zero if *p is "equal to" *q, and *positive* if *p is "greater than" *q. The terms "less than," "equal to," and "greater than" are in quotes because it's our responsibility to determine how *p and *q are compared.

qsort has the following prototype:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

base must point to the first element in the array. (If only a portion of the array is to be sorted, we'll make base point to the first element in this portion.) In the simplest case, base is just the name of the array. nmemb is the number of elements to be sorted (not necessarily the number of elements in the array). size is the size of each array element, measured in bytes. compar is a pointer to the comparison function. When qsort is called, it sorts the array into ascending order, calling the comparison function whenever it needs to compare array elements.

To sort the inventory array of Section 16.3, we'd use the following call of qsort:

Q&A

```
qsort(inventory, num_parts, sizeof(struct part), compare_parts);
```

Notice that the second argument is num_parts, not MAX_PARTS; we don't want to sort the entire inventory array, just the portion in which parts are currently stored. The last argument, compare_parts, is a function that compares two part structures.

Writing the compare_parts function isn't as easy as you might expect. qsort requires that its parameters have type void *, but we can't access the members of a part structure through a void * pointer; we need a pointer of type struct part * instead. To solve the problem, we'll have compare_parts assign its parameters, p and q, to variables of type struct part *, thereby converting them to the desired type. compare_parts can now use these variables to access the members of the structures that p and q point to. Assuming that we want to sort the inventory array into ascending order by part number, here's how the compare_parts function might look:

```
int compare_parts(const void *p, const void *q)
{
  const struct part *p1 = p;
  const struct part *q1 = q;

  if (p1->number < q1->number)
    return -1;
  else if (p1->number == q1->number)
    return 0;
  else
    return 1;
}
```

The declarations of p1 and q1 include the word const to avoid getting a warning from the compiler. Since p and q are const pointers (indicating that the objects