Laboratory 13  Structures and Yet More Arrays

13.1 Objectives

- to understand the difference between binary and sequential search
- to be able to program using structures

13.2 Prerequisites

You should understand arrays and for loops in order to do this lab.

13.3 Correspondence to Text

The material in this lab best corresponds to sections 9.5, 9.7, and 9.8 in the text.

13.4 Introduction

We have studied the array, and you have done relatively complex programming with arrays. Arrays allow us to store many different objects together, but these objects all need to be the same type.

What if we want to have a collection of objects, but the objects are different types? For example, if we want to store information about an employee for payroll, we probably would want the employee’s name, address, SSN, salary, tax withholding, state of residence, and possibly other information. It would be nice if we could store this all as one chunk of data, and that is possible with structures.

13.5 Array Review

13.5.1 Searching an array

Suppose that we have an array of integers in an array called labstudents. The integers represent the student-id. For example, if there were 8 students in labstudents with the ids 34500, 23487, 13546, 55490, 28934, 88224, 74567, and 97345, the array would representing this would be (assuming that we keep the student ids in a sorted order):

<table>
<thead>
<tr>
<th>13546</th>
<th>23487</th>
<th>28934</th>
<th>34500</th>
<th>55490</th>
<th>74567</th>
<th>88224</th>
<th>97345</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If I wanted to know if a particular person were in the array, I could write a function to look at each element, and return true if the student id is found and false if the student id is not found. But because the array is sorted, I don’t even need to look at the whole array. I can do a binary search. A binary search is similar to what you might do if you are looking up a word in the dictionary. You might open the dictionary to the middle, look at the guide words, then go right (towards z) or left (towards a) depending on how your word compares with the guide words.
In this array, I can start at the middle part of the array, and look back and forth depending on how the element that I’m looking for compares to the element in the array. A function that performed a binary search would return true if the student id that I’m looking for matches the middle element (the one that I’m looking at now). Otherwise, the function would look at the left part of the array if the student id were less and the right part of the array if the student id were more than the element currently being examined (i.e., the “guide number” to use the dictionary analogy). The function would continue looking at the middle element of the array portion being examined until we found the element or there was no more of the array to search. If there were no more array to search, the function would return false (i.e., not found).

For example, if I looked for student element 30000, I would look at the middle element (index 3). The element stored in index 3 is 34500. 34500 > 30000, so I look to the left in the array. I know that 30000 will not be in slots 4-7 because 30000 < 34500, and the array is sorted. It’s like looking a name up in the telephone book or a word up in the dictionary. If the word in the book is George, and you’re looking for Adams, you know you need to go to the left (to earlier in the alphabet). If Adams is in the book at all, it will be before George not after George.

So, I search the array from 0-2, knowing that if 30000 is in the array, it will be in one of these elements (we’ve eliminated everything in slots 3-7). In a binary search, the middle element is chosen for comparison. In this case, the middle is index 1 (between 0 and 2) which holds 23487. So, I compare 23487 to 30000. 23487 < 30000, so I look to the right since if 30000 is there, it will be to the right of 23487.

Finally, I search the array from 2 to 2 (since we’ve eliminated 3-7 and 0-1), and do the comparison. 28934 ≠ 30000, and there is no place else to go since I’ve looked everywhere else, so I know that I’m done looking. Thus, the search consisted of looking at 3 elements: in slot 3 (34500), slot 1 (23487), and slot 2 (28934). This is a lot fewer checks than if we’d looked at every element (which would have been 8 checks).

- Do lab 13 questions Binary Search.
- Copy binsearch.cpp from the public directory.
- This program does a binary search using the student ids given above.
- Copy infile13.
- Compile and test the program with 30000 and with 97345. The program prints which element is currently being examined so that you can see how the search is getting closer. Try it with several numbers.
- Modify binsearch.cpp to include a function writearray that prints out all of the elements in the sorted array. Call this function after the searching is complete (i.e., after the if statement in the main function). I have written the cout statements for you in a comment, but I have not written the prototype, call, or definition.
- What if the file were not sorted? You would have to check every element. Copy binsearch.cpp to search.cpp, and modify the function search so that it looks at every element starting from 0. No other function should be modified. Do not change any of the cout statements.
Remember this function does two things: 1) it finds the next guide words, and puts them in begin and end, and 2) it returns 1 if the element is in the array and equal to middle and 0 otherwise (it only looks at one element, not the entire array).

For the sequential search program (the one that you’re writing now), middle will not be the middle of the array; it will be the first element that you have not searched (0 the first time the array is called, then 1, then 2, etc. until either the element is found or the end of the array is reached).

- Compile and test the file.

### 13.6 Structures

You have learned one mechanism that allows you to store multiple values in one data structure; now we’ll learn another: the structure.

Arrays require that all of the elements be of the same type; structures allow you to store different types in the same data structure. For example, if your school wanted to keep track of you, they might create a C++ data structure called student. The data structure would have your initials (they might have your name, but that’s a little more complicated), your SSN, your GPA, and the total number of credits that you have successfully completed.

That structure might look like:

```c
struct student
{
    char first_initial;
    char last_initial;
    int SSN;
    float GPA;
    float credits;
};
```

Unlike the way we’ve been declaring arrays, this structure declaration is not a variable; it is a type. It occupies no space; we can’t access any part of `student` without declaring a variable of that type. Let’s do that:

```c
student labstudent;
```

Now we have a variable, `labstudent`, that has the structure given above. It has a first_initial, a last_initial, a SSN, a GPA, and a credits part. Each of these parts is called a member of the structure. In order to access the GPA for labstudent, you would write:

```c
labstudent.GPA
```

The other fields would be accessed in a similar way.

Structures can be used in pretty much exactly the same way as arrays, but there are some differences. An entire structure can be passed as an argument, but unlike arrays, structures can be passed either by value or by reference, so you must specify an & when you are passing a structure by reference (remember that arrays are passed by reference only). Like array elements,
individual structure members may also be passed by value or reference. Structures may not be printed using `cin` and `cout` statements; individual members must be accessed.

- Answer questions **Structures** in lab 13.

### 13.6.1 Hierarchical Structures

You can use any type inside of a structure including other structures and arrays. For example, the type `student` might be:

```c
struct name
{
    char firstname[8];
    char lastname[8];
};

struct SSNform
    // I have made SSN chars because they are treated that way;
    // they are not treated as numbers
{
    char first3[3];
    char second2[2];
    char third4[4];
};

struct student
{
    name studentname;
    SSNform SSN;
    float GPA;
    float credits;
};
```

I have changed the name to be an array of characters to make the example more realistic. The variable declaration, `student labstudent`, is the same as before. To assign 1 to the first character (number) of labstudent’s SSN, you would write:

```c
labstudent.SSN.first3[0] = '1';
```

All of the types within a structure must have already been declared which is why `SSNform` and `name` were declared before `student`.

In general, to access a hierarchical structure (so called because it has other structures inside it), you write: `varname.outermostname.innername` (though structures can be more deeply embedded than this).

You can see how this would be useful for a `student` structure because you could have structures about other courses, about bills, etc. that are used in the `student` structure as well as other structures.
13.7 Synthesis

Write a program that reads in information for and writes out information for a variable of type `student`.

- Copy my program `student.cpp`.
- Write the functions that are not written: `readSSN`, `readGPA`, `writeSSN`, and `writeGPA`. Also, write the function calls for these functions. The function headers have been written.
- Save, compile, and test the program.