Quinton Heald 811-897-219

1 Functions, Pointers, and Tricky Declarations Activity

1. The output of the following code will be 22 on the first line, and 11 on the next line. The first line is 22 because the function divBy2 that takes in the parameter of the value pointed to by the given variable is used. The next line will be 11 because the function that takes in the address of the provided variable is used. Both functions have the same name, but they take different parameters. The function “divBy2(& n)” creates the variable in the code that is “passed-by-reference”. Passed-by-reference implies that a variable is modified without a copy of said variable being created. In this case, the reference of x is used to change the value of x without an additional copy being created.
2. &x is used as the parameter in line 16 to provide the proper parameter for the function divBy2(int & n). If just x was used, this would call the function divBy2(int * n) since that takes in the value pointed to by the provided variable; however the first version of divBy2 takes in the parameter of a reference to provide distinction between the 2.
3.

int x;	x is an int
int * x;	x is a pointer to an int
char ** x;	x is a double pointer to a char
int * x [5];	x is an array of 5 pointers to an int
int (* x) [5];	x is a pointer to an array of 5 ints
int (* x [5]) [5];	x is an array of 5 pointers to arrays of 5 ints
int * (* x [5]) [5];	x is an array of 5 pointers to arrays of 5 pointers to ints
int x();	x is a function with no parameters returning an int
int x(int);	x is a function with a parameter of an int returning an int
int * x();	x is a function with no parameters returning a pointer to an int
int * x(int *);	x is a function with a parameter of a pointer to an int that returns a pointer to an int
int (* x)();	x is a pointer to a function with no parameters returning an int

int ** (* x) (int **);	x is a pointer to a function with a parameter of a double pointer to an int that returns a double pointer to an int

2 Const Pointers Activity

1a. The code of part a is valid. The first line establishes a nonconstant pointer to a nonconstant int. The next line modifies the data to which the pointer points to, which since they are both nonconstant, is legal.

1b. The first line of part b creates a nonconstant pointer p2 to a constant int set to the address of variable x. The next line is invalid, since it changes the data to which p2 points to, but since p2 points to a constant int, its data cannot be modified. The next line, however, is valid, since p2 is a nonconstant pointer which means the pointer can be modified to point to other data of the appropriate type.

1c. In the first line of part c, a constant pointer is set to a constant int. This makes the next two lines invalid, as they both attempt to change the values of both the address pointed to and the value of the address. This cannot be done since both values are set to constants.

2a. In part a, a nonconstant pointer is set to a nonconstant int at the value of a. Next line, a nonconstant pointer is set to a constant int, set to the value of a. Next line, a constant pointer is set to a constant int at the value a. All of these lines are valid initializers.

2b. In part b, on the first line, p1 is increased by one, which is valid since it is a nonconstant pointer, allowing for the address it points to to be changed. The next line is also valid, since while p2 points to a constant integer, p2 itself is a nonconstant pointer, which allows the address it points to to be changed as well. However, the third line where p3 is increased by one is invalid, since p3 is a constant pointer, the address to which it points to cannot be changed.

2c. The first line of part c is invalid, since an array cannot simply be reassigned to another separate array. The second line is also invalid since the array itself cannot be increased by a value of one. The third line of a, however, is valid, and will simply increase the first value of the a array by one.

3a. In part a, line one is valid, and will result in x being assigned to 2. Line 2 is also valid since x is not supposed to be constant, and its value can change, and in this case, increase by 1. The third line, however, is not valid. The function of foo() itself cannot be increased by a value of one, and is only able to return the integer of 2, also due in part to the fact that foo returns a constant integer, which the value of cannot be changed.

3b. In part b, a nonconstant pointer to a constant integer is set to the value of function bar() which returns a constant integer of a (1). The second line of part b is valid, since p1 is a nonconstant pointer, therefore the address to which it points to may be changed. However, the third line is invalid, since p1 points to a constant int value, which since it is constant, makes it unchangeable. The fourth line of part b is also invalid, since the function of bar itself cannot be increased. The final line of part b is also invalid, since the value to where bar is cannot be incremented either, even if read aloud it implies the value of where the bar function is, it cannot be incremented.

3c. The first line of part c creates a nonconstant pointer of a constant integer set to the value of the function baz(), which returns a constant integer of the first value of a. The second line of part c is valid, since p2 is a nonconstant pointer, therefore, the address to which it points to can be changed. The third line is invalid however, since p2 points to a constant integer, which means that the value of which it points to is unchangeable.
Line 4 is also invalid, because baz is a constant pointer returning a nonconstant integer, which means that the address of which the baz function points to is unchangeable.
However, because of how the baz function is initialized, line 5 is valid, since the value that the baz function returns is nonconstant, and therefore it may be incremented by 1.