
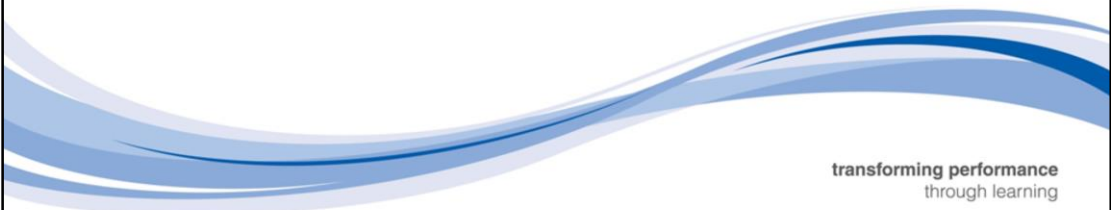


QACADV_v1.0



Advanced C

12 Sorting and Searching



transforming performance
through learning

Sorting and Searching

- **Objectives**
 - Use library functions to work with large arrays of data
- **Contents**
 - Sorting with *qsort()*
 - Comparison functions
 - Sorting pointers vs sorting data
 - Searching with *bsearch()*
- **Practical**
- **Summary**



Sorting in standard C

- **The Standard defines a `qsort()` function which will sort an array of objects *in place***

```
typedef int (*_cmpfn)(const void *, const void *);  
void qsort(void * base, size_t num, size_t width, _cmpfn f);
```

- **It is an implementation specific version of the QuickSort algorithm invented in 1960 by C. A. R. Hoare**
 - Basically it divides the array into two and calls itself recursively to sort an array which is half as big as in the earlier call
 - On average $N \log_2 N$ operations are required to sort N items
 - If two elements are equal their order in the sorted array is unspecified

The Standard requires an implementation of the Quicksort algorithm in the function `qsort()`. It is a very popular algorithm because it is not too difficult to implement and it is a good "general purpose" sort (i.e. it works well in a variety of situations). It also consumes fewer resources than any other sorting method in many situations.

One problem is that the algorithm is recursive. Although the recursion can be removed, the implementation then becomes rather complicated. Some optimisers optimise away recursion, but by no means all of them do. When N nearly sorted data items are thrown at Quicksort, the number of recursive calls will be of the order of N . This could easily exceed the stack limit with large amounts of data. Fortunately there are relatively easy ways to ensure that the worst case does not occur in actual applications.

Many attempts have been made to improve Quicksort. Sorting algorithms are the computing version of mousetraps. Various attempts have been made to invent a better mousetrap. Similarly many and various attempts have been made to invent better sorting algorithms. The algorithm is so well balanced that an improvement in one direction will be offset by bad performance in another.

qsort()

- **The function is generic, i.e. it uses void pointers**
- **Since a void pointer may hold the address of any data item, an array of any data type may be sorted**
- **The sorting comparison function is written by the caller and its address passed in as the fourth parameter**
- **It is called by qsort each time two items in the array need to be compared**
- **The return value is an integer and should be:**
 - Negative if the first item is less than the second
 - Zero if the two items are equal
 - Positive if the first item is greater than the second

The qsort is written to use void pointers and to process lumps of memory. Since a void pointer may point just about anywhere (as seen earlier in the course), this enables qsort to sort any data type.

Characters, integers, floats, etc. pose no problems at all. To be useful, we need qsort to sort structures.

To do this, we provide the function that does the comparing. This is called by qsort whenever it needs to compare two particular items within the array. As seen on the previous page the prototype of this function is:

```
int f(const void * first, const void * second);
```

The function receives a pointer to two data items in the array. If the object at the end of the first pointer is greater than the object at the end of the second pointer, a positive number should be returned. If less, a negative number and if equal, zero.

The "const" before void ensures the function does not accidentally alter the element within the array.

