

CSC 209 Review 5 Solution

August 22, 2020

1 Exercises

1. a) 14
b) 34
c) 4
d) true
e) false

Notes

- **Pointer Arithmetic**

- Adding an integer to a pointer

Example



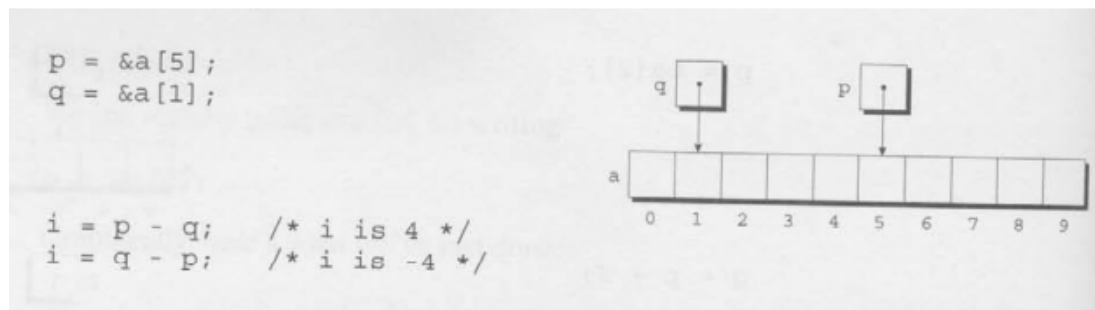
- Subtracting an integer from a pointer

Example



- Subtracting one pointer from another

Example



• Comparing pointers

- Can compare pointers using relational operators (i.e. `<`, `<=`, `>`, `>=`) and the equality operators (i.e. `==`, `!=`)
- Returns 1 if **true** and 0 if **false**

Example

```
p = &a[5];
q = &a[1];
```

`p <= q` is 0 and `p >= q` is 1

2. `low` and `high` are memory addresses.

So, `low + high` is out of bound, and it could potentially point to an undesirable or wrong value.

To fix this, we subtract the from high value to the low value:

$$\text{middle} = \frac{\text{low} + \text{high}}{2} \quad (1)$$

3. I need to write the contents of an array `a` after the execution of statements outlined in problem sheet.

After execution, the array would have contents of `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`.

Notes

- **Combining the `*` and `++` Operators**

- `*p++` or `*p++` → Value of expression is `*p` before increment; increment `p` later
- `(*p)++` → Value of expression is `*p` before increment; increment `*p` later
- `++*p` or `*(++p)` → Increment `p` first; value of expression is `*p` after increment
- `+++p` or `++(*p)` → Increment `*p` first; value of expression is `*p` after increment

Example

`a[i++] = j`

Means assign the value `j` to `a[i]` before increment

Example 2

```
for (p = &a[0]; p < &a[N]; p++)  
    sum += *p;
```

Is the same as

```
p = &a[0];  
while (p < &a[N])  
    sum += *p++;
```

4. I need to re-write prototype `make_empty`, `is_empty` and `is_full` of the following code to use the pointer variable `top_ptr` instead of the integer variable `top`.

```
1  #include <stdbool.h>
2
3  #define STACK_SIZE 100
4
5  /*external variables*/
6  int contents[STACK_SIZE]
7  int top = 0;
8
9  void make_empty(void) {
10     top = 0;
11 }
12
13 bool is_empty(void) {
14     return top == 0;
15 }
16
17 bool is_full(void) {
18     return top == STACK_SIZE;
19 }
```

And after re-write using `top_ptr` instead of `top` have:

```
1  #include <stdbool.h>
2
3  #define STACK_SIZE 100
4
5  /*external variables*/
6  int contents[STACK_SIZE]
7  int *top_ptr = &contents[0];
8
9  void make_empty(void) {
10     top_ptr = &contents[0];
11 }
12
13 bool is_empty(void) {
14     return top_ptr == &contents[0];
15 }
16
17 bool is_full(void) {
18     return top_ptr == &contents[STACK_SIZE-1];
19 }
```

5. First, I need to identify which of the following expressions are illegal because of mismatched types.

- a) `p == a[0]`
- b) `p == &a[0]`

- c) `*p == a[0]`
- d) `p[0] == a[0]`

Here, only a) is illegal.

Second, I need to write which of the remaining expressions are true.

Here, the expressions that return true are b), c) and d).

Notes

- `*(a+i)` is equal to `a[i]`
- `*p` and `a[]` are the same given `p == a`
- **Using an Array Name as a Pointer**
 - The name of an array can be used as a pointer to the first element in the array.

