What does this do?

```
	int foo(char *p)
	{
		int i = 0;
		while (*p++) ++i;
		return i;
	}

while (*p++)   // when does this loop exit?
```

note: we have unary * and postfix ++
	pointer dereference and post-increment

this is either *(p++) or (*p)++
	*(p++) returns the value that p points to, and then increments p
	(*p)++ increment the char that p points to

	++ has higher precedence than *, so *p++ is the same as *(p++)

	recall: p++ evaluates to the current value of p, and increments p later
		++p increments p first, and evaluates to the new value

when is a char considered false?
	chars are integers, so the char with int value 0 is false
		-> '\0'
		-> 0 (another way to write the same value)

so the loop could be rewritten while (*(p++) != '\0') ++i;

or even: while (*p != '\0') { ++p; ++i; }

```
char *p = "false";
	p would be considered true, because only NULL is false for pointers
```

contrast with

```
	int i;
	for (i = 0; p[i] != '\0'; i++) {};

		// theoretically less efficient (due to array indexing),
		// but compilers can optimize it
```

note:

```
	p++  evaluates to current value of p and increments p
	*p++ evaluates to the value that p points to and increments p

	if we have char *p

		p++ is a pointer (char *)
		*p++ is a char
```

assignment 1

Assignment 1 is out, due in a week

This is not a project: work by yourself

It is primarily about getting comfortable with the compiler and writing code

Submit your final program, including

	the rewritten triangle() using a for loop, h_triangle(), and v_triangle()

Due to a transcription error, the code I posted prints a blank line above the triangle
feel free to ignore that (it should have been i = 1;)

any library in the C standard library is fair game
	-> as long is it is available on the iLab

This will be hand-graded, so don't make life too hard for the TA

If you want to use make, you can use a simple makefile

```
	triangle: triangle.c
		gcc -Wall triangle.c -o triangle
```

this is entirely optional

malloc & free
----

last time: we talked about three kinds of objects
	global objects exist for the entire process runtime
	stack objects are created when a function starts, and destroyed when it returns

global and stack objects are managed for you
	every variable refers to a stack or global object

heap objects are managed by the program
	program decides when to allocate/deallocate heap objects

	pointer variables can point to heap objects (or any object)

We allocate space on the heap using malloc()

```
	void *malloc(size_t);   // #include <stdlib.h>

		size_t is just an unsigned integer type that is "big enough"
```

what is void*?

```
	void* is the generic pointer type
		a void* is a pointer where we don't specify the type of what it points to

	you can't do anything interesting with a void*
		-> you can't dereference, because you don't know the size of the data it points to
		-> you can't sensibly do pointer arithmetic, because we don't know the size

	what can you do?
		pass to functions
		receive from functions
		cast to some other pointer type
```

Use of malloc: allocate space for 20 integers

```
	int *p = (int *) malloc(sizeof(int) * 20);

		The "(int *)" says we want to "cast" the void pointer to an int pointer
		This does not do anything at run-time: it is just bookkeeping for the compiler

	Note: C does not cast pointers by default (no promotion)

		char *p;
		int *q;

		p = q;  // type error: char* and int* are incompatible

		p = (char *) p;  // tells C to do the cast anyway

	Casting pointers is a good way to get bizarre unexpected behavior
		or bizarre expected behavior

	Exception: C automatically casts to/from void*

		int *p = malloc(20 * sizeof(int));
			// not including the (int *) is fine; it will produce the same code

	Whether to explicitly cast the result of malloc is a matter of taste

	Including the cast may catch some errors

		p = (float *) malloc(20 * sizeof(float));
				// this will be a type error, because we declared p as int *

		p = (int *) malloc(20 * sizeof(float));
				// valid C, but probably not intended
```

Why does malloc return void*?
- C does not have type parameters
	- no way to have malloc return a specified pointer type
	- we don't want to have a different malloc for every possible pointer type
	- void * is how we get around not having polymorphism

Related function: free

```
	void free(void *);

		free takes a void *, because it needs to work with any type of pointer


	int *p = malloc(...);  // implicitly cast void* to int*
	...
	free(p);  // implicitly cast int* to void*
```

free deallocates heap objects
	-> the space used by the heap object will be available for later use

only give free addresses obtained from malloc!

BAD:

```
	int i;
	free(&i);

		compiler will accept this, but your program is undefined!
```

BAD

```
	int *p = malloc(100 * sizeof(int));
	free(p + 1);

		don't free in the middle of the heap object!
```

OKAY

```
	int *p, *q;

	p = malloc(100 * sizeof(int));
	q = p;
	free(q);

		the variable isn't important
		we just have to pass an address that we got from malloc
```

BAD: use after free

```
	free(p);
	int i = *p;   // can't use p after it has been freed
```

BAD: double free

```
	free(p);
	free(p);   // can't free an object that no longer exists
```

```
struct node {
	int data;
	struct node *next;
};

struct node *head = NULL;

void push(int i)
{
	// allocate a new linked list node on the heap
	struct node *new = malloc(sizeof(struct node));

	// add the new node to the heap
	new->data = i;
	new->next = head;
	head = new;
}

	// note that "new" is a stack variable that ceases to exist when push returns
	// but the struct it pointed to persists in the heap
		// in Java, we use "new" instead of malloc

int pop()
{
	if (!head) return 0;   // do something if the list is empty

	struct node *old = head; // hold on to ptr to current head
	head = head->next;  // remove current head from stack

	int i = old->data;  // remember the value we are popping
	free(old);          // deallocate old head

	return i;
}
```

function pointers

Functions exist in memory and we can find their addresses and store those addresses
in variables or pass them to other functions

	- the syntax for this is not very friendly

Let's say I want a pointer to a function that takes two ints and returns an int

```
	int (*p)(int, int);


		p
		is a pointer (*p)
		to a function that takes two integers (*p)(int, int)
		and returns an integer (int (*p)(int, int))


	How do we obtain a function pointer?
		- write the name of a function with no arguments
			... and that's it; we can't allocate new functions at run-time or
			    do pointer arithmetic
			... functions are always global objects

	We can pass function pointers as arguments to other functions
```

Why would we do this?
	1. multithreading requires us to indicate what code the new thread will execute
	2. many algorithms are parameterized by sub-algorithms

Example: sorting
	to write a general sort function, we have to know how to compare the objects we
		are sorting
	there is no way to associate a comparison function with a type
	-> instead, we pass the comparison function to our sorting function as an argument

```
// assume data is non-NULL, and len > 0
int infimum(int *data, int len, int (*compare)(int, int))
{
	int inf = data[0];
	int i;

	for (i = 1; i < len; ++i) {
		if (compare(inf, data[i]) > 0) inf = data[i];

			// note that calling a function pointer looks just like a regular
			// function call!
	}

	return inf;
}
```