing” in your code
- How do we know where an object is in memory?

# This is an address

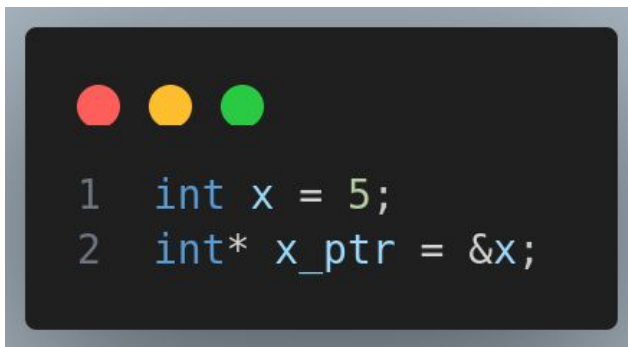
1600 Amphitheatre Pkwy,  
Mountain View, CA 94043,  
United States



screenshot from Google Maps

# Objects in code also have addresses

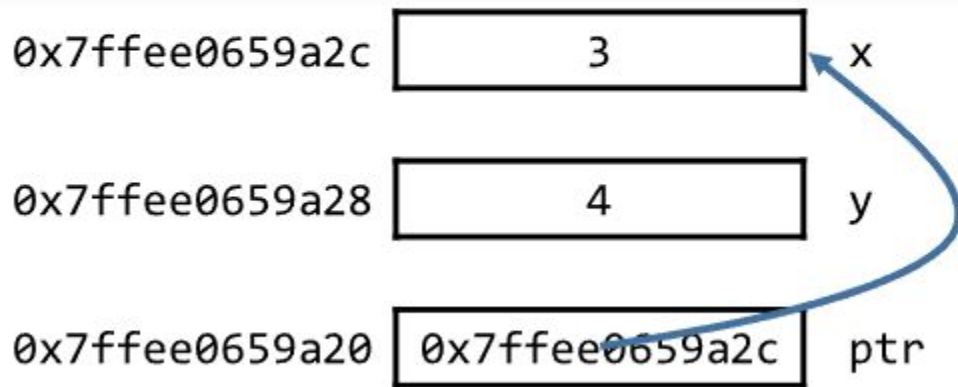
- **Address:** where an object resides in memory
- We can create a special kind of object that stores the address of another object
  - This is called a **pointer**



```
1 int x = 5;
2 int* x_ptr = &x;
```

# Visualization of addresses, variables, and values from EECS 280

We often say that a pointer “points to” an object, and that the `*` operator “follows” the pointer to the object it is pointing at. In keeping with this terminology, a pointer is often pictured as an arrow from the pointer to the object it is pointing at, as shown in Figure 13.




[https://eecs280staff.github.io/notes/03\\_Pointers.html](https://eecs280staff.github.io/notes/03_Pointers.html)

Figure 13 An arrow indicates the object whose address a pointer holds.

# Declaring a pointer

- `<type> * <variable name> = & <other variable you want address of>`
  - You can also point to arrays
  - For placeholder value, use NULL
- Examples:



```
1 int* ptr_to_integer = &some_int;
2 double* ptr_to_double = &some_double;
3 char* string = "hello";
4 int** ptr_to_integer_ptr = &ptr_to_integer;
```



# Dereference a pointer to get the object it is pointing at

- Use the `*` operator
  - Doesn't work for strings! Strings in C are arrays and are more complicated.




```
1  int x = 5;
2  int* ptr_to_x = &x;
3  printf("Value of x is %d\n", *ptr_to_x); // prints 5
4  *ptr_to_x = 10;
5  printf("Value of x is %d\n", *ptr_to_x); // prints 10
```

# Pass by value vs pass by pointer

# Let's run the following program

What value do you think will be printed?



```
1  int f(int x) {  
2      int y = 7;  
3      x = 5;  
4      return y;  
5  }  
6  
7  int main() {  
8      int x = 9;  
9      int y = f(x);  
10     printf("x has value %d, y has value %d\n", x, y);  
11 }
```

# All parameters in C are technically pass by value

- Pass by value: the parameter will be a different object with the same value
- Pointers, however, allow for something informally called pass by pointer
  - Pass by pointer lets you change the original value

# Let's try that again, but with some slight modifications

- What has changed?
- What do you think will print now?



```
1  int f(int* x) {  
2      int y = 7;  
3      *x = 5;  
4      return y;  
5  }  
6  
7  int main() {  
8      int x = 9;  
9      int y = f(&x);  
10     printf("x has value %d, y has value %d\n", x, y);  
11 }
```

# When writing larger, interconnected programs, remember how objects are passed around and copied

- C uses pass by pointer frequently - you often don't want to make copies of large objects
- Primitives (e.g. int, char, etc.) are usually fine to leave as pass by value because they are so small

# Arrays as pointers

# Array variables can become pointers by array decay

What do you think prints here?