Example

```
int a[10];

*a = 7; /* stores 7 in a[0] */

*(a+1) = 12; /* stores 7 in a[1] */
```

Example 2



```
To simplify the loop, we can replace &a[0] by a and &a[N] by a + N:

for (p = a; p < a + N; p++)
    sum += *p;
```

6. I need to re-write the following to use pointer arithmetic instead of array subscripting, and I need to make as few change as possible.

```
1  int sum_array(const int a[], int n) {
2      int i, sum;
3
4      sum = 0;
5
6      for (i = 0; i < n; i++)
7          sum += a[i];
8      return sum;
9  }
```

After making changes to above code to use pointer arithmetic, we have

```
1  int sum_array(const int a[], int n) {
2      int i, sum;
3
4      sum = 0;
5
6      for (i = 0; i < n; i++)
7          sum += *(a+i);
8      return sum;
9  }
```

7. I need to write the following using pointer arithmetic so it finds an element in **a** that matches to value **key**. I need to return true if there is a match.

```
bool search(const int a[], int n, int key);
```

And the solution is:

```
1  bool search(const int a[], int n, int key) {
2
3      for (int i = 0; i < n; i++) {
4          if (*(a+i) == key) {
5              return true
6          }
7      }
8
9      return false;
10 }
```

8. Here, I need to re-write the following function to use pointer arithmetic instead of array subscripting.

```
1  void store_zeros(const int a[], int n) {
2      int i;
3
4      for (i = 0; i < n; i++) {
5          a[i] = 0;
6      }
7  }
```

After re-writing above code, we have

```
1  void store_zeros(const int a[], int n) {
2      int *p;
3
4      for (p = a; p < a + n; p++) {
5          *p = 0;
6      }
7  }
```

9. Here, I need to write the function

```
double inner_product(const double *a, const double *b, int n)
```

using pointer arithmetic such that it returns $a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + \dots + a[n-1] * b[n-1]$.

The solution is provided below

```
1  double inner_product(const double *a, const double *b, int n) {
2      double sum = 0, p*;
3
4      p = a;
5
6      while (p < a + n) {
7          sum += *a * *b;
8
9          a++;
10         b++;
11         p++;
12     }
13
14     return sum;
15 }
```

Correct Solution

```
1  double inner_product(const double *a, const double *b, int n) {
2      double sum = 0, *p;
3
4      p = a;
5
6      while (p++ < a + n) {
7          sum += *a++ * *b++;
8      }
9
10     return sum;
11 }
```

10. Here, I need to rewrite the function `find_middle` so that it uses pointer arithmetic - not subscripting - to visit any element.

The solution to this exercise is provided below.

```
1  int *find_middle(int a[], int n) {
2      return a + (n/2)
3  }
```

11. Here I need to modify the function `find_largest` function so that it uses pointer arithmetic - not subscripting - to visit array elements

```
1  int *find_largest(int a[], int n) {
2      int *p, *max;
3
4      max = a;
5
6      for (p = a; p < a + n; p++){
7          if (*p > *max) {
8              max = p;
9          }
10     }
11
12     return *max;
13 }
```

12. I need to write the function

```
void find_two_largest(const int *a, int n, int *largest, int *second_largest)
```

using pointer arithmetic.

The solution to this exercise is:

```
1  #include <stdbool.h> // bool
2  #include <limits.h>  // INT_MIN
3
4  bool is_largest(int current_max, int val);
5
6  void find_two_largest (int a[], int n, int *largest, int*
second_largest) {
7      int current_max = INT_MIN;
8      int current_second_max = INT_MIN;
9
10     for (int i = 0; i < n; i++) {
11         if (is_largest(current_max, a[i])) {
12             current_second_max = current_max;
13             current_max = *(a + i);
14         }
15     }
16
17     *largest = current_max;
18     *second_largest = current_second_max;
19 }
20
21 bool is_largest(int current_max, int val) {
22     if (val > current_max) {
23         return true;
24     }
25 }
```



```

26     return false;
27 }

```

Correct Solution:

```

1  void find_two_largest(const int *a, int n, int *largest, int *
2  second_largest) {
3
4      const int *p = a;
5      *largest = *second_largest = *a;
6
7      while (p++ < a + n) {
8          if (*p > *largest) {
9              *second_largest = *largest;
10             *largest = *p;
11         } else if (*p > *second_largest)
12             *second_largest = *p;
13     }
14 }

```

13. I need to rewrite the following program such that it uses a single pointer through the array one element at a time.

```

1  #define N 10
2
3  double ident[N][N];
4  int row, col;
5
6  for (row = 0; row < N; row++) {
7      for (col = 0; col < N; col++) {
8          if (row == col) {
9              ident[row][col] = 1.0;
10         } else {
11             ident[row][col] = 0.0;
12         }
13     }
14 }

```

The solution to this exercise is:

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  #define N 10
5
6  bool is_row_eq_col(int i, int n);
7
8  int main() {

```

```
9
10     int i = 0;
11
12     double ident[N][N], *p;
13
14
15     for (p = &ident[0][0]; p < &ident[N-1][N-1]; p++) {
16         if (is_row_eq_col(i, N)) {
17             *p = 1.0;
18         } else {
19             *p = 0.0;
20         }
21         i++;
22     }
23
24
25     return 0;
26 }
27
28 bool is_row_eq_col(int i, int n) {
29     if (i % (n+1) != 0) {
30         return false;
31     }
32
33     return true;
34 }
```

Notes

- Learned that the memory address of each row in two dimensional array are placed right next to each other

14. I need to write a statement that uses the `search` function from Exercise 7 to search the entire `temperatures` array for the value 32.

The solution to this problem is:

```
1     int row = 0;
2
3     for (int row = 0; row < 7; row++) {
4         if (search(temperatures[row], 24, 32)) {
5             return true
6         }
7     }
8
9     return false;
```

15. Create a loop that prints all temperature readings in row `i` of the following array using a pointer to visit each element of the row.