Comparison functions

- Here are examples of comparison functions

```
struct person_tag
{
    int id;
    char name[50];
    char address[100];
    char postcode[15];
};
typedef struct person_tag person;

int comp_name(const void * va,
              const void * vb)
{
    const person * pa = va;
    const person * pb = vb;
    return strcmp(pa->name, pb->name);
}
```

```
int comp_int(const void * va,
             const void * vb)
{
    const int * pa = va;
    const int * pb = vb;
    return (*pa - *pb);
}
```

Above are two examples of comparison functions. The `compInt` function will compare two integers whose addresses are passed as parameters. If you imagine the addresses of 13 and 8 being passed into the function, 13 and 8 will be subtracted leaving 5. This value indicates to **qsort** that the first object is greater than the second. Two equal integers passed in will give a value of 0.

The `compName` function compares two structures. Our criteria for comparison is the name of the person. We ignore the id, address and postcode information. Remember here that the Standard says that if two elements compare as equal, their order in the sorted array is unspecified. Thus if there are three John Smiths in the array, they will occur in a random order within the sorted array. That is to say the Smarts and Smedleys will occur before them and the Smocks and Smythes will occur after. The three Smiths themselves will occur in any order within their three slots.

To avoid this randomness other fields could be examined in the event of a "tie":

```
int comp_name(const void * va, const void * vb)
{
    int i;
    const person * pa = (const person *)va;
    const person * pb = (const person *)vb;

    i = strcmp(pa->name, pb->name);
    if (i != 0)
        return i;
    i = strcmp(pa->town, pb->town);
    if (i != 0)
        return i;
    i = strcmp(pa->postcode, pb->postcode);
    if (i != 0)
        return i;
    return pa->id - pb->id;
}
```

Calling qsort

- Once the comparison function is written, the call to qsort is straightforward
- It requires a pointer to the first element in the array, the number of items in the array, the size of each element and the address of the comparison function

```
#include <stdlib.h>
int main(void)
{
    int ia[100];
    person pa[50];
    /* Initialise the two arrays... */
    qsort(ia, ASIZE(ia), sizeof(ia[0]), comp_int);
    qsort(pa, ASIZE(pa), sizeof(pa[0]), comp_name);
    return 0;
}
```



Once the comparison function is written, the rest is straightforward. `qsort` requires 4 parameters, the start address of the array to be sorted, the number of items in the array, the size in bytes of each element and the address of the comparison function.

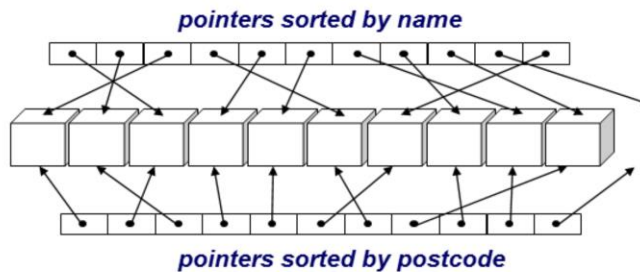
The example assumes that both arrays are completely filled. It is important NOT to sort the whole array if only half of it is filled. Otherwise you will be sorting random values into your data!!

Thus, if the array is 100 elements in size, but only 68 values are filled in, the call to `qsort` would be:

```
qsort(ia, 68, sizeof(ia[0]), comp_int);
```

Sorting pointers vs. sorting data

- **Sorting data structures poses two problems**
- **Physically moving the structures in memory will be time consuming**
- **The data must be re-sorted when the "key" changes**
- **qsort() can sort pointers or the structures themselves**
- **Several arrays of pointers may be maintained and sorted separately on different keys**



The larger a structure becomes, the larger the overhead of copying the structure around in memory. Returning to our idea of sorting arrays of Person structures, we may want the array sorted by name in one part of the application and by address in another. One possible solution would be to maintain two independently sorted arrays of Persons. However, there may be too many Person structures to maintain two copies.

Since `qsort()` will sort any data type, it can sort pointers. A pointer is a fixed size and will be orders of magnitude smaller than one of the structures. Creating two arrays of pointers will have far less overhead than creating two arrays of structures.

qsorting pointers

```
#include <stdlib.h>
#define NUM 50
int main(void)
{
    person pa[NUM];
    person * byname[NUM];
    person * bycode[NUM];
    size_t i;
    /* Assume pa initialised ... */
    for (i = 0; i < NUM; i++)
        byname[i] = bycode[i] = &pa[i];
    qsort(byname, NUM, sizeof(byname[0]), comp_person_name);
    qsort(bycode, NUM, sizeof(bycode[0]), comp_person_code);
    return 0;
}
```

```
int comp_person_name(const void * va,
                    const void * vb)
{
    const person * const * pa = va;
    const person * const * pb = vb;
    return strcmp((*pa)->name, (*pb)->name);
}
```

The example above shows how pointers may be sorted. One minor problem area is to remember to provide the size of a pointer as the 3rd parameter to `qsort()`.

