CS 214 / 2021-02-01
==========

An "object" is anything that is stored in memory
	-> this can include code, depending on how you think about it

What do we know about objects generally?
1. location in memory (the address)
2. type of data
3. size (bytes needed to store data)

Generally, this information is tracked by the compiler or the run-time system
-> C does not have "introspection", so we have no way to ask questions about
	objects in general

Objects can be grouped into three categories

global or static objects
	- allocated when process starts, deallocated when process ends
	- cannot create new ones during runtime, or change sizes
	- global variables
	- string literals
	- static local variables

	- compiler always knows where these are
		- no need to track these at run-time; program is written implicitly knowing
		 where globals, etc. are located

	- implicitly initialized to 0 (all zero bits), or NULL

stack objects
	- associated with the "call stack"
		- stores the return address; says where to go when a function exits
	- most languages store local variables and function arguments in the stack
	- the stack as a whole is dynamically sized
	- individual stack frames are usually static
		- the number/size of local variables doesn't change during run-time
		- there are ways to dynamically allocate stack space, but we won't use them

	- compiler knows where these are, relative to the stack pointer
		compiler tracks local variables using offsets from the stack pointer

			e.g. for a local variable, int x, compiler will have table that says
				x is 4 bytes and located in a specific part of the stack frame

	- program does not need to manage these
		- compiler and run-time manage allocation/deallocation of stack frames
		- these are also called "automatic" variables

			i.e., auto int x; // you can write auto explicitly, but it is implied

			by default, local variables are always auto

	- stack objects are not initialized by default
		- unless you explicitly initialize, they will contain unspecified data
			before they are first written to
		- never read from an uninitialized variable
		- C gives you the option of not initializing, because you might want to
		 save time by not initializing a variable that you won't need

heap objects
	- not associated with variables
	- explicitly allocated/deallocated by the program
		- malloc reserves space for an object
		- free marks that space as no longer in use
	- dynamic memory management
		- run-time system keeps track of what parts of memory are in use/available
		- no reliable way for the program to obtain this information

	- in Java, "new" allocates a heap object
		the garbage collector detects unused objects and frees them

	when I call malloc(), I provide a size (the number of bytes I want)
	and I get back a pointer to an object that is at least that big
		we can call this a "heap pointer"

	when I no longer need the object, I pass that pointer to free()
	-> the only pointers I should pass to free() are ones I got from malloc()
		e.g., do not pass free() the address of variables, or pointers to the middle
		of a heap object

---

aside: static variables
	- essentially global variables that are private to a single function

	the first time you call strtok(), you pass it a pointer to a string
	each subsequent time, you pass it NULL, and it remembers the string it was
		working with and where it was in the string

	another use: random number generation
		set a seed, then update the seed each time it is called

	static variables are rarely useful

	// this function returns an increasingly large number each time it is called
	int count()
	{
		static int c = 0; // c is preserved across function calls
		return c++;
	}

----

malloc/free

- why do we need free?
	if we never deallocate memory, we could run out (a memory leak)
	for short-lived programs, this is not much of an issue
		-> but it is good to practice cleaning up after yourself

- why do we need malloc?
	- we might not know how big an object needs to be until run-time
		e.g., an array of data read from a file
		global and local variables are statically sized (we can't make them bigger
		or smaller at run-time)
	- we might want to create data structures like trees/linked lists/graphs
	 that require indirect references

