Programming with C

Terry Marris December 2010

16 Searching and Sorting

In the previous chapter we looked at arrays of *numbers*. Now we see how to search through an array and how to sort the contents of an array into order.

16.1 Linear Search

There are two possible results from a search: either you find the item you are looking for, or you don't.

We start with a simple array of integers:

```
int array[] = { 19, 10, 18, 11, 17, 12, 16, 13, 15, 14 };
```

We search through the array item by item until either we find what we are looking for, or we reach the end of the array and conclude the item is not there.

```
/* linearsearch.c: searches an integer array */

#include <stdio.h>

/* linearSearch: returns index of if target in array if found,
   otherwise return -1 */
int linearSearch(int array[], int size, int target)
{
  int i;

  for (i = 0; i < size; i++)
    if (array[i] == target)
      return i;
  return -1;
}

int main()
{
  enum { size = 10 };
  int array[size] = { 19, 10, 18, 11, 17, 12, 16, 13, 15, 14 };

  printf("Expect found: ");
  linearSearch(array, size, 19) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect found: ");
  linearSearch(array, size, 14) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect not found: ");
  linearSearch(array, size, 20) < 0?
      printf("not found\n") : printf("found\n");

  return 0;
}
```

```
Command Prompt
$ gcc linearsearch.c -ansi -Wall -pedantic -o linearsearch.exe

$ linearsearch
Expect found: found
Expect found: found
Expect not found: not found

$
```

16.2 Selection Sort

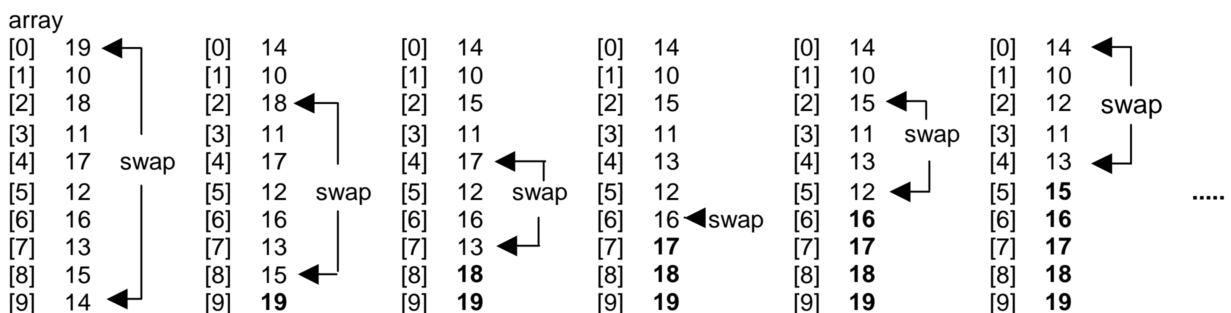
Sorting involves re-arranging the contents of a collection into some kind of order, for example into ascending numerical order. For example, starting with an array such as

```
int array[] = { 19, 10, 18, 11, 17, 12, 16, 13, 15, 14 };
```

we seek to end up with

```
int array[] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
```

The essence of the selection sort is this. We look for the largest item in the array and place it at the end. Then we consider the array without its last element: we look for the largest item in this sub-array and place it at its end. Then we consider this array without its last element: we look for the largest item in this sub-array and place it at its end. We repeat this process until the sub-array has just one element.



As you can see, the size of the sorted portion shown in bold increases from the bottom for each swap made.

Our selection sort has two helper functions. *swap()* exchanges the value at the given indices in the array.

```
/* swap: exchanges the values at index i and j in array */
int swap(int array[], int i, int j)
{
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
  return 0;
}
```

indexOfLargest() looks for the location of the largest value in an ever-decreasing part of the array.

```
/* indexOfLargest: returns index of largest value in
   array[0]..array[lastInded] */
int indexOfLargest(const int array[], int lastIndex)
{
  int index = 1;
  int i;

  for (i = 0; i <= lastIndex; i++)
    if (array[i] > array[index])
      index = i;
  return index;
}
```

We look through the given part of the array, from 0 up to *lastIndex*. If we find a value in the array that is larger than the value at *array[index]*, then *index* is updated with the new location.