```
int temperatures[7][24]
```

The solution to this problem is:

```
1  int *p;
2
3  for (p = &a[i]; p < &a[i] + 24; p++) {
4      printf("%d", *p);
5  }
```

Correct Solution:

```
1  int *p;
2
3  for (p = &temperatures[i]; p < &temperatures[i] + 24; p++) {
4      printf("%d", *p);
5  }
```

16. I need to write a loop that prints the highest temperature in the `temperatures` array for each day of the week using `find_largest` function.

The solution to this problem is:

```
1  int *p, *val;
2
3  for (p = &temperatures[i]; p < &temperatures[i] + 24; p++) {
4      val = find_largest(p, 24);
5      printf("%d\n", *val);
6  }
```

17. I need to rewrite the following function to use pointer arithmetic and single loop, instead of array subscripting.

```
1  int sum_two_dimensional_array(const int a[][LEN], int n) {
2      int i, j, sum = 0;
3
4      for (i = 0; i < n; i++) {
5          for (j = 0; j < LEN; j++) {
6              sum += a[i][j];
7          }
8      }
9
10     return sum;
11 }
```

The solution to this problem is:

```
1      int sum_two_dimensional_array(const int a[][LEN], int n) {
2          int *p, sum = 0;
3
4          for (p = &a[0][0]; p < &a[n-1][LEN-1]; p++) {
5              sum += *p;
6          }
7
8          return sum;
9      }
10
```

18. I need to rewrite the `evaluation_position` function described in exercise 13 of chapter 9 using pointer arithmetic and single loop.

The solution to this problem is:

```
1      int evaluate_position(char board[8][8]) {
2
3          int white = 0, black = 0;
4          char *p;
5
6          for (p = &board[0][0]; p < &board[7][7]; p++) {
7              switch(*p) {
8                  case 'Q':
9                      white += 9;
10                     break;
11                 case 'q':
12                     black += 9;
13                     break;
14                 case 'R':
15                     white += 5;
16                     break;
17                 case 'r':
18                     black += 5;
19                     break;
20                 case 'B':
21                     white += 3;
22                     break;
23                 case 'b':
24                     black += 3;
25                     break;
26                 case 'N':
27                     white += 3;
28                     break;
29                 case 'n':
30                     black += 3;
31                     break;
32                 case 'P':
33                     white++;
34                     break;
35                 case 'p':
36                     black++;
37                     break;
38             }
39         }
40     }
```

```
38         default:
39             break;
40     }
41 }
42
43 return white - black;
44 }
```

References

- 1) Github (William Gherman), c-solutions (Chapter 9 Exercise 13), link

2 Programming Projects

1. a) I need to write a program that reads a message, then prints the reversal of the message.

The additional requirement for this program are:

- Read the message one character at a time using `getchar`
- Read characters should be stored in an array
- Read should be stopped when the array is full or the character read is `'\n'`

The solution to this problem is:

```
1  #include <stdio.h>
2
3  #define SIZE 10
4
5  int main(void) {
6
7      char array[SIZE], c, temp;
8      int i = 0, n;
9
10     printf("Enter a message: ");
11
12     // fill array
13     do {
14         c = getchar();
15
16         if (c == '\n') {
17             break;
18         }
19
20         array[i] = c;
21         i++;
22     } while (i < SIZE);
23
24     n = i;
```

```
25
26     // reverse characters in array
27
28     for (int j = 0; j < (n - j); j++) {
29         temp = array[j];
30         array[j] = array[n - (1 + j)];
31         array[n - (1 + j)] = temp;
32     }
33
34     printf("Reversal is: ");
35
36     for (int j = 0; j < n; j++) {
37         printf("%c", array[j]);
38     }
39
40     return 0;
41 }
```

```
b) #include <stdio.h>
2
3 #define SIZE 10
4
5 int main(void) {
6
7     char array[SIZE], c, temp, *p;
8     int i = 0, n;
9
10    p = &array[0];
11
12    printf("Enter a message: ");
13
14    // fill array
15    do {
16        c = getchar();
17
18        if (c == '\n') {
19            break;
20        }
21
22        *p++ = c;
23        i++;
24    } while (p < array + SIZE);
25
26    n = i;
27
28    // reverse characters in array
29    char *q = &array[0], *r = &array[n-1];
30
31    while (p < q) {
32        temp = *q;
33        *q++ = *r;
34        *r-- = temp;
35    }
36 }
```

```
37     printf("Reversal is: ");
38
39     while (array < array + SIZE) {
40         printf("%c", *array);
41     }
42
43     return 0;
44 }
```