Practice Quiz 0 - Answer Key

CS50 — Fall 2010

Prepared by: Doug Lloyd '09

October 11, 2010

1. Pointers are nothing more than addresses, and give us direct access to the memory located at those addresses. The "answers" being given by the character on the right are also addresses, so he is providing pointers in the computer science sense, not the typical English sense.

2. 8
   1
   8
   4
   8
   4

3. 10011000

4. Hexadecimal is another word for base 16. It is related to binary because a group of 4 binary digits can be represented in 1 hexadecimal digit. This is because $24 = 16$. Thus, for example,
   $1101_2 \equiv 13_{10} \equiv D_{16}$

5. A `do-while` loop guarantees that the body of the loop is run at least once. A `while` loop offers no such guarantee. A `do-while loop` is often more useful for user input, because you want to grab input at least once. A `while` loop is cleaner in most other cases. For instance, if you wanted to implement a "forever" loop, youd generally do:

   ```
   while(1) { }
   ```

   instead of a `do-while`. This is simply for clarity while reading.

6. ```
   int sum = 0;
   for(int i = 1; i <= 10; i++)
      sum += i;
   ```

   ```
   int sum = 0, i = 1;
   while(i <= 10)
      sum += i++;
   ```

7. ```
   int *mult_binomials(int A[2], int B[2]) {

      int *product = (int *) malloc(3 * sizeof(int));

      product[0] = A[0] * B[0];
      product[1] = (A[0] * B[1]) + (A[1] * B[0]);
      product[2] = A[1] * B[1];

      return product;
   }
   ```

8. As a matter of substance, there is no difference between returning pointer-to-`int` (`int *`) or `int []`, as arrays are really just pointers in disguise. But as a matter of syntax, the `int *` format must be used instead.

9. I am not `free`ing `product`!

10. Typecasting involves temporary viewing a variable of one type as a variable of another. This may be useful, for instance, to get around the way C divides integers. Storing the result of 10/4, each an integer, in a float would result in 2.0000. But casting one of the integers to a float, only temporarily, solves that problem, resulting in the correct answer of 2.5000.

11. ```
    gcc example.c -lcs50
    gcc -o example1 example.c -lcs50
    ```

12. It "pastes" the entire contents of `stdio.h` atop your file, so that you may use the functions contained therein. The most commonly used `stdio.h` function is `printf()`.

13. ```
    struct student {
       char *lastName;
       int idNum;
       double gpa;
       char classYear;
    };
    ```

14. ```
    struct student student1;
    student1.lastName = "Harvard";
    student1.idNum = 1636;
    student1.gpa = 3.78;
    student1.classYear = 'C';
    ```

15. ```
    struct student studArray[100];
    ```

16. ```
    struct student *studArray = (struct student *) malloc(sizeof(struct student) * records);
    ```

17. ```
    #include <stdio.h>
    #include <cs50.h>
    #include <ctype.h>
    #include <stdlib.h>
    #include "definition.h"

    int main(int argc, char *argv[]) {
       if(argc != 2)
          return 1;
       for(int i = 0, n = strlen(argv[1]); i < n; i++)
          if(!isdigit(argv[1][i]))
             return 2;
       int records = atoi(argv[1]);

       struct student *studArray = (struct student *) malloc(sizeof(struct student) * records);

       return 0;
    }
    ```

18. Header files enclosed in angle brackets describe system libraries. Header files enclosed in quotation marks are user-created and are generally located in the same directory as the C file that references them.

19. There's no error-checking...so what happens if `y` is equal to 0? We'll have a floating-point exception resulting from a division by zero!

20. 1,000 times

21. There are no `break` statements, meaning that our cases will "fall through" until they hit one! So, everyone who scored 60 or better will get a D, and everyone else will get an F.

22. This is what happens with C's integer arithmetic. The trailing decimal portion will be truncated. So, for example, 89/10, which would otherwise be 8.9, gets truncated to just 8. Multiplying this by 10 puts us up to 80. It helps here though, as it allows us to use a `switch`, instead of an `if-else` combination, as would otherwise be necessary.

23. It prints the letters A-Z. It leverages the fact that characters can be incremented by 1 and used in comparisons the same way integers can to accomplish this.

24. x^y

25. constant, logarithmic, linear, polynomial, exponential, factorial

26. ```
    bool div_by_3(int k) {
       return (k % 3) ? false : true;
    }
    ```

27. ```
    bool div_by_n(int k, int n) {
       return (k % n) ? false : true;
    }
    ```

28. ```
    for(int i = 1; i <= 10; i++)
       for(int j = 1; j <= 10; j++)
          (j == 10) ? printf("%-3d\n", i*j) : printf("%-3d ", i*j);
    ```

29. ```
    int letter_appears(char c, string word) {
       int pos = 0;
       while(word[pos] != '\0') {
          if(word[pos] == c)
             return (pos + 1);
          pos++;
       }
       return 0;
    }
    ```

30. $O(n)$

31. ```
    int gcd(int x, int y) {
       int lesser = (x < y) ? x : y;
       for(int i = lesser; i > 1; i--)
          if(!(x % i) && !(y % i))
             return i;
       return 1;
    }
    ```

32. ```
    int lcm(int a, int b) {
       return (a * b) / gcd(a, b);
    }
    ```

33. The algorithm runs in $O(n)$, where n is the length, in digits, of the lesser of the two numbers.

34.
```
void swap(int a, int b) {
   int tmp = a;
   a = b;
   b = tmp;
   return;
}
```
This is pretty useless to us though, since `a` and `b` were passed by value, not reference. One way to fix this and make the values *actually* swap would be to pass pointers to `a` and `b` instead, and dereference those pointers to actually swap the values they point to.

