CSI 1430
Week 14 Resource

Colin Burdine

4/18/21

Major Topics:

1. Pointers

Keywords: *pointers, indirection, references*

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Pointers

One of the most powerful tools that imperative programming languages (like C++) provide is the ability to perform *indirection*. Indirection is performed when the location of a data type (such as an integer, double, or even an object) in logical memory can be stored and manipulated as a data type itself. By storing the address of a type in physical memory, we are able to efficiently look up a value in, say, and array or some other data structure that we have created in our program. In fact, as you may learn in future computer science courses, pointers provide the backbone of countless data structures that are commonly used in programs. Effectively, pointers allow us to "hop" from one address to the next in memory.

In C++, we can extract the address of any object by using the `&` operator (which we recall also indicates that a variable is passed by reference in a function signature). We can store the address of a particular data type using the corresponding data type pointer (signified

by a * after the type name). We can access the value stored in a pointer by putting a * before the name of the pointer. An example of this is given below:

```
1
2  // declare an integer:
3  int someValue = 42;
4
5  // initialize a pointer and set it to hold the address of someValue:
6  int* valuePtr = &someValue;
7
8  // print the address of someValue using the pointer:
9  cout << "The ADDRESS of someValue in logical memory is: "
10      << valuePtr << endl;
11
12 // print the value of someValue using the pointer:
13 cout << "The VALUE of someValue in logical memory is: "
14      << *valuePtr << endl;
```

One possible output for this program is:

```
The ADDRESS of someValue in logical memory is: 0x7ffc6be9fd3c
The VALUE of someValue in logical memory is: 42
```

Note that the value `0x7ffc6be9fd3c` is printed when pass the address stored in `valuePtr` to cout. This value is a *hexidecimal* representation of the address of `someValue`. Hexidecimal is a base-16 number system that is similar to the base-10 number system we are familiar with., but is more compact and translates much easier from binary and is a common way of representing computer memory addresses.

We can see that using indirection gives us quite a bit of power to be able to look up almost any value in logical memory, but with this power comes responsibility; we need to be careful that we do not attempt to look up (or worse, modify) random values in memory. Consider the following program:

```
1
2  // initialize a pointer (to hold the address of an integer x)
3  int x = 42;
4  int* memPtr = x;
5
6  // attempt
7  while(true){
8      cout << "Clearing the value stored at: "
9           << memptr << endl;
10
11     // clear the four bytes referenced by memPtr
```

```
12      *memPtr = 0;
13
14      // move on to the next four bytes:
15      memPtr += 1;
16  }
```

One possible output of the program is:

```
Clearing the value stored at: 0x7ffef63187fc
Clearing the value stored at: 0x7ffef6318800
Clearing the value stored at: 0x7ffe00000004
Segmentation fault (core dumped)
```

There are several things to observe here. First, note that on line 15 we are incrementing the value of a pointer. Since an address in memory is essentially an integer value, we can perform numerical operations on it such as addition and subtraction, incrementing and decrementing. In particular, because we are incrementing a pointer to an `int` type, the memory address is incremented by four every time that the pointer is incremented by one. This gives us the general pointer formula:

```
int(&(ptr + x)) == int(&ptr) + x*sizeof(ptrType)
```

Second, we note that even though our program should run in an infinite loop, it somehow terminated with the ominous message `segmentation fault (core dumped)`. This abrupt termination of the program is done by the operating system, which has detected a request to read a logical memory value outside the range of allocated logical memory values, called the program *segment*. We were able to get away with the first two reads past the integer value we allocated, however the OS stopped us when we stepped outside the segment boundaries. It is worth noting that even though a program may attempt to write outside its allocated segment, the OS will prevent one program writing to the memory space of another, so you should not be concerned about trying to run this program on your own machine. (This is why we are referring to addresses stored in pointers as *logical addresses* as opposed to *physical addresses*. Logical addresses live in an abstract memory space presented to the program by the OS, while *physical addresses* are the actual hardware addresses of data stored in your computer's random access memory (RAM).

2 Programming Example Problem

To illustrate how we can use pointers in C++, consider the following example programming problem:

Write a program that implements and tests the following two functions:

```
// returns a pointer to the first instance of 'value':
int* findFirst(int (&arr)[], int arrLen, int value);

// returns a pointer to the last instance of 'value':
int* findLast(int (&arr)[], int arrLen, int value);
```

The first function should take an unsorted array (passed by reference) and return a pointer to the last occurrence of an integer value in a given array with the indicated size. The second function should do the same, but instead return a pointer to the first occurrence of the value in the (unsorted) array. If the value cannot be found in the array, then simply return the C++ value `nullptr`, which indicates that the address is not found.

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week.

3 Closing Remarks

I hope that the resources I shared with you this semester are helpful and efficient in addressing your academic needs. As we approach the end of the spring semester, I would like to let you know that all resources for the course content this semester can be found here: https://www.baylor.edu/support_programs/index.php?id=967950
I would highly encourage you to take a look at these resource as you begin making preparations to study for final exams.

```cpp
#include <iostream>
using namespace std;

// returns a pointer to the first instance of 'value':
int* findFirst(int (&arr)[], int arrLen, int value);

// returns a pointer to the last instance of 'value':
int* findLast(int (&arr)[], int arrLen, int value);

int main(){
    const int ARRAY_LEN = 10;
    int ARRAY[ARRAY_LEN] = { 0, 2, 3, 1, -7, 3, 4, 5 -2, 100 };

    // search for values:
    int* first = findFirst(ARRAY, ARRAY_LEN, 3);
    int* last = findLast(ARRAY, ARRAY_LEN, 3);
    int* notFound = findFirst(ARRAY, ARRAY_LEN, 42);

    // print results:
    cout << "Address of first '3': " << first << endl;
    cout << "Address of last '3': " << last << endl;
    cout << "Address of '42' (should be 0): " << notFound << endl;
    return 0;
}

int* findFirst(int (&arr)[], int arrLen, int value){
    int* p = &(arr[0]);
    int* end = p + arrLen;
    bool found = false;
    while(!found && p != end){
        found = (*p == value);
        p++;
    }

    return found? p : nullptr;
}

int* findLast(int (&arr)[], int arrLen, int value){
    int* end = &(arr[0])-1;
    int* p = end + arrLen;
    bool found = false;
    while(!found && p != end){
        found = (*p == value);
        p--;
    }

    return found? p : nullptr;
}
```