Another problem is the writing of the comparison function. Although there is nothing inherently difficult in writing this function it is VITAL to remember that what is passed is NOT a pointer to a structure but a pointer to a pointer to a structure.

To understand this last point, imagine we were sorting integers. The comparison function is called with pointers to the integers. When sorting structures the comparison function is called with pointers to structures. Therefore if we are sorting pointers, the comparison function is called with a pointer to a pointer.

Searching

- The Standard provides a searching function *bsearch()*
- A binary search through a sorted array of objects
- The same comparison function is used as before
- Undefined behaviour if the array elements are not sorted

```
typedef int (*_cmpfn)(const void *, const void *);  
void * bsearch(const void * key, const void * base,  
              size_t num, size_t width, _cmpfn f);
```

- If the key (address passed as first parameter)...
- Is found, the address of the matching array element is returned
- Is not found, NULL is returned
- If two elements match the key, the element referenced is unspecified

The Standard provides a searching routine `bsearch()`. This is an implementation specific version of a binary search. Again the function is generic in that void pointers are used.

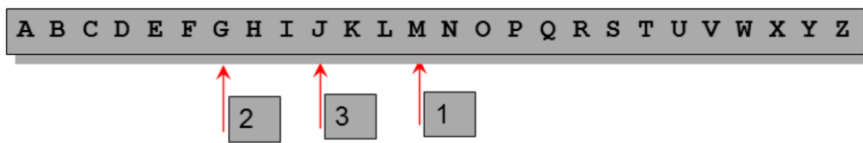
The ADDRESS of the key must be passed into the `bsearch` function as the first parameter. The remaining parameters are as for `qsort()`, including the address of the same comparison function as `qsort()` uses.

The array must be sorted. The reason for this will be seen next.

If `bsearch()` finds the key, it returns a pointer to it within the array. If it does not find the key, a NULL pointer will be returned.

How binary searching works

- **Binary searching uses a "divide and conquer" strategy**
 - The array is divided into two parts, the part the key belongs to is determined, the search is concentrated in that part
- **For instance, search for "J" in the letters of the alphabet**
 - ① Find mid point of array (M and N are likely candidates),
J is less than M so jump left by $\frac{1}{2}$ the distance (13 / 2 letters)
 - ② J is greater than G so jump right by $\frac{1}{2}$ that distance (6 / 2 letters)
 - ③ Arrive at J



The example above illustrates how binary searching works. The algorithm keeps on dividing the array in two. At each stage it has only half the amount of data to consider.

The example above illustrates an important point about the algorithm. Both "M" and "N" are likely candidates for the first comparison. An arbitrary decision must be made as to which to consider. If we chose "M" to compare and we were searching for "N", we'd do a lot of unnecessary work. Even so, we'd still do less work than in a linear search where first "A", then "B" then "C" etc. were compared.

Initially, vast swathes of the data are eliminated, then as iterations proceed it zooms in taking smaller and smaller steps each time.

Example

```
#include <stdlib.h>
int main(void)
{
    person pa[50];
    person key;
    person * p;
    /* Intialise the array of Persons */
    qsort(pa, ASIZE(pa), sizeof(pa[0]), comp_name);
    printf("Enter the name of the person to search for ");
    scanf("%49s", key.name);
    p = bsearch(&key, pa, ASIZE(pa), sizeof(pa[0]), comp_name);
    if (p) {
        printf("details are: ");
        print_person(p);
    } else
        printf("no one found by that name\n");
    return 0;
}

int comp_name(const void * va,
              const void * vb)
{
    const person * pa = va;
    const person * pb = vb;
    return strcmp(pa->name, pb->name);
}
```

There are a few things to notice about this example:

Firstly, the same comparison routine is used to both sort the array and to find the key. The routine is actually called in two subtly different ways each time. When called via `qsort()` it compares two elements WITHIN THE ARRAY. When called from `bsearch()` it compares the key with ONE element within the array. The address of the key is passed in as the first parameter, the address of the array element as the second.

