Sorting Functions

Another example of using function pointers as parameters is a generic sorting function—one that can sort any type of data. Sorting an array is the process of arranging the elements of that array into increasing (or decreasing) order. Sorting an array is a common task in programs, as sorted data can be accessed more efficiently than unsorted data. We will formalize this notion of efficiency later, but for now, imagine trying to find a book in a library where the books are arranged alphabetically (*i.e.*, sorted) versus in one where they are stored in no particular order.

As we will see when we learn more about sorting, there are many different sorting algorithms, but none of them care about the specific type of data, just whether one piece of data is "less than," "equal to," or "greater than" another piece of data. Correspondingly, we could make a generic sorting function—one that can sort an array of any type of data—by having it take a parameter which is a pointer to a function which compares two elements of the array. In fact, the C library provides such a function (which sorts in ascending order—smallest to largest):

```
2

1

                int (*compar)(const void *, const void *));

void qsort(void *base, size_t nmemb, size_t size,
```

The first parameter to this function, **void** * base, is the array to sort. Recall that **void** * is "a pointer to an unspecified type of data"—allowing qsort to take an array of any type. The second parameter, *size_t nmemb* specifies the number of elements (or *members*) in the array (recall that *size_t* is an unsigned integer type appropriate to use for the size of things). The third parameter, *size_t size* specifies the size of each element of the array—that is, how many bytes each elements takes in memory. This information is required because otherwise *qsort* has no way to tell where one element of the array ends and the next begins. The final parameter is the one we are most interested in for this discussion—*compar* is a pointer to a function which takes two **const void** *s and returns a **int**. Here, the **const void** *s point at the two elements to be compared (they are **const** since the

comparison function should not modify the array). The function returns a positive number if the first pointer points at something greater than what the second pointer points at, 0 if they point at equal things, and a negative number for less than.

This description of *qsort* may seem like a lot to take in, but is more easily understood by seeing a couple examples. These two examples will both be *wrapper functions* around *qsort*—small functions that do little or no real computation, but provide a simpler interface. The first example is a wrapper to sort arrays of **int**s:

```
9

}
```

First, we write a comparison function, *compareInts*, whose behavior is compatible with the interface of *qsort*—it takes pointers, which are declared to be **const void** *s, and returns an **int**. Since this function is intended to be used only when sorting arrays of **int**s, it converts **const void** *s to **const int** *s, and dereferences them to get the actual **int**s in the array. Subtracting these two **int**s gives a result which conforms to the expectations of the *qsort* function (it will be positive if the first is greater, 0 if they are equal, or negative if the first is less).

Once this function is written, we can write the *sortIntArray* function which wraps the *qsort* function. Observe how *sortIntArraydoes* no real computation (it just calls qsort to do all the work), but provides a much simpler interface (you pass it an array of **int**s and the number of elements in the array; you should be able to *use* this function to sort an array without any explanation). The *sortIntArrayfunction* passes its arguments as the first two arguments to *qsort*, and then passes **sizeof** (**int**) as the third argument, since each element of the array will be **sizeof** (**int**) bytes large (probably 4, but the correct way to write it is with the **sizeof** operator, in case you ever compile it somewhere where it is not 4). For the fourth argument, the function passes a pointer to compareInts—recall that the name of the function is a pointer to that function. The *qsort* function will then call *compareInts* to determine the relative ordering of the elements in the array.

We can make use of *qsort* in similar ways for other types. For example, we could write some similar functions to sort an array of strings (an array of **const char** *s):

```
11

}
```

We again start by writing a function to compare strings which conforms to the interface required by *qsort*. This function,*compareStrings*, looks much the same as *compareInts*. The main difference is that we use *strcmp* to perform the string comparison. We saw *strcmp* earlier to test if two strings are equal, however, it returns a positive/zero/negative value based on the ordering of the strings, as *qsort* expects.

Note that the pointers passed in are pointers to the elements in the array (that is, they point at the boxes in the array), even though those elements are themselves pointers (since they are strings). When we convert them from **void** *s, we must take care to convert them to the correct type—here, **const char * const** *—and use them appropriately, or our function will be broken in some way. For example, consider the following broken code:

```
3

4

5

6

1

2

  const char * s1 = s1vp;

  const char * s2 = s2vp;

  return strcmp(s1, s2);

}

// BROKEN DO NOT DO THIS!

int compareStrings(const void * s1vp, const void * s2vp) {
```

This code will actually compile without any errors or warnings, but will not work correctly. This is a danger anytime you use **void** *—the flexibility gives you "enough rope to hang yourself" because you have no guarantees that you will use the pointer in the correct way. As we will see later, C++ has features in its type system that allow us to have generic functions in a much safer way.

We can use function pointers pretty much anywhere we can use any other type—not only as the types of parameters, but also as the types of variables, the type of elements in an array, or fields in **struct**s. In fact, as we will see later, function pointers in structures lie at

the heart of object-oriented programming, although object-oriented languages hide this implementation detail from the casual programmer.

memb

Completed