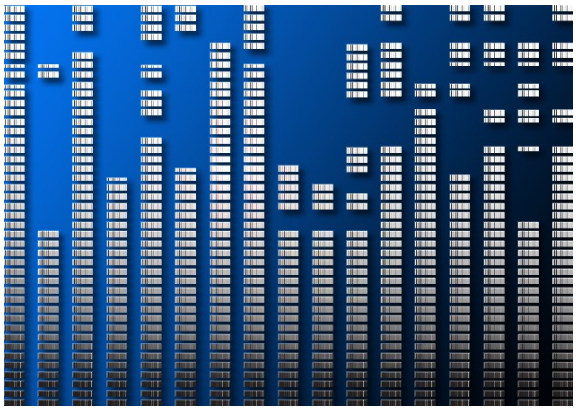
Sorting in C with the **qsort** Function

Introducing qsort



The C standard library provides a function named **qsort**, defined in stdlib.h. The function is designed to sort an array of items of any data type, as long as we accurately provide to the qsort function some important details about our data. The qsort function requires us to specify the following information about the array we need to sort:

1. The address of (a pointer to) the first element of the array to be sorted.
2. The total number of elements in the array.
3. The size, in bytes, of each element in the array.
4. The address of (a pointer to) a C function that will perform a comparison between two elements of the array.

The first three items are straightforward. We know the address of the array – it's either the name of an explicitly-defined array, or a pointer variable containing the address of the first element of an array. We also know the total number of elements in the array, as well as the size of each element. The only part we don't have on hand is the comparison function. Defining and using that comparison function is the primary focus of the following sections.

Role of the Comparison Function

The comparison function controls exactly how the array will be sorted. While sorting the array, whenever qsort needs to compare two array elements, it calls the comparison function we provided (also known as a *callback* function) to perform the actual comparison. By defining a comparison function and then passing this function's address into qsort, we are effectively *teaching* qsort how to compare two items in our array, and thus, we are specifying exactly how the array needs to be sorted.

Because qsort is designed to work with any type of data, from characters to integers to doubles to large structures, it uses generic (void) pointers to the elements of the array, and relies on our comparison function to work directly with the data we're trying to sort. While this approach makes qsort generally useful for sorting all types of data, it introduces some syntax complexity for us when defining our comparison functions.

Here is a function prototype for the type of comparison function expected by qsort:

```
int Compare(const void *p1, const void *p2);
```

The comparison function must return an integer, which indicates the result of the comparison, and the function must receive two pointers. The two pointers will eventually contain the addresses of the two array elements that qsort needs to compare. Generic (void) pointers are

used here, so that the pointers can point to any data type. So, inside our comparison function definitions, we will need to convert those generic pointers to the correct pointer data type, to allow us to access the actual data we need to compare.

The return value of the comparison function must behave as follows:

- If the item pointed to by p1 is equal to the item pointed to by p2, return 0.
- If the item pointed to by p1 is greater than the item pointed to by p1, return a positive value.
- If the item pointed to by p1 is less than the item pointed to by p1, return a negative value.

These comparison functions are typically quite short, but getting access to the data through the void pointers can take some thought. The good news is that there are just a few patterns we need to allow us to create comparison functions for a variety of situations. In the sections below, we will cover several increasingly complex sorting scenarios, which will provide the background needed to write our own comparison functions tailored to our specific requirements.

Sorting an Array of Integers

Let's start with a basic example, sorting an array of integers. Given an array of integers, *myArray*, and a variable containing the number of elements, *elementCount*, in that array:

```
int myArray[] = { 42, 64, 12, 0, -18, -42, 1066, 2016, 2525, 1960 };
unsigned elementCount = sizeof(myArray) / sizeof(myArray[0]);
```

we need to define a function that will compare two integers. Here's a first attempt at a comparison function that assumes we're sorting an array of integers:

```
int CompareInts(const void *p1, const void *p2)
{
	int i1 = *(int *)p1;
	int i2 = *(int *)p2;

	if (i1 == i2)
	{
		return 0;
	}
	else if (i1 > i2)
	{
		return 1;
	}
	else
	{
		return -1;
	}
}
```

The name chosen for the function is arbitrary. We will pass that name to qsort when we want it to use this comparison function to sort an array of integers.

The first order of business inside any of these comparison functions is to somehow get from the void pointers to the data we're actually interested in comparing. The incoming pointers are pointing to two elements of the array, or more specifically to two integers in the array of integers. So, the correct data types of the incoming pointers will be "pointer to int." In this situation, we need to type cast each of the two void pointers to be integer pointers, with a cast of (int *). Doing so will allow us to access the correct number of bytes when we then dereference the pointer with the dereference operator *. (We can't just dereference a void pointer, because the complier has no idea what data type is pointed to. We must perform the type cast to the correct pointer type, so that we can dereference a pointer to the correct data type.)

