```c
// Single-line comments start with // - only available in C99 and later.

/*
Multi-line comments look like this. They work in C89 as well.
*/

/*
Multi-line comments don't nest /* Be careful */  // comment ends on this line...
*/ // ...not this one!

// Constants: #define <keyword>
// Constants are written in all-caps out of convention, not requirement
#define DAYS_IN_YEAR 365

// Enumeration constants are also ways to declare constants.
// All statements must end with a semicolon
enum days {SUN, MON, TUE, WED, THU, FRI, SAT};
// SUN gets 0, MON gets 1, TUE gets 2, etc.

// Enumeration values can also be specified
enum days {SUN = 1, MON, TUE, WED = 99, THU, FRI, SAT};
// MON gets 2 automatically, TUE gets 3, etc.
// WED get 99, THU gets 100, FRI gets 101, etc.

// Import headers with #include
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// File names between <angle brackets> tell the compiler to look in your system
// libraries for the headers.
// For your own headers, use double quotes instead of angle brackets, and
// provide the path:
#include "my_header.h"   // local file
#include "../my_lib/my_lib_header.h" //relative path

// Declare function signatures in advance in a .h file, or at the top of
// your .c file.
void function_1();
int function_2(void);

// At a minimum, you must declare a 'function prototype' before its use in any function.
// Normally, prototypes are placed at the top of a file before any function definition.
int add_two_ints(int x1, int x2); // function prototype
// although `int add_two_ints(int, int);` is also valid (no need to name the args),
// it is recommended to name arguments in the prototype as well for easier inspection

// Function prototypes are not necessary if the function definition comes before
// any other function that calls that function. However, it's standard practice to
// always add the function prototype to a header file (*.h) and then #include that
// file at the top. This prevents any issues where a function might be called
// before the compiler knows of its existence, while also giving the developer a
// clean header file to share with the rest of the project.

// Your program's entry point is a function called "main". The return type can
// be anything, however most operating systems expect a return type of `int` for
// error code processing.
int main(void) {
  // your program
}

// The command line arguments used to run your program are also passed to main
// argc being the number of arguments - your program's name counts as 1
// argv is an array of character arrays - containing the arguments themselves
// argv[0] = name of your program, argv[1] = first argument, etc.
int main (int argc, char** argv)
{
  // print output using printf, for "print formatted"
  // %d is an integer, \n is a newline
  printf("%d\n", 0); // => Prints 0

  // take input using scanf
  // '&' is used to define the location
  // where we want to store the input value
  int input;
  scanf("%d", &input);

  ///////////////////////////////////////
  // Types
  ///////////////////////////////////////

  // Compilers that are not C99-compliant require that variables MUST be
  // declared at the top of the current block scope.
  // Compilers that ARE C99-compliant allow declarations near the point where
  // the value is used.
  // For the sake of the tutorial, variables are declared dynamically under
  // C99-compliant standards.

  // ints are usually 4 bytes (use the `sizeof` operator to check)
  int x_int = 0;

  // shorts are usually 2 bytes (use the `sizeof` operator to check)
  short x_short = 0;

  // chars are defined as the smallest addressable unit for a processor.
  // This is usually 1 byte, but for some systems it can be more (ex. for TMS320 from TI it's
2 bytes).
  char x_char = 0;
  char y_char = 'y'; // Char literals are quoted with ''

  // longs are often 4 to 8 bytes; long longs are guaranteed to be at least
  // 8 bytes
  long x_long = 0;
  long long x_long_long = 0;

  // floats are usually 32-bit floating point numbers
  float x_float = 0.0f; // 'f' suffix here denotes floating point literal

  // doubles are usually 64-bit floating-point numbers
  double x_double = 0.0; // real numbers without any suffix are doubles

  // integer types may be unsigned (greater than or equal to zero)
  unsigned short ux_short;
  unsigned int ux_int;
  unsigned long long ux_long_long;

  // chars inside single quotes are integers in machine's character set.
  '0'; // => 48 in the ASCII character set.
  'A'; // => 65 in the ASCII character set.

  // sizeof(T) gives you the size of a variable with type T in bytes
  // sizeof(obj) yields the size of the expression (variable, literal, etc.).
  printf("%zu\n", sizeof(int)); // => 4 (on most machines with 4-byte words)

  // If the argument of the `sizeof` operator is an expression, then its argument
  // is not evaluated (except VLAs (see below)).
