Exercises

The file `/home/stone/c/code/stars.c` contains a type definition for a structure that holds four pieces of information about a star: its common name and scientific name, its apparent magnitude, and its distance from the Earth (in parsecs; a parsec is about 3.2616 light-years). It also includes a `main` function that declares and initializes an array of thirty of these structures.

Since these stars are all relatively close to the Earth and unobscured by gas and dust, a star’s absolute magnitude (which indicates how bright it would appear to an observer at a fixed distance of ten parsecs from the star) can be computed from its apparent magnitude $m$ and its distance $D$ in parsecs using the equation

$$M = m - 5 \cdot \log_{10} D - 1.$$  

Exercise 000: Make a copy of `stars.c` and add and test a function that takes a `star` as argument and returns its absolute magnitude $M$ according to the formula given above. (Hint: The `math` library provides functions `log10`, `log10f`, and `log10l` for computing base-ten logarithms of `double`, `float`, and `long double` values, respectively.)

Exercise 001: Add to `main` so that it determines which of the stars in the array is the brightest (as determined by absolute magnitude) and prints out the common and scientific names for that brightest star. Recall that lower magnitudes correspond to greater brightnesses, so you’re looking for the star with an absolute magnitude less than that of every other star in the array.

Let’s imagine a deck of forty-eight playing cards in which there are four suits (fish, dogs, cats, and birds, say) and six ranks (in ascending order: ace, deuce, trey, cater, cinque, and seece). There are two cards in the deck for each combination of a suit and a rank (e.g., there are two copies of the “ace of cats” card).

Exercise 010: Define separate enumeration types `suit` and `rank` to help you describe and model the cards. Then define a structure type `card` that combines the two. Finally, write a program in which you declare and initialize an array of forty-eight `card` values...
representing the deck described above and then print out the elements of the array, in order.

An efficient way to permute the elements of an array so that each permutation is equally likely to be the result is to run a counter from the size of the array down to 2 and, for each value of the counter, to swap the element in position counter - 1 with the element in a position randomly selected from the range from 0 up to, but not including, counter.

Exercise 011: Write and test a function that uses this algorithm to model the operation of shuffling the deck that you constructed in the preceding exercise. (That is: Your function should, as a side effect, randomly permute the elements of the array of card values.)

When we were studying the single- and double-precision representations of floating-point numbers, some students wondered aloud whether there might be a short cut to recovering the bit patterns for those representations without having to execute all the arithmetic that is encoded in the floating-point hardware circuitry. Unions provide exactly such a short cut.

Exercise 100: Write a function display_bits that takes a value of type uint32_t (a thirty-two-bit unsigned integer, as defined in the standard library header file stdint.h) and prints out its bit pattern as a sequence of 0 and 1 characters. (Use shift operations and bitwise operations in a loop to achieve this result.)

Exercise 101: Write a type definition for a union type called visible_float that “superimposes” a value of type float and a value of type uint32_t.

Exercise 110: Using your solutions to the two previous exercises, write a program that extracts a real number from the command line, stores it in the float field of a visible_float, and then invokes the display_bits function on the uint32_t field of that same visible_float. This will display the bit pattern of the float.

In most programming languages, the compiler would block you from carrying out a stunt like this, complaining about a type inconsistency. C is designed to allow programmers more leeway, relying on them to use their power responsibly and not to make bonehead mistakes with it. Don’t let the designers down. Use unions only when you’re sure you know what you’re doing.