An expression such as *(int *)p1 first converts the void pointer p1 to a pointer to int, and then dereferences that pointer to get the actual integer it points to. We then do the same for the second pointer. Once we have the two actual integer values stored in local variables, we can proceed with the comparison operation.

The above comparison function meets the qsort requirements, returning 0 if the two integers are equal, a positive 1 if the first is greater than the second, and negative 1 if the second is greater than the first. However, because we're comparing signed integers, we can replace the somewhat cumbersome if-else-if ladder with a single subtraction:

```
int CompareInts(const void *p1, const void *p2)
{
	int i1 = *(const int *)p1;
	int i2 = *(const int *)p2;

	return (i1 - i2);
}
```

This shorter version also meets the requirements of the a qsort comparison function. It returns 0 if the two values are equal, a positive integer of the first value is greater than the second value, and a negative integer if the second value is greater than the first value.

Note that this shorter subtraction technique works only because of the way *signed* integers behave – if we were working with *unsigned* integers, we would have to resort to the if-else-if ladder approach, shown in the previous version of the comparison function. If we were working with a floating-point data type (e.g., double, float, long double), a result of the subtraction that is less than 1.0 would be reported as integer 0, even though the two values are not equal. So, we can't use the more compact subtraction approach with floating-point types either. Thus, the subtraction approach works only with signed integer types (e.g., int, short, char, long, long long).

Now that we have our comparison function that compares two integers in our integer array, we can call qsort to sort the array of integers:

```
qsort(myArray, elementCount, sizeof(myArray[0]), CompareInts);
```

Again, the first parameter is the address of the first element of the array to be sorted, the second parameter is the number of elements in the array, the third parameter is the size (in bytes) of one element of the array, and the fourth parameter is the address of our comparison

function. As usual in C, we must either define or prototype the comparison function *before* we use its name in qsort, so that the compiler knowns what it is, and can check that its prototype matches the type of function expected by qsort.

Sorting an Array of Pointers to Strings

Sorting the contents of an array potentially requires lots of swapping, swapping requires copying, and copying is expensive (in terms of CPU time). Thus, we typically try to minimize the amount of information that needs to be swapped. To accomplish this, we often use an array of pointers to the data, and just tell qsort to swap the pointers instead of swapping the data itself. The data itself remains in memory wherever it was originally, but the pointers to the data are swapped during sorting, so that we end up with an array of pointers that represents the sorted order of the data.

With that in mind, let's look at another very common sorting example, in which we have an array of pointers to strings, where each element contains the address of the first character in a null-terminated string. Given an array of pointers to strings, *myWords*, and a variable containing the number of elements, *elementCount*, in that array:

```
char *myWords[] =          // array of pointers to strings
{
    "zebra", "wildebeest", "duck", "swan", "author", "programmer", "engineer",
    "elephant", "donkey", "house", "home", "computer", "rabbit", "ear", "nose"
};
unsigned elementCount = sizeof(myWords) / sizeof(myWords[0]);
```

we need to define a comparison function that will compare the strings pointed to by the pointers in the array. To compare the actual strings, we can use the strcmp function, defined in string.h. As luck (or actually, as design) would have it, strcmp follows the exact same comparison rules expected by qsort comparison function, returning a 0 if the strings are equal, a positive value if the first string is greater (or appears later in a sort) than the second, and a negative value if the first string is less than (or appears earlier in a sort) than the second. Here is a comparison function that assumes we're sorting an array of pointers to strings:

```
int CompareStrings(const void *p1, const void *p2)
{
       char *s1 = *(char **)p1;
       char *s2 = *(char **)p2;

       return strcmp(s1, s2);
}
```

Again, the name chosen for the function is arbitrary, and we will pass that name of the function (i.e., the address of the function) to qsort when we want it to use this comparison function.

As before, our first job inside the comparison function is get to the pointers to the strings themselves, so that we can pass them to the strcmp function. The incoming pointers are pointing to two elements of the array, or more specifically, they are pointing to the two char pointers within the array. So, the correct data types of the incoming pointers are "pointer to pointer to char" or char **. Thus, we need to type cast each of the two void pointer

parameters to (char **). Then, we use the dereference operator * to get the actual pointers to the strings themselves. To summarize, we received pointers to two array elements, each array element is itself a pointer to a string, and to get at those pointers to strings, we cast the void pointers to char **, and then dereference each pointer to get the char * (pointers to strings) we're interested in comparing with strcmp.

Once we have those two pointers to strings in local variables, we can pass them directly into the strcmp function, and simply return whatever result strcmp gives us, because strcmp uses the same return value conventions as the qsort comparison functions.

Now that we have our comparison function that compares two strings, whose addresses are contained in our array, we can call qsort to sort the array of pointers to strings. Note that qsort is only sorting the pointers in the array, while the actual strings themselves remain in their original memory locations:

```
qsort(myWords, elementCount, sizeof(myWords[0]), CompareStrings);
```

After this call to qsort completes, the pointers in our array will be sorted according to the strings they point to.