  // The value it yields in this case is a compile-time constant.
  int a = 1;
  // size_t is an unsigned integer type of at least 2 bytes used to represent
  // the size of an object.
  size_t size = sizeof(a++); // a++ is not evaluated
  printf("sizeof(a++) = %zu where a = %d\n", size, a);
  // prints "sizeof(a++) = 4 where a = 1" (on a 32-bit architecture)
```

```c
// Arrays must be initialized with a concrete size.
char my_char_array[20]; // This array occupies 1 * 20 = 20 bytes
int my_int_array[20]; // This array occupies 4 * 20 = 80 bytes
// (assuming 4-byte words)

// You can initialize an array of twenty ints that all equal 0 thusly:
int my_array[20] = {0};
// where the "{0}" part is called an "array initializer".
// All elements (if any) past the ones in the initializer are initialized to 0:
int my_array[5] = {1, 2};
// So my_array now has five elements, all but the first two of which are 0:
// [1, 2, 0, 0, 0]
// NOTE that you get away without explicitly declaring the size
// of the array IF you initialize the array on the same line:
int my_array[] = {0};
// NOTE that, when not declaring the size, the size of the array is the number
// of elements in the initializer. With "{0}", my_array is now of size one: [0]
// To evaluate the size of the array at run-time, divide its byte size by the
// byte size of its element type:
size_t my_array_size = sizeof(my_array) / sizeof(my_array[0]);
// WARNING You should evaluate the size *before* you begin passing the array
// to functions (see later discussion) because arrays get "downgraded" to
// raw pointers when they are passed to functions (so the statement above
// will produce the wrong result inside the function).

// Indexing an array is like other languages -- or,
// rather, other languages are like C
my_array[0]; // => 0

// Arrays are mutable; it's just memory!
my_array[1] = 2;
printf("%d\n", my_array[1]); // => 2

// In C99 (and as an optional feature in C11), variable-length arrays (VLAs)
// can be declared as well. The size of such an array need not be a compile
// time constant:
printf("Enter the array size: "); // ask the user for an array size
int array_size;
fscanf(stdin, "%d", &array_size);
int var_length_array[array_size]; // declare the VLA
printf("sizeof array = %zu\n", sizeof var_length_array);

// Example:
// > Enter the array size: 10
// > sizeof array = 40

// Strings are just arrays of chars terminated by a NULL (0x00) byte,
// represented in strings as the special character '\0'.
// (We don't have to include the NULL byte in string literals; the compiler
//  inserts it at the end of the array for us.)
char a_string[20] = "This is a string";
printf("%s\n", a_string); // %s formats a string

printf("%d\n", a_string[16]); // => 0
// i.e., byte #17 is 0 (as are 18, 19, and 20)

// If we have characters between single quotes, that's a character literal.
// It's of type `int`, and *not* `char` (for historical reasons).
int cha = 'a'; // fine
char chb = 'a'; // fine too (implicit conversion from int to char)

// Multi-dimensional arrays:
int multi_array[2][5] = {
  {1, 2, 3, 4, 5},
  {6, 7, 8, 9, 0}
};
// access elements:
```

```c
int array_int = multi_array[0][2]; // => 3

///////////////////////////////////////
// Operators
///////////////////////////////////////

// Shorthands for multiple declarations:
int i1 = 1, i2 = 2;
float f1 = 1.0, f2 = 2.0;

int b, c;
b = c = 0;

// Arithmetic is straightforward
i1 + i2; // => 3
i2 - i1; // => 1
i2 * i1; // => 2
i1 / i2; // => 0 (0.5, but truncated towards 0)

// You need to cast at least one integer to float to get a floating-point result
(float)i1 / i2; // => 0.5f
i1 / (double)i2; // => 0.5 // Same with double
f1 / f2; // => 0.5, plus or minus epsilon

// Floating-point numbers are defined by IEEE 754, thus cannot store perfectly
// exact values. For instance, the following does not produce expected results
// because 0.1 might actually be 0.099999999999 inside the computer, and 0.3
// might be stored as 0.300000000001.
(0.1 + 0.1 + 0.1) != 0.3; // => 1 (true)
// and it is NOT associative due to reasons mentioned above.