Does `(arr + 1)` remind you of an array operation?



```
1  int main() {  
2      int arr[] = {0, 1, 2, 3};  
3      printf("%d", *arr);  
4      printf("%d", *(arr + 1));  
5  }
```



# Array decay: arrays have a pointer value when required

- If you ask the program for the value of a variable that corresponds to an array, the program will tell you that it is a pointer
- Basically, arrays can be used like pointers

# Static vs automatic vs dynamic memory

# Some variables live in static memory

- These variables last for the program's entire lifetime
  - Global variables
  - Static variables
  - Constant variables
- You don't usually work with these in a meaningful way until OOP (object oriented programming)
  - So don't worry about this for now

# Most objects live in local memory (aka stack / automatic memory)

- Local objects only last as long as the scope they are in
  - In this example, x only remains in memory for as long as we are in the function f
  - More efficient not to keep variables alive all the time



```
1  int* f() {  
2      int x = 5;  
3      return &x;  
4  }
```

# What happens with this code?



```
1  int* f() {  
2      int x = 5;  
3      return &x;  
4  }  
5  
6  int main() {  
7      int* x_ptr = f();  
8      printf("value of x is %d\n", *x_ptr);  
9  }
```

# We need dynamic memory (aka heap memory) to keep variables alive across scopes


- Lifetime of a dynamically allocated variable is controlled by the programmer
- This code will reliably work!

```
1  int* f() {  
2      int* x_ptr = malloc(4);  
3      *x_ptr = 5;  
4      return x_ptr;  
5  }  
6  
7  int main() {  
8      int* x_ptr = f();  
9      printf("value of x is %d\n", *x_ptr);  
10     free(x_ptr);  
11 }
```

malloc **and** free

# malloc = memory allocate

- Allocate = to make space for something
- Must specify the right number of bytes
- Use `malloc` to allocate space for dynamic memory




```
1  int main() {  
2      int* x = malloc(4); // int is 4 bytes on my machine  
3      free(x);  
4  }
```



## malloc function returns a pointer

- So the return value of `malloc` should be assigned to something with a pointer type



```
1  int main() {  
2      int* x = malloc(4); // int is 4 bytes on my machine  
3      free(x);  
4  }
```

# Sizes of primitive types in bytes

Data type	Size(bytes)	Range	Format String
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short	2	-32,768 to 32,767	%d
unsigned short	2	0 to 65535	%u
int	2	32,768 to 32,767	%d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to +2147483647	%ld
Unsigned long	4	0 to 4294967295	%lu
float	4	-3.4e-38 to +3.4e-38	%f
double	8	1.7 e-308 to 1.7 e+308	%lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf


Ints can be different sizes, most systems have it as 4 bytes

Pointers can be 4 or 8 bytes depending on the system, usually 8

[https://miro.medium.com/v2/resize:fit:712/1\\*cemNFCrMA3MK27nCuUuG\\_Q.png](https://miro.medium.com/v2/resize:fit:712/1*cemNFCrMA3MK27nCuUuG_Q.png)

# `free` removes an object from dynamic memory

- Always use a `free` for each `malloc`



```
1  int main() {  
2      int* x = malloc(4); // int is 4 bytes on my machine  
3      free(x);  
4  }
```

- Don't use the object after the `free`!

# You can dynamically allocate arrays as well



```
1  int main() {  
2      int* arr = malloc(20); // 5 ints  
3      for (size_t i = 0; i < 5; ++i) {  
4          arr[i] = i + 10;  
5      }  
6      for (size_t i = 0; i < 5; ++i) {  
7          printf("element %lu is %d\n", i, arr[i]);  
8      }  
9      free(arr);  
10 }
```

# Always use `malloc` and `free` together

- A list of problems that can happen from misusing `malloc` and `free`:
  - dangling pointer
  - double free
  - bad free
  - memory leak - `malloc` is used, but not `free`
  - And more...
- These errors are often hard to detect
  - Undefined behavior: when an error might or might not cause problems depending on where your program is in memory

## Further English-language resources

University of Michigan, EECS 280: [Memory Models and Dynamic Memory](#)

More on static vs dynamic allocation of variables and what this looks like behind the scenes.

Some very important rules regarding the design of programs with dynamic memory – you should learn more in your OOP class.

Note that they use C++ and not C! (replace new with malloc, delete with free)

# Stacks and queues

# Data structure: any method of organizing data

- In C / C++, usually represented by a struct / class
- Arrays are the most basic data structure
- There are many fundamental data structures you will learn:
  - Arrays
  - Stacks, queues
  - Linked lists
  - Hash tables
  - Graphs
  - Trees, tries
- For today, we will introduce stacks and queues



# Queue: FIFO array (first in, first out)

Think about standing in line at the grocery store



[https://upload.wikimedia.org/wikipedia/commons/6/62/Covid-19\\_%27Alert\\_Level\\_3%27\\_New\\_World\\_supermarket\\_social\\_distancing\\_queue.jpg](https://upload.wikimedia.org/wikipedia/commons/6/62/Covid-19_%27Alert_Level_3%27_New_World_supermarket_social_distancing_queue.jpg)

# Queues are just FIFO arrays

The first element you put in is the first element that goes out

```
// init. with empty stack
```

--	--	--

```
insert(1)
```

1		
---	--	--

```
insert(2)
```

1	2	
---	---	--

```
pop()
```

4	2	
---	---	--

```
insert(5)
```

2	5	
---	---	--

```
pop()
```

2	5	
---	---	--

# Stack: LIFO array (last in, first out)

Think about eating a stack of pancakes



[https://live.staticflickr.com/3798/33007641255\\_6e01dd22e5\\_b.jpg](https://live.staticflickr.com/3798/33007641255_6e01dd22e5_b.jpg)

# Stacks are just LIFO arrays

The last element you put in is the first element that goes out

```
// init. with empty stack
```

--	--	--

```
insert(1)
```

1		
---	--	--

```
insert(2)
```

1	2	
---	---	--

```
pop()
```

1	2	
---	---	--

```
insert(5)
```

1	5	
---	---	--

```
pop()
```

1	5	
---	---	--

# Assignments for today

# Complete warmup exercises and analyze functions for errors

Go to GitHub  $\Rightarrow$  michigan-musicer  $\Rightarrow$  al.exercise\_6 for code files.