17.1 Dynamic Storage Allocation

variable-length arrays ➤8.3

C's data structures are normally fixed in size. For example, the number of elements in an array is fixed once the program has been compiled. (In C99, the length of a variable-length array is determined at run time, but it remains fixed for the rest of the array's lifetime.) Fixed-size data structures can be a problem, since we're forced to choose their sizes when writing a program; we can't change the sizes without modifying the program and compiling it again.

Consider the `inventory` program of Section 16.3, which allows the user to add parts to a database. The database is stored in an array of length 100. To enlarge the capacity of the database, we can increase the size of the array and recompile the program. But no matter how large we make the array, there's always the possibility that it will fill up. Fortunately, all is not lost. C supports ***dynamic storage allocation:*** the ability to allocate storage during program execution. Using dynamic storage allocation, we can design data structures that grow (and shrink) as needed.

Although it's available for all types of data, dynamic storage allocation is used most often for strings, arrays, and structures. Dynamically allocated structures are of particular interest, since we can link them together to form lists, trees, and other data structures.

Memory Allocation Functions

<stdlib.h> header ➤26.2

To allocate storage dynamically, we'll need to call one of the three memory allocation functions declared in the `<stdlib.h>` header:

- `malloc`—Allocates a block of memory but doesn't initialize it.
- `calloc`—Allocates a block of memory and clears it.
- `realloc`—Resizes a previously allocated block of memory.

Of the three, `malloc` is the most used. It's more efficient than `calloc`, since it doesn't have to clear the memory block that it allocates.

When we call a memory allocation function to request a block of memory, the function has no idea what type of data we're planning to store in the block, so it can't return a pointer to an ordinary type such as `int` or `char`. Instead, the function returns a value of type `void *`. A `void *` value is a "generic" pointer—essentially, just a memory address.

Null Pointers

When a memory allocation function is called, there's always a possibility that it won't be able to locate a block of memory large enough to satisfy our request. If