1 + (1e123 - 1e123) != (1 + 1e123) - 1e123; // => 1 (true)
// this notation is scientific notations for numbers: 1e123 = 1*10^123

// It is important to note that most all systems have used IEEE 754 to
// represent floating points. Even python, used for scientific computing,
// eventually calls C which uses IEEE 754. It is mentioned this way not to
// indicate that this is a poor implementation, but instead as a warning
// that when doing floating point comparisons, a little bit of error (epsilon)
// needs to be considered.

// Modulo is there as well, but be careful if arguments are negative
11 % 3;    // => 2 as 11 = 2 + 3*x (x=3)
(-11) % 3; // => -2, as one would expect
11 % (-3); // => 2 and not -2, and it's quite counter intuitive

// Comparison operators are probably familiar, but
// there is no Boolean type in C. We use ints instead.
// (C99 introduced the _Bool type provided in stdbool.h)
// 0 is false, anything else is true. (The comparison
// operators always yield 0 or 1.)
3 == 2; // => 0 (false)
3 != 2; // => 1 (true)
3 > 2;  // => 1
3 < 2;  // => 0
2 <= 2; // => 1
2 >= 2; // => 1

// C is not Python - comparisons do NOT chain.
// Warning: The line below will compile, but it means `(0 < a) < 2`.
// This expression is always true, because (0 < a) could be either 1 or 0.
// In this case it's 1, because (0 < 1).
int between_0_and_2 = 0 < a < 2;
// Instead use:
int between_0_and_2 = 0 < a && a < 2;

// Logic works on ints
!3; // => 0 (Logical not)
```

```
!0; // => 1
1 && 1; // => 1 (Logical and)
0 && 1; // => 0
0 || 1; // => 1 (Logical or)
0 || 0; // => 0

// Conditional ternary expression ( ? : )
int e = 5;
int f = 10;
int z;
z = (e > f) ? e : f; // => 10 "if e > f return e, else return f."

// Increment and decrement operators:
int j = 0;
int s = j++; // Return j THEN increase j. (s = 0, j = 1)
s = ++j; // Increase j THEN return j. (s = 2, j = 2)
// same with j-- and --j

// Bitwise operators!
~0x0F; // => 0xFFFFFFF0 (bitwise negation, "1's complement", example result for 32-bit int)
0x0F & 0xF0; // => 0x00 (bitwise AND)
0x0F | 0xF0; // => 0xFF (bitwise OR)
0x04 ^ 0x0F; // => 0x0B (bitwise XOR)
0x01 << 1; // => 0x02 (bitwise left shift (by 1))
0x02 >> 1; // => 0x01 (bitwise right shift (by 1))

// Be careful when shifting signed integers - the following are undefined:
// - shifting into the sign bit of a signed integer (int a = 1 << 31)
// - left-shifting a negative number (int a = -1 << 2)
// - shifting by an offset which is >= the width of the type of the LHS:
//   int a = 1 << 32; // UB if int is 32 bits wide

///////////////////////////////////////
// Control Structures
///////////////////////////////////////

if (0) {
  printf("I am never run\n");
} else if (0) {
  printf("I am also never run\n");
} else {
  printf("I print\n");
}

// While loops exist
int ii = 0;
while (ii < 10) { //ANY value less than ten is true.
  printf("%d, ", ii++); // ii++ increments ii AFTER using its current value.
} // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

int kk = 0;
do {
  printf("%d, ", kk);
} while (++kk < 10); // ++kk increments kk BEFORE using its current value.
// => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

// For loops too
int jj;
for (jj=0; jj < 10; jj++) {
  printf("%d, ", jj);
} // => prints "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");
```

```c
// *****NOTES*****:
// Loops and Functions MUST have a body. If no body is needed:
int i;
for (i = 0; i <= 5; i++) {
  ; // use semicolon to act as the body (null statement)
}
// Or
for (i = 0; i <= 5; i++);

// branching with multiple choices: switch()
switch (a) {
case 0: // labels need to be integral *constant* expressions (such as enums)
  printf("Hey, 'a' equals 0!\n");
  break; // if you don't break, control flow falls over labels
case 1:
  printf("Huh, 'a' equals 1!\n");
  break;
  // Be careful - without a "break", execution continues until the
  // next "break" is reached.