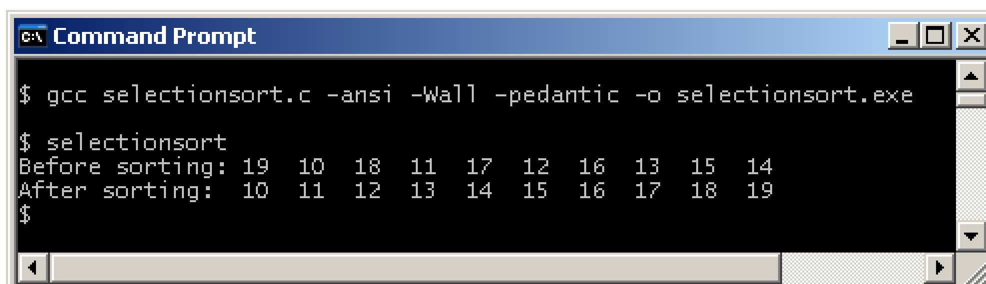
The *selectionSort()* function provides the ever-decreasing range of the array to be searched through. Here is the complete function.

```
/* selectionSort: sorts array into ascending order */
int selectionSort(int array[], int arraySize)
{
  int index;
  int i;

  for (i = arraySize - 1; i > 0; i--) {
    index = indexOfLargest(array, i);
    swap(array, i, index);
  }
  return 0;
}
```

And here is the complete program and its run.

```
/* selectionsort.c: sorts array of integers into ascending order */

#include <stdio.h>

/* indexOfLargest: returns index of largest value in
   array[0]..array[lastInded] */
int indexOfLargest(const int array[], int lastIndex)
{
  int index = 1;
  int i;

  for (i = 0; i <= lastIndex; i++)
    if (array[i] > array[index])
      index = i;
  return index;
}
```

```
/* swap: exchanges the values at index i and j in array */
int swap(int array[], int i, int j)
{
  int temp = array[i];
  array[i] = array[j];
  array[j] = temp;
  return 0;
}

/* selectionSort: sorts array into ascending order */
int selectionSort(int array[], int arraySize)
{
  int index;
  int i;

  for (i = arraySize - 1; i > 0; i--) {
    index = indexOfLargest(array, i);
    swap(array, i, index);
  }
  return 0;
}

int printArray(const int array[], int arraySize)
{
  int i;

  for (i = 0; i < arraySize; i++)
    printf("%d  ", array[i]);
  return 0;
}

int main()
{
  enum { size = 10 };
  int array[size] = { 19, 10, 18, 11, 17, 12, 16, 13, 15, 14 };

  printf("Before sorting: ");
  printArray(array, size);
  selectionSort(array, size);
  printf("\n");
  printf("After sorting:  ");
  printArray(array, size);
  return 1;
}
```



16.3 Binary Search

The essence of the binary search function is to repeatedly cut the search area in half until either the required item is found or the array cannot be divided any further.

array

0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	17	18	19	20	21

We are looking for the *target*, 16

The *middle* index is (0 + 9) / 2 = 4 (integer division)

The *target*, 16, is greater than *array[middle]*, 15. So we discard the left half from the search.

array

0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	17	18	19	20	21

The *middle* index of the search area is (5 + 9) / 2 = 7

The *target*, 16, is less than *array[middle]*, 19. So we discard the right half from the search.

array

0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	17	18	19	20	21

The *middle* index of the search area is (5 + 6) / 2 = 5

The *target*, 16, is less than *array[middle]*, 17. So we discard the right half from the search.

array

0	1	2	3	4	5	6	7	8	9
11	12	13	14	15	17	18	19	20	21

The array cannot be divided any further. So we conclude the item we are looking for is not in the array.

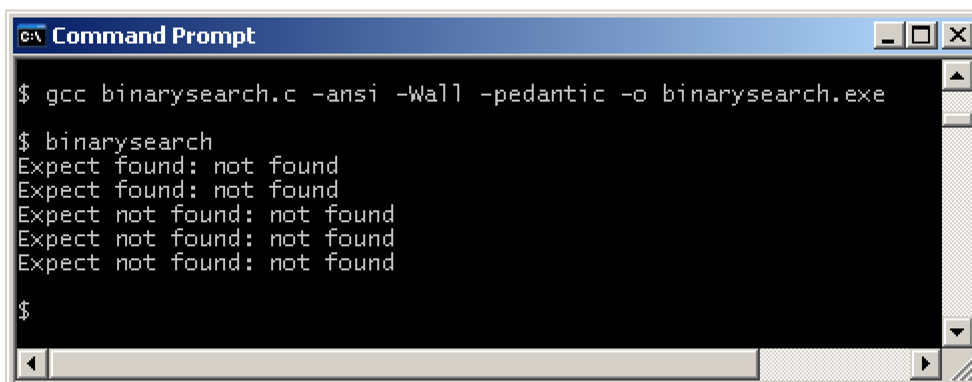
Another example. We are looking for the *target*, 15.

The *middle* index is (0 + 9) / 2 = 4 (integer division)

The *target*, 15, is neither less than *array[middle]*, 15, nor more than *array[middle]*, so it must be equal to *array[middle].* We have found our target.