Secondly, like is compared with like. Even though only the name represents the key

Problems

- **The Standard says:**

**'If two or more elements match the key,
the element returned is unspecified'**

- **When a match is returned, we have no way of knowing if this is the only match or one of a number of matches**
- **If one of a number, we do not know if this is the first or last match or somewhere in between**
- **It is worth moving the final pointer backwards and forwards to see if other elements match**

Example

```
person * p;
p = bsearch(&key, pa, ASIZE(pa), sizeof(pa[0]), comp_name);
if (p)
{
    const person * first = p;
    const person * const first_ptr = &pa[0];
    const person * last = p;
    const person * const last_ptr = &pa[ASIZE(pa)-1];
    while (first != first_ptr && comp_name(&key, first-1) == 0)
        first--;
    while (last != last_ptr && comp_name(&key, last+1) == 0)
        last++;
    if ((last - first) == 0)
        printf("there is 1 match\n");
    else
        printf("there are %u matches\n", last - first + 1);
    while (first <= last)
        print_person(first++);
}
```

Care must be taken not to overstep the bounds of the array, i.e. a check should be made when initialising `first` and `last`. For example, the following code exhibits undefined behaviour.

```
const person * first = p - 1;
while (comp_name(&key, first) == 0)
    first--;
first++;
```

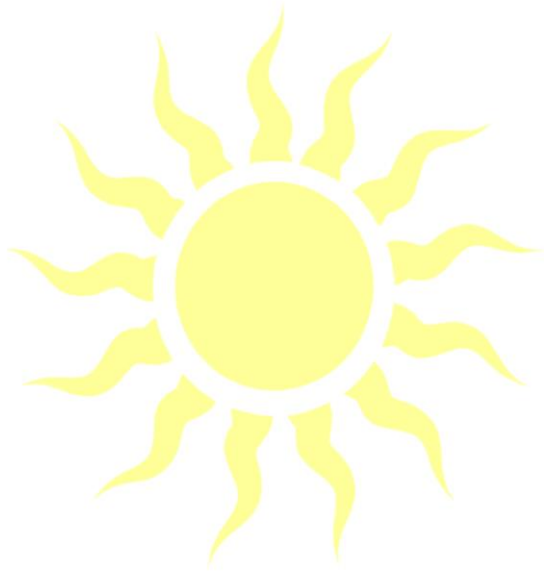
Consider when the non null pointer `p` points to the initial element of `pa`. In other words, `p` equals `&pa[0]`. This would cause `first` to point to `pa[-1]` which would then be dereferenced inside `compName`. All bets are off! Similarly, the following code also exhibits undefined behaviour.

```
const person * last = p + 1;
while (comp_name(&key, last) == 0)
    last++;
last--;
```

Consider when the non null pointer `p` points to the last element of `pa`. In other words, `p` equals `&pa[49]` assuming there are 50 elements in the array `pa`. This would cause `last` to point to `pa[50]` which would then be dereferenced inside `comp_name`. Again, all bets are off.

Summary

- **Sorting with `qsort()`**
- **Comparison functions**
- **Sorting pointers**
- **Searching with `bsearch()`**
- **Visiting all matching objects**



A number of functions in the C library are implemented as variadic functions, most notably `printf()` and `scanf()`. In this chapter, we have seen how to write variadic functions from scratch, and how to integrate them with existing variadic functions if so desired.

The standard library also provides a great many functions for determining the current date and time, as well as the processor time elapsed during the current process. There are also a number of non-standard functions available with most compilers for modifying the current time.

Random number generation is useful in simulations and arcade games, and is achieved using the standard functions `rand()` and `srand()`.

The standard library also provides two extremely useful and flexible functions for searching and sorting an array: `bsearch()` and `qsort()` respectively. It makes a great deal of sense to use these functions in your applications rather than trying to write your own, since these functions are tried and tested and are implemented as efficiently as possible by the compiler vendors.