case 3:
case 4:
  printf("Look at that.. 'a' is either 3, or 4\n");
  break;
default:
  // if `some_integral_expression` didn't match any of the labels
  fputs("Error!\n", stderr);
  exit(-1);
  break;
}
/*
  Using "goto" in C
*/
typedef enum { false, true } bool;
// for C don't have bool as data type before C99 :(
bool disaster = false;
int i, j;
for(i=0; i<100; ++i)
for(j=0; j<100; ++j)
{
  if((i + j) >= 150)
      disaster = true;
  if(disaster)
      goto error;  // exit both for loops
}
error: // this is a label that you can "jump" to with "goto error;"
printf("Error occurred at i = %d & j = %d.\n", i, j);
/*
  https://ideone.com/GuPhd6
  this will print out "Error occurred at i = 51 & j = 99."
*/
/*
  it is generally considered bad practice to do so, except if
  you really know what you are doing. See
  https://en.wikipedia.org/wiki/Spaghetti_code#Meaning
*/

///////////////////////////////////////
// Typecasting
///////////////////////////////////////

// Every value in C has a type, but you can cast one value into another type
// if you want (with some constraints).

int x_hex = 0x01; // You can assign vars with hex literals
                  // binary is not in the standard, but allowed by some
                  // compilers (x_bin = 0b0010010110)
// Casting between types will attempt to preserve their numeric values
printf("%d\n", x_hex); // => Prints 1
printf("%d\n", (short) x_hex); // => Prints 1
printf("%d\n", (char) x_hex); // => Prints 1

// If you assign a value greater than a types max val, it will rollover
// without warning.
printf("%d\n", (unsigned char) 257); // => 1 (Max char = 255 if char is 8 bits long)

// For determining the max value of a `char`, a `signed char` and an `unsigned char`,
// respectively, use the CHAR_MAX, SCHAR_MAX and UCHAR_MAX macros from <limits.h>

// Integral types can be cast to floating-point types, and vice-versa.
printf("%f\n", (double) 100); // %f always formats a double...
printf("%f\n", (float)  100); // ...even with a float.
printf("%d\n", (char)100.0);

///////////////////////////////////////
// Pointers
///////////////////////////////////////

// A pointer is a variable declared to store a memory address. Its declaration will
// also tell you the type of data it points to. You can retrieve the memory address
// of your variables, then mess with them.

int x = 0;
printf("%p\n", (void *)&x); // Use & to retrieve the address of a variable
// (%p formats an object pointer of type void *)
// => Prints some address in memory;

// Pointers start with * in their declaration
int *px, not_a_pointer; // px is a pointer to an int
px = &x; // Stores the address of x in px
printf("%p\n", (void *)px); // => Prints some address in memory
printf("%zu, %zu\n", sizeof(px), sizeof(not_a_pointer));
// => Prints "8, 4" on a typical 64-bit system

// To retrieve the value at the address a pointer is pointing to,
// put * in front to dereference it.
// Note: yes, it may be confusing that '*' is used for _both_ declaring a
// pointer and dereferencing it.
printf("%d\n", *px); // => Prints 0, the value of x

// You can also change the value the pointer is pointing to.
// We'll have to wrap the dereference in parenthesis because
// ++ has a higher precedence than *.
(*px)++; // Increment the value px is pointing to by 1
printf("%d\n", *px); // => Prints 1
printf("%d\n", x); // => Prints 1

// Arrays are a good way to allocate a contiguous block of memory
int x_array[20]; //declares array of size 20 (cannot change size)
int xx;
for (xx = 0; xx < 20; xx++) {
  x_array[xx] = 20 - xx;
} // Initialize x_array to 20, 19, 18,... 2, 1

// Declare a pointer of type int and initialize it to point to x_array
int* x_ptr = x_array;
// x_ptr now points to the first element in the array (the integer 20).
// This works because arrays often decay into pointers to their first element.
// For example, when an array is passed to a function or is assigned to a pointer,
// it decays into (implicitly converted to) a pointer.
// Exceptions: when the array is the argument of the `&` (address-of) operator:
int arr[10];
int (*ptr_to_arr)[10] = &arr; // &arr is NOT of type `int *`!
```

```c
  // It's of type "pointer to array" (of ten `int`s).