Shown below is the entire program and a program run.

```
/* binarysearch.c: looks for an item in a sorted array */

#include <stdio.h>

/* binarySearch: returns index where target is found in array,
   -1 otherwise.  Array must be sorted */
int binarySearch(int array[], int arraySize, int target)
{
  int low, high, middle;

  low = 0;
  high = arraySize - 1;
  while (low <= high) {
    middle = (low + high) / 2;
    if (target < array[middle])
      high = middle - 1;
    else if (target > array[middle])
      low = middle + 1;
    else
      return middle;  /* target == array[middle] */
  }
  return -1;
}

int main()
{
  enum { size = 10 };
  int array[size] = { 11, 12, 13, 14, 15, 17, 18, 19, 20, 21 };

  printf("Expect found: ");
  binarySearch(array, size, 11) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect found: ");
  binarySearch(array, size, 21) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect not found: ");
  binarySearch(array, size, 16) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect not found: ");
  binarySearch(array, size, 10) < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect not found: ");
  binarySearch(array, size, 22) < 0?
      printf("not found\n") : printf("found\n");

  return 0;
}
```

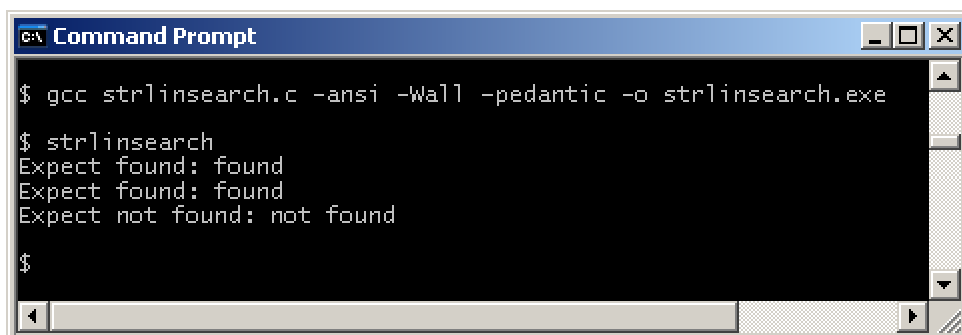


It is possible for *(low + high)* to result in overflow, i.e. an integer that is too large to be stored in memory. We acknowledge the possibility and move on.

16.4 String Linear Search

Seaching through an array of strings is similar to searching through an array of integers.

```
/* strlinsearch.c: searches an array of strings */

#include <stdio.h>
#include <string.h>

/* strLinearSearch: returns index if target is in array,
   -1 otherwise */
int strLinearSearch(char *array[], int arraySize, char* target)
{
  int i;

  for (i = 0; i < arraySize; i++)
    if (strcmp(array[i], target) == 0)
      return i;
  return -1;
}

int main()
{
  enum { size = 10 };
  char *array[size] = { "tom", "max", "jim", "ann", "may",
                        "sue", "rob", "tim", "kim", "sam" };

  printf("Expect found: ");
  strLinearSearch(array, size, "tom") < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect found: ");
  strLinearSearch(array, size, "sam") < 0?
      printf("not found\n") : printf("found\n");
  printf("Expect not found: ");
  strLinearSearch(array, size, "seb") < 0?
      printf("not found\n") : printf("found\n");

  return 0;
}
```



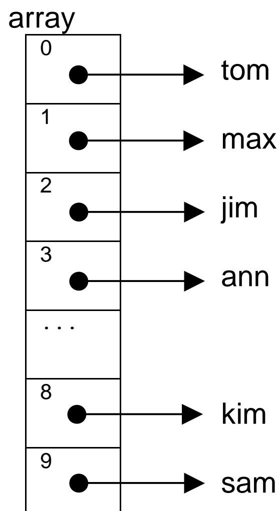
The declaration

```
char *array[size];
```

says each element of the array is a pointer to *char*.

```
char *array[size] = { "tom", "max", "jim", "ann", "may",
                      "sue", "rob", "tim", "kim", "sam" };
```

initialises each element of the array with a string value. Since it is part of the variable declaration, space is automatically set aside for each element value. We do not need to use *maloc()*.



We use *strcmp()* to compare two strings for equality:

```
/* strLinearSearch: returns index if target is in array,
   -1 otherwise */
int strLinearSearch(char *array[], int arraySize, char* target)
{
  int i;

  for (i = 0; i < arraySize; i++)
    if (strcmp(array[i], target) == 0)
      return i;
  return -1;
}
```

Exercise 16.1

1. Implement and test a sorting function on an array of string values

2. Implement and test a binary search function on an array of sorted string values.

We have looked at searching and sorting techniques. **Next** we look at structures in C.

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988
Mark Williams Company *ANSI C A Lexical Guide* Prentice Hall 1988
Stubbs D and Webre N *Data Structures with Abstract Data Types and Modula-2* Thomson 1987