----

aside: sizeof

	sizeof() is an operator that returns the size (in bytes) of a type
		sizeof(int) might be 2 or 4
		sizeof(double) usually 8
		sizeof(int*) might be 4 or 8
	sizeof() is not a function, and does not have any run-time cost:
		the compiler replaces it with an appropriate constant

	if I write something like 100 * sizeof(int), the compiler will replace
	sizeof(int) with the actual size and then do the multiplication
		-> on the iLab, it will be as efficient as just writing 400
		-> but using sizeof() guarantees that we get the correct size

	we can also use sizeof() with variables, but this always gives us the size
	of the variable's object

		int a[4]; // local array of 4 ints
		int *p = malloc(sizeof(int) * 4); // pointer to object that holds 4 ints

		sizeof(a) is equivalent to 4 * sizeof(int), because a in an array variable
		sizeof(p) is equivalent to sizeof(int*), because p is a pointer variable

		sizeof(*p) is equivalent to sizeof(int), because the type of p is int*

----

array variables vs array objects

array variables always have a static (constant) size

	int a[400]; <- dimension must be known at compile-time

	array variables alway refer to global or stack objects

	technically, the type of "a" is int[400]

when I want a dynamically sized array, I have to allocate a heap object

	int arraylen = ...;

	int *p = malloc(sizeof(int) * arraylen);
		// allocates a heap object big enough to store arraylen ints
		// or returns NULL if not enough space is available
		// technically unsafe to use a pointer from malloc if you haven't checked
		// whether it returned NULL

	the type of "p" is int*
		<- note that it is a pointer to an int, not a pointer to an array

in C, array indexing works using pointer arithmetic
	what happens behind the scenes is that we multiply the index by the size of
	the array element type and add to the pointer

p[10] <- adds 10 * sizeof(int) to the address of p to get the address of p[10]

	p[10] means "10th integer after p"
	this is why pointers are typed - so we know how big the elements are

nothing checks whether this address points to anything/the correct thing
	<- no bounds checking!
	<- you can go past the end of an array and never find out
		<- if you are lucky, your program will crash with a segfault

	<- the only way to avoid this is to make sure your algorithms are correct
	<- tools like valgrind or AddressSanitizer can detect some errors

negative array indexes are allowed!
	p[-1] reads an integer before the start of p
	this is almost never useful, and never necessary
	but it won't be checked for
	recommendation: never do this on purpose

4[p] <- actually the same as p[4], because early C compilers were weird

	in general, x[y] translates to *(x + y)
		<- the + turns into pointer arithmetic, if x or y is a pointer
		<- the * dereferences the new pointer
		<- conclusion: 4[p] is the same as p[4] is the same as *(p + 4)

	this is trivia: don't write code like this; it is confusing for no reason

what about array accessing?

	a[1] works the same way as p[1]

		"array decay" transforms array references to pointers in most circumstances

	often, we ignore the difference between array variables and array objects

differences
	pointers can point to different arrays
	array variables always refer to the same array
	multidimensional arrays

	int a[4]; // a is a fixed stack object
	int *a = malloc(4 * sizeof(int)); // *a is a new heap object

aside: char **argv
	What is this?
	argv refers to an array of pointers
	argc gives us the length of the array

	argv is a char** (double-pointer)
	argv[0] is a char* (pointer to a char or string)
	argv[0][0] is a char (the first byte in argument 0)

	The C language standard guarantees that the pointers in argv always point
		to '\0'-terminated strings

	we can call this a multi-level array, because each "row" can have a different
		length

why doesn't C do bounds checking?
- we don't keep the size information anywhere at run-time
	-> the language can't check, even if it wanted to
- most of the time, your algorithms already rely on the size of the array

	if I write this loop,

		for (i = 0; i < arraylen; ++i) {
			total += a[i];
		}

	note that I already check i each iteration to make sure we didn't go past the end
	the C philosophy says that having the array access do the check a second time
		is wasteful
	the Java philosophy says that safety is more important (what if arraylen isn't
		actually the length of a?)

	C puts you in charge of safety, so don't screw up

If you want, you can write your own functions

	struct safearray {
		unsigned length;
		int *data;
	};

	int saferead(struct safearray *a, int i) {
		if (i >= 0 && i < a->length) {
			return a->data[i];
		}

		return 0; // or explode with exit(EXIT_FAILURE) or something
	}

next time:
	why is malloc declared as void *malloc(size_t)?