Sorting an Array of Pointers to Structures

Finally, let's look an example in which we have an array of pointers to structures, where each array element contains the address of (or pointer to) a structure. Let's start with an array of structures:

```
typedef struct SensorReading
{
	double   temperature;
	double   pressure;
	unsigned sensorNumber;
	char     *pLocationName;
} SensorReading;

SensorReading myReadings[] =
{
	{ 98.6, 29.2, 6, "North Pole" },
	{ 32.1, 28.7, 5, "Cleveland"  },
	{ 21.3, 27.0, 3, "South Pole" },
	{ 55.7, 30.9, 4, "Seattle"    },
	{ 100., 29.9, 1, "Tucson"     },
	{ 18.3, 28.8, 2, "Dallas"     }
};

unsigned elementCount = sizeof(myReadings) / sizeof(myReadings[0]);
```

We could opt to just sort these structures directly, but as discussed in the previous section, we want to avoid copying a lot of data whenever a swap is performed during the sorting process. In this case, if we decided to sort the array of structures, entire structures would have to be copied during swaps. Thus, when working with structures, we typically choose to sort an array of pointers to the structures. Here's how such an array could be dynamically created and initialized:

```
SensorReading **ppReadings = malloc(elementCount * sizeof(SensorReading *));

for (unsigned i = 0; i < elementCount; i++)
{
	ppReadings[i] = &myReadings[i];
}
```

Notice that ppReadings is defined as a pointer to (the address of) the first element of an array of pointers to SensorReading structures. Thus, the data type of ppReadings is a pointer to a pointer to a SensorReading (a pointer to the first element of an array of pointers to SensorReading structures). Each element of this dynamically-allocated array contains a pointer to one SensorReading structure.

The for loop copies the addresses of each element of the array of structures into successive elements of the new pointer array. After this loop completes, ppReadings is an array of pointers to the actual structures in myReadings.

It is this array of pointers that we want to sort using the qsort function. If we wanted to sort the array of pointers based on sensorNumber, and unsigned integer, we would need a comparison function like this:

```
int CompareSensorNumber(const void *p1, const void *p2)
{
	unsigned sensor1 = (*(SensorReading **)p1)->sensorNumber;
	unsigned sensor2 = (*(SensorReading **)p2)->sensorNumber;

	if (sensor1 == sensor2)
	{
		return 0;
	}
	else if (sensor1 > sensor2)
	{
		return 1;
	}
	else
	{
		return -1;
	}
}
```

Once again, because this function receives pointers to the array elements, and each array element is itself a pointer to a SensorReading structure, the data type we need to convert the incoming pointers to is SensorReading ** (a pointer to a pointer to a SensorReading). With the proper pointer type, we can then use the dereference operator to access the actual pointer to the structure itself. Notice that, because the arrow operator -> has higher precedence than the dereference operator *, we must have an extra set of parentheses to cause the dereference to happen first. The expression (*(SensorReading **)p1) gets us to the actual pointer to the SensorReading structure. With this pointer, we can use the arrow operator to get to the member of the structure we're interested in. Using this comparison function, we can sort the array of pointers by sensor number, using the following call to the qsort function:

```
qsort(ppReadings, elementCount, sizeof(ppReadings[0]), CompareSensorNumber);
```

As before, the structures themselves remain where they were originally in memory, and the qsort function has sorted pointers to those structures.

Because our structure contains multiple members, we can use the same techniques already discussed to sort the same array of pointers based on other members of the structure. For example, if we wanted to sort the same array of pointers by pLocationName, a string pointer (pointer to char, or char *) inside the structure, we would create and use a comparison function like this one:

```
int CompareSensorLocationName(const void *p1, const void *p2)
{
	char *s1 = (*(SensorReading **)p1)->pLocationName;
	char *s2 = (*(SensorReading **)p2)->pLocationName;

	return strcmp(s1, s2);
}
```

The only difference between this function and the one in the previous section, which was used to sort an array of pointers to strings, is that here we need to get to the string pointer member inside the structure after dereferencing the pointer. We're still sorting an array of pointers by the contents of a string. But in this case, we're sorting pointers to structures, and the pointer to the string is inside the structure.

The same array of pointers to structures can be sorted using this comparison function, with the following call to qsort:

```
qsort(ppReadings, elementCount, sizeof(ppReadings[0]), CompareSensorLocationName);
```

In fact, by just creating and using more comparison functions, we can potentially sort based on any member of the structure, just by calling qsort and passing in different comparison functions.

Conclusion

Although the syntax can sometimes be a bit daunting, the patterns and approaches discussed here allow us to use the qsort function to sort arrays containing any type of data. And when sorting arrays of pointers to structures, we have the flexibility of choosing to sort based on the contents of any member within that structure, just by creating and using a new comparison function that performs a comparison on that structure member.

The qsort function is a useful tool in the C programmer's toolkit. It may require looking up the requires syntax before using it – this document can be that resource – but by applying the same patterns to new types of data, qsort provides a flexible, general, and efficient solution to the task of sorting data.