  // or when the array is a string literal used for initializing a char array:
  char otherarr[] = "foobarbazquirk";
  // or when it's the argument of the `sizeof` or `alignof` operator:
  int arraythethird[10];
  int *ptr = arraythethird; // equivalent with int *ptr = &arr[0];
  printf("%zu, %zu\n", sizeof(arraythethird), sizeof(ptr));
  // probably prints "40, 4" or "40, 8"

  // Pointers are incremented and decremented based on their type
  // (this is called pointer arithmetic)
  printf("%d\n", *(x_ptr + 1)); // => Prints 19
  printf("%d\n", x_array[1]); // => Prints 19

  // You can also dynamically allocate contiguous blocks of memory with the
  // standard library function malloc, which takes one argument of type size_t
  // representing the number of bytes to allocate (usually from the heap, although this
  // may not be true on e.g. embedded systems - the C standard says nothing about it).
  int *my_ptr = malloc(sizeof(*my_ptr) * 20);
  for (xx = 0; xx < 20; xx++) {
    *(my_ptr + xx) = 20 - xx; // my_ptr[xx] = 20-xx
  } // Initialize memory to 20, 19, 18, 17... 2, 1 (as ints)

  // Be careful passing user-provided values to malloc! If you want
  // to be safe, you can use calloc instead (which, unlike malloc, also zeros out the memory)
  int* my_other_ptr = calloc(20, sizeof(int));

  // Note that there is no standard way to get the length of a
  // dynamically allocated array in C. Because of this, if your arrays are
  // going to be passed around your program a lot, you need another variable
  // to keep track of the number of elements (size) of an array. See the
  // functions section for more info.
  size_t size = 10;
  int *my_arr = calloc(size, sizeof(int));
  // Add an element to the array
  size++;
  my_arr = realloc(my_arr, sizeof(int) * size);
  if (my_arr == NULL) {
    //Remember to check for realloc failure!
    return
  }
  my_arr[10] = 5;

  // Dereferencing memory that you haven't allocated gives
  // "unpredictable results" - the program is said to invoke "undefined behavior"
  printf("%d\n", *(my_ptr + 21)); // => Prints who-knows-what? It may even crash.

  // When you're done with a malloc'd block of memory, you need to free it,
  // or else no one else can use it until your program terminates
  // (this is called a "memory leak"):
  free(my_ptr);

  // Strings are arrays of char, but they are usually represented as a
  // pointer-to-char (which is a pointer to the first element of the array).
  // It's good practice to use `const char *' when referring to a string literal,
  // since string literals shall not be modified (i.e. "foo"[0] = 'a' is ILLEGAL.)
  const char *my_str = "This is my very own string literal";
  printf("%c\n", *my_str); // => 'T'

  // This is not the case if the string is an array
  // (potentially initialized with a string literal)
  // that resides in writable memory, as in:
  char foo[] = "foo";
  foo[0] = 'a'; // this is legal, foo now contains "aoo"

  function_1();
} // end main function
```

```c
///////////////////////////////////////
// Functions
///////////////////////////////////////

// Function declaration syntax:
// <return type> <function name>(<args>)

int add_two_ints(int x1, int x2)
{
  return x1 + x2; // Use return to return a value
}

/*
Functions are call by value. When a function is called, the arguments passed to
the function are copies of the original arguments (except arrays). Anything you
do to the arguments in the function do not change the value of the original
argument where the function was called.

Use pointers if you need to edit the original argument values (arrays are always
passed in as pointers).

Example: in-place string reversal
*/

// A void function returns no value
void str_reverse(char *str_in)
{
  char tmp;
  size_t ii = 0;
  size_t len = strlen(str_in); // `strlen()` is part of the c standard library
                               // NOTE: length returned by `strlen` DOESN'T
                               //       include the terminating NULL byte ('\0')
  // in C99 and newer versions, you can directly declare loop control variables
  // in the loop's parentheses. e.g., `for (size_t ii = 0; ...`
  for (ii = 0; ii < len / 2; ii++) {
    tmp = str_in[ii];
    str_in[ii] = str_in[len - ii - 1]; // ii-th char from end
    str_in[len - ii - 1] = tmp;
  }
}
//NOTE: string.h header file needs to be included to use strlen()

/*
char c[] = "This is a test.";
str_reverse(c);
printf("%s\n", c); // => ".tset a si sihT"
*/
/*
as we can return only one variable
to change values of more than one variables we use call by reference
*/
void swapTwoNumbers(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
/*
int first = 10;
int second = 20;
printf("first: %d\nsecond: %d\n", first, second);
swapTwoNumbers(&first, &second);
printf("first: %d\nsecond: %d\n", first, second);
// values will be swapped
*/
```

```c
// Return multiple values.
