Stephen Nemeth

1. Functions, Pointers, and Tricky declarations activity.

    1) The proper output of the code will be 22 on one line then 11 on a new line. The original variable for the int x is 44. The next instruction is divBy2(x) which calls the function divBy2(int & n) that takes in n as a reference to an integer also known as pass by reference. In this case x with the value of 44, and divides it by two yielding 22. Since the function takes in a reference to an integer it modifies the original integer meaning the x in main and n in divBy2() are the same integer being manipulated. Next the function divBy2(&x) is called, which goes to the divBy2(int * np) function. This is because the "&" operator gives the function the address of x as the parameter and divBy2(int * np) takes in a pointer to an int which is now housing the address of x. Then within divBy2(int * np) it is using the "*" operator to perform the operation *np = *np / 2, this changes the actual value that np is pointing to rather than the memory address. Since x is 22 before this function call and np is now pointing to the address of x, the value stored at the address np is pointing to is now 11 which is also x since np is pointing to the address of x.

    2) &x is used as a function parameter because it is calling the divBy2(int * np) function. Where np is a pointer to an int and needs a memory address of an int. The "&" supplies the memory address for x and supplies that to the function being called assigned the memory address of x to the pointer np. This allows it to manipulate it with a dereference operator "*" just like how it is doing by dividing

it by 2 with the statement *np = *np / 2, this also changes the value of the int np points to which is x.

3)

Declaration Syntax	Meaning
int x	x is an int
int * x	x is a pointer to an int
char ** x	x is a pointer to a pointer to a char
int * x [5]	x is an array of 5 pointers to ints
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 that returns an int
int x(int)	x is a function that has one int parameter and returns an int
int * x()	x is a function with no parameters that returns a pointer to an int
int * x(int *)	x is a function that has one pointer to an int parameter that returns a pointer to an int
int (* x) ()	x is a pointer to a function that has no parameters and returns an int
int ** (* x)(int **)	x is a pointer to a function that has one pointer to a pointer to an int parameter that returns a pointer to a pointer to an int

2. Const Pointer Activity

1)

a) Both statements are valid for part a because p1 is a non-constant pointer to non-constant data, and x is a non-constant the assignment int * p1 = &x works and causes no errors. Since x is non-constant that statement *p1 = 11, also works since x is allowed to be modified.

b) The first statement const int * p2 = &x is a valid statement but the next statement *p2 = 11, is not because p2 is declared as a pointer to a constant integer. Since the value that it is pointed to is constant in the pointer's eyes it cannot be modified. The third statement p2 = &y is valid, since p2 is not a constant pointer it can change the address it is pointing to.

c) The first statement const int * const p3 = &x is valid, but the next two statements are invalid. Since p3 is a constant pointer to constant data that data cannot be modified which is why *p3 = 11 is invalid. Also, since p3 is a constant pointer to constant data the address that p3 points to cannot change which is why the statement p3 = &y is invalid and not allowed.

2)

a) All three statements are valid, it just creates different types of pointers pointing to the first element in a.

b) The first statement p1++ is valid because p1 is a non-constant pointer to non-constant data and this operation shifts p1 to point to another location in memory. The second statement p2++ is also valid because p2 is a non-constant pointer meaning it is able to be shifted over and point to a new location in

memory. The third statement p3++ is invalid because p3 is a constant pointer and cannot change the location in memory that it is pointing to.

c) The first statement,a = b, and the second statement, a++, is invalid because a is an array and arrays in C++ cannot be assigned new addresses. a = b attempts to assign the address that a is pointing to towards the address b is pointing to, but a is an array and this is invalid. The next statement, a++ attempts to increment the address a is pointing to by 1 but arrays cannot be assigned new addresses, therefore invalid. The third statement (*a)++ is valid because we are using the dereference operator on a which is referring to the value in the first element and incrementing it by one which is valid.

3)

a) Both the first two statements are valid and will compile and run just fine. The last statement. foo()++ is invalid because foo() is a function call and this will cause a compiler error since the postfix increment operator requires an lvalue to increment and foo() is not a value or a variable.

b) Only the first two statements are valid. The third statement, (*p1)++, is invalid because p1 is pointing to constant data and therefore the data cannot be modified with a dereference operator. The third statement, bar()++, bar returns is invalid because bar() is a function call and not a value, therefore it cannot be used as an lvalue for the postfix increment operator. The last statement, (*bar())++ is invalid because the value that bar() returns is a pointer to constant data and therefore cannot be modified with a dereference operator.

c) Only the third and fourth statements are invalid. The third statement is invalid because the value returned by baz() is being assigned to a pointer to constant data and therefore cannot be modified by the statement with the dereference operator (*p2)++ because p2 is pointing to constant data. The fourth statement is invalid because baz() is a function call and therefore cannot be incremented with a postfix increment operator since baz() is not an lvalue and the postfix increment requires an lvalue.