35. `pow()`

36. They improve readability of our code, and they also eliminate magic numbers.

37. `#define` is used to "replace" one value (e.g. `52`) with a different identifier (e.g. `DECK_SIZE`). `#include`, on the other hand, fetches the contents of the included file and "pastes" them atop your code, so that you are able to use the functions declared in the included file.

38. `strlen(a)` = 13. But it takes 14 bytes to store it (have to take into account the null terminator character!)

39.
```
for(int i = 0, n = strlen(s), i < n; i++)
   printf("%c\n", s[i]);
```

40. `i`

41. (a)
```
for(int i = 0; i < 5; i++)
      for(int j = 0; j < 3; j++)
         printf("%d ", numbers[i][j]);
```
(b)
```
for(int i = 0; i < 3; i++)
      for(int j = 0; j < 5; j++)
         printf("%d ", numbers[j][i]);
```

42.
```
bool ordered(int array[], int size) {
   for(int i = 0; i < size-1; i++)
      if(arr[i] > arr[i+1])
         return false;
   return true;
}
```

43.
```
bool str_equal(char *str1, char *str2) {
   if(strlen(str1) != strlen(str2))
      return false;
   for(int i = 0, n = strlen(str1); i < n; i++)
      if(str1[i] != str2[i])
         return false;
   return true;
}
```

44. It "counts down" the number of arguments and prints out that countdown. For example, if the program was called with `./program`, it would print `1 0`, but if it were called `./program a b c d e`, it would print `6 5 4 3 2 1 0`. Note that `main()` was employed recursively here!

45.
```
int rollDie(int n) {
   return (randomInt() % n) + 1;
}
```

46. The dot operator allows you to access the fields of a `struct` that you actually have (e.g. one declared statically), whereas the arrow operator allows you to simultaneously dereference the `struct` and access the fields of that `struct` when you only have a pointer to it. Note that `a->b` is equivalent to `(*a).b`.

47. Recall that `string` is just CS50's alias for a `char *`. A `char *` of course is a pointer to a character (or, in this case, an array of characters). A pointer is just an address. So when we assign one pointer's value to another pointer's value, both pointers have the same address. Ergo, they are pointing to the exact same thing. So, any change made to `input` will also show up as a change to `input_copy`, because they are pointing to the same location in memory. To achieve the effect likely desired here, one would have to use `strcpy()` or its more secure cousin `strncpy()`, to obtain two different pointers to two different addresses (and therefore two different strings).

48. It will print `3`, the value stored in `k`.

49. The heap.

50. `void *` is a generic pointer type that allows our functions to be generally applicable, instead of being focused specifically to one particular type. For example, `malloc()` returns a `void *`. Why? So that its return type can match what is needed. Imagine if instead we had to have separate functions to `malloc()` a `char`, an `int`, a `double`, a `float`, a `struct` that we wrote? It would be cumbersome. Using `void *` allows us the flexibility to use a generic pointer type that we can later cast to something more useful.

51. A `double` can hold a floating-point value with twice the level of precision as a `float`. Consequently, it also takes up 8 bytes of memory, as opposed to a `float`'s 4.

52. The table fills in as follows:

Code	a	b	c	pa	pb	pc
`b = a + c;`	3	9	6	$0x4$	$0x8$	$0xC$
`a /= c;`	0	5	6	$0x4$	$0x8$	$0xC$
`*pc = *pa * a;`	3	5	9	$0x4$	$0x8$	$0xC$
`pb = *pb;`	3	5	6	$0x4$	$0x5$	$0xC$
`*pc *= *pb;`	3	5	30	$0x4$	$0x8$	$0xC$
`a = *pb * *pc;`	30	5	6	$0x4$	$0x8$	$0xC$
`pc = &*pa;`	3	5	6	$0x4$	$0x8$	$0x4$

53. It is termed a memory leak. We can also err by double `free`ing or `free`ing memory that was never `malloc`ed in the first place.

54.
```
int sumOddDigs(int x) {
   if(x == 0)
      return 0;
   else {
      int add = (x%2) ? (x%10) : 0;
      return add + sumOddDigs(x/10);
   }
}
```

55.
```
int sumParityDigs(int x, bool even) {
   if(x == 0)
      return 0;
   int add;
   if(even)
      add = !(x%2) ? (x%10) : 0;
   else
      add = (x%2) ? (x%10) : 0;
   return add + sumParityDigs(x/10, even);
}
```

56. You will get an error saying "lvalue required as increment operand". Meaning: You can't increment a constant!

57. `i = 1, m = 0, n = 3`

58. A doubly-linked list.

59. `r`

60. Filling in the blanks:

```
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
   FILE *textFile = fopen("mymoves.txt", "w");

   if(textFile != NULL) {
      while(!won()) {
         fputc(convertToCharacter(tile), textFile);
         fputc('\n', textFile);
      }
   } else
      return 1;

   fclose(textFile);
   return 0;
}
```

61. The program, despite the crazy variable names, takes one command line argument, prints it out, and then prints it out backwards.