// C does not allow for returning multiple values with the return statement. If
// you would like to return multiple values, then the caller must pass in the
// variables where they would like the returned values to go. These variables must
// be passed in as pointers such that the function can modify them.
int return_multiple( int *array_of_3, int *ret1, int *ret2, int *ret3)
{
    if(array_of_3 == NULL)
        return 0; //return error code (false)

    //de-reference the pointer so we modify its value
   *ret1 = array_of_3[0];
   *ret2 = array_of_3[1];
   *ret3 = array_of_3[2];

   return 1; //return error code (true)
}

/*
With regards to arrays, they will always be passed to functions
as pointers. Even if you statically allocate an array like `arr[10]`,
it still gets passed as a pointer to the first element in any function calls.
Again, there is no standard way to get the size of a dynamically allocated
array in C.
*/
// Size must be passed!
// Otherwise, this function has no way of knowing how big the array is.
void printIntArray(int *arr, size_t size) {
    int i;
    for (i = 0; i < size; i++) {
        printf("arr[%d] is: %d\n", i, arr[i]);
    }
}
/*
int my_arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int size = 10;
printIntArray(my_arr, size);
// will print "arr[0] is: 1" etc
*/

// if referring to external variables outside function, you should use the extern keyword.
int i = 0;
void testFunc() {
  extern int i; //i here is now using external variable i
}

// make external variables private to source file with static:
static int j = 0; //other files using testFunc2() cannot access variable j
void testFunc2() {
  extern int j;
}
// The static keyword makes a variable inaccessible to code outside the
// compilation unit. (On almost all systems, a "compilation unit" is a .c
// file.) static can apply both to global (to the compilation unit) variables,
// functions, and function-local variables. When using static with
// function-local variables, the variable is effectively global and retains its
// value across function calls, but is only accessible within the function it
// is declared in. Additionally, static variables are initialized to 0 if not
// declared with some other starting value.
//**You may also declare functions as static to make them private**

///////////////////////////////////////
// User-defined types and structs
///////////////////////////////////////

// Typedefs can be used to create type aliases
typedef int my_type;
```

```c
my_type my_type_var = 0;

// Structs are just collections of data, the members are allocated sequentially,
// in the order they are written:
struct rectangle {
  int width;
  int height;
};

// It's not generally true that
// sizeof(struct rectangle) == sizeof(int) + sizeof(int)
// due to potential padding between the structure members (this is for alignment
// reasons). [1]

void function_1()
{
  struct rectangle my_rec = { 1, 2 }; // Fields can be initialized immediately

  // Access struct members with .
  my_rec.width = 10;
  my_rec.height = 20;

  // You can declare pointers to structs
  struct rectangle *my_rec_ptr = &my_rec;

  // Use dereferencing to set struct pointer members...
  (*my_rec_ptr).width = 30;

  // ... or even better: prefer the -> shorthand for the sake of readability
  my_rec_ptr->height = 10; // Same as (*my_rec_ptr).height = 10;
}

// You can apply a typedef to a struct for convenience
typedef struct rectangle rect;

int area(rect r)
{
  return r.width * r.height;
}

// Typedefs can also be defined right during struct definition
typedef struct {
  int width;
  int height;
} rect;
// Like before, doing this means one can type
rect r;
// instead of having to type
struct rectangle r;

// if you have large structs, you can pass them "by pointer" to avoid copying
// the whole struct:
int areaptr(const rect *r)
{
  return r->width * r->height;
}

///////////////////////////////////////
// Function pointers
///////////////////////////////////////
/*
At run time, functions are located at known memory addresses. Function pointers are
much like any other pointer (they just store a memory address), but can be used
to invoke functions directly, and to pass handlers (or callback functions) around.
However, definition syntax may be initially confusing.

Example: use str_reverse from a pointer
```

```c
*/
void str_reverse_through_pointer(char *str_in) {
  // Define a function pointer variable, named f.
  void (*f)(char *); // Signature should exactly match the target function.
  f = &str_reverse; // Assign the address for the actual function (determined at run time)
  // f = str_reverse; would work as well - functions decay into pointers, similar to arrays
  (*f)(str_in); // Just calling the function through the pointer
  // f(str_in); // That's an alternative but equally valid syntax for calling it.
}

/*
As long as function signatures match, you can assign any function to the same pointer.
Function pointers are usually typedef'd for simplicity and readability, as follows:
*/

typedef void (*my_fnp_type)(char *);

// Then used when declaring the actual pointer variable:
// ...
// my_fnp_type f;


///////////////////////////////
// Printing characters with printf()
///////////////////////////////

//Special characters:
/*
'\a'; // alert (bell) character
'\n'; // newline character
'\t'; // tab character (left justifies text)
'\v'; // vertical tab
'\f'; // new page (form feed)
'\r'; // carriage return
'\b'; // backspace character
'\0'; // NULL character. Usually put at end of strings in C.
//   hello\n\0. \0 used by convention to mark end of string.
'\\'; // backslash
'\?'; // question mark
'\''; // single quote
'\"'; // double quote
'\xhh'; // hexadecimal number. Example: '\xb' = vertical tab character
'\0oo'; // octal number. Example: '\013' = vertical tab character

//print formatting:
"%d";    // integer
"%3d";   // integer with minimum of length 3 digits (right justifies text)
"%s";    // string
"%f";    // float
"%ld";   // long
"%3.2f"; // minimum 3 digits left and 2 digits right decimal float
"%7.4s"; // (can do with strings too)
"%c";    // char
"%p";    // pointer. NOTE: need to (void *)-cast the pointer, before passing
         //                 it as an argument to `printf`.
"%x";    // hexadecimal
"%o";    // octal
"%%";    // prints %
*/

////////////////////////////////////////
// Order of Evaluation
////////////////////////////////////////

// From top to bottom, top has higher precedence
//---------------------------------------------------//
//        Operators                   | Associativity //
```

```c
//----------------------------------------------------//
// () [] -> .                          | left to right //
// ! ~ ++ -- + = *(type) sizeof        | right to left //
// * / %                               | left to right //
// + -                                 | left to right //
// << >>                               | left to right //
// < <= > >=                           | left to right //
// == !=                               | left to right //
// &                                   | left to right //
// ^                                   | left to right //
// |                                   | left to right //
// &&                                  | left to right //
// ||                                  | left to right //
// ?:                                  | right to left //
// = += -= *= /= %= &= ^= |= <<= >>= | right to left //
// ,                                   | left to right //
//----------------------------------------------------//

/******************************** Header Files **********************************

Header files are an important part of C as they allow for the connection of C
source files and can simplify code and definitions by separating them into
separate files.

Header files are syntactically similar to C source files but reside in ".h"
files. They can be included in your C source file by using the precompiler
command #include "example.h", given that example.h exists in the same directory
as the C file.
*/

/* A safe guard to prevent the header from being defined too many times. This */
/* happens in the case of circle dependency, the contents of the header is    */
/* already defined.                                                            */
#ifndef EXAMPLE_H /* if EXAMPLE_H is not yet defined. */
#define EXAMPLE_H /* Define the macro EXAMPLE_H. */

/* Other headers can be included in headers and therefore transitively */
/* included into files that include this header.                       */
#include <string.h>

/* Like for c source files, macros can be defined in headers */
/* and used in files that include this header file.          */
#define EXAMPLE_NAME "Dennis Ritchie"

/* Function macros can also be defined.  */
#define ADD(a, b) ((a) + (b))

/* Notice the parenthesis surrounding the arguments -- this is important to   */
/* ensure that a and b don't get expanded in an unexpected way (e.g. consider */
/* MUL(x, y) (x * y); MUL(1 + 2, 3) would expand to (1 + 2 * 3), yielding an  */
/* incorrect result)                                                           */

/* Structs and typedefs can be used for consistency between files. */
typedef struct Node
{
    int val;
    struct Node *next;
} Node;

/* So can enumerations. */
enum traffic_light_state {GREEN, YELLOW, RED};

/* Function prototypes can also be defined here for use in multiple files,  */
/* but it is bad practice to define the function in the header. Definitions */
/* should instead be put in a C file.                                       */
Node createLinkedList(int *vals, int len);
```

```c
/* Beyond the above elements, other definitions should be left to a C source */
/* file. Excessive includes or definitions should also not be contained in   */
/* a header file but instead put into separate headers or a C file.          */

#endif /* End of the if precompiler directive. */
```