This document is an overview of the concepts explored in the GS C Coding Worksheet, and meant as an accompaniment to the solutions document to provide another perspective on the material.

1. BuggC
    a. This problem tests your understanding of pointers and their behavior in a for loop. The key observation in catching this bug is the variable being iterated over in the loop is not being incremented by the loop. There are many ways to fix this issue, but the official solution does it by modifying the for loop's termination condition. Recall that the termination condition (the second section) of a for loop is tested every iteration. Because the assignment statement `(c = *s)` has a higher precedence than the logical `!=` operation the solution first assigns the value at s (which is being modified by the loop) to c before getting a true or false result and continuing the iteration as appropriate.

    b. The pointer s is incremented in the for loop even before fixing 1a. However, s is pointer and this line makes an assignment to the pointer, not the value being pointed at! In other words, we're missing a * before s. If the 'a' - 'A' part is confusing to you, I recommend looking at an ASCII table like http://www.asciitable.com/ . This line works because the numeric difference between a lower and uppercase letter's ASCII encoding is the same for all letters.

    c. There's nothing wrong with this line, although it can certainly be confusing at first glance. Usually while loops have a body following them in brackets, but this one doesn't. This one just iterates over the termination condition until that condition yields a falsy value. Notice the termination condition in the loop has two operators attached to it: "*" and "++". One of them takes effect before the other. This link https://en.cppreference.com/w/c/language/operator_precedence tells us that a suffix ++ happens before the dereference. You may have seen a suffix ++ used just to increment a variable, but it actually does two things. It first evaluates to the current value of the variable in the expression where its used and then it increments the variable after the expression. So in the while loop s2 is first evaluated at its current value then that current value is dereferenced, and used to test the while loop's termination condition. Only after that is s2 incremented. If s2 dereferences to the null terminating character at the end of the string it will evaluate to false in the while loop and terminate iteration.

    d. There's nothing wrong with this line either. The while loop above has incremented s2 one past the '\0' at the end of the string before terminating. This is because the while loop terminated when the expression evaluated to '\0', but then the ++ on s2 incremented the pointer after the expression was evaluated. s2 - s -1 gives the length of the string.

e. There are two mistakes on this line! As written, *numbers + i, adds i to the value pointed to by numbers. It doesn't increment the pointer to do the intended array access. We can fix this with parenthesis. Secondly, the && operator is a logical operator not a bitwise operator! Clearly a logical AND has nothing to do with counting odds. There are a lot of ways to fix this line, but the simplest way is to use a bitwise &. Every odd unsigned integer has a 1 at its least significant bit, which will result in a nonzero result when ANDed with 0x1.

2. Insert TreeNode
    a. This question tests your understanding of memory allocation, c structs, correct usage of operators for assigning pointers to structs, and logical handling of the different cases for node insertion. First you should begin by allocating memory for a new tree node. Be sure to correctly use sizeof within malloc to allocate enough memory, refer to the struct as "struct TreeNode" instead of just TreeNode, and typecast the pointer returned from malloc with (struct TreeNode *). Next, assign the correct values to the new struct's fields using the "->" operator. Recall, "node->value" is shorthand for "(*node).value" which is necessary when working a pointer to a node. Finally, handle the logic for inserting the new node into the tree by checking to see if the parent's left child is null.

3. Number Pushing and Popping
    a. This question is much like the previous one, only with respect to a stack instead of a tree. The same idea for the solution works: correctly use malloc with the struct, update struct fields with ->, and handle the logic for a stack when inserting the new stack element.

    b. To implement popadd correctly, you must handle the cases where the stack is empty, has one element, has two elements, or has more than two elements. In the last case, a correct implementation should remove a node from the data structure by updating top_of_stack to have the correct value, pointer to next element, and free the memory used by the detached element.

4. A lot of Pointing
    a. This question evaluates your knowledge of pointers and arrays in memory. Pointer arithmetic in C is smart enough to know about the number of bytes in memory needed to store an integer. Incrementing a pointer will cause it to point to the next integer (or what C thinks is the next integer) in memory. Therefore *(p+3) evaluates to the fourth element in the array, 0x4.

b. The array access syntax used here is another way, besides pointers, to get at a spot in memory.

c. This line combines the two different styles of array access used above to sum together elements of the array. Adding the integers 0x4 and 0x4ffc in hex produces 0x5000. To see this, notice that 0xc is only 4 away from carrying over to the next place. The addition is analogous to adding 4 to 4996 in decimal.

d. C lets you do strange things, like cast the value of an integer as a pointer to an integer. p is a pointer, p[1] is an integer, (int *) p[1] is a pointer, *(int *)p[1] is an integer dereferenced from a pointer. It just so happens that the value of p[1] when interpreted as a memory address is a location in this question's array. On modern computers, C ints are 32 bits which means each spot in the array is separated by 4 bytes. By that logic 0x5008 is two array spots down 0x5000 and we're looking at the third element in the array.

e. This expression is similar to part d above, but here the memory address 0x4ffc is not part of the array. This expression will do something undetermined and is an error.

5. Reverse!
    a. This question is similar to questions 2 and 3 above in that it involves memory allocation, structs, and data structure logic. Here the data structure is a linked list. For the first line, this question asks us to give the type of head_ptr. Looking down to the code already written in the while loop we can see that head_ptr must be some kind of pointer because it is dereferenced. But notice the right side of the last line of the while loop contains the line (*head_ptr)->next which is shorthand for (*(*headptr)).next. That means it must be typed as a struct list node **. The next blank line assigns something to ret, which on the next few lines is used via ret->val and ret->next. It makes sense for these next lines to be an assignment to the new node we are copying into, which means ret must be allocated here. The remainder of the lines in the while loop setup pointers for the new node, but because this list is being created in reverse with respect to the original list we want the next node from the original list to have a reversed relationship to the current node in the new list. Think of the new list as growing backards. By returning ret at the end, we are returning a pointer to the first node in the new list.

    b. In my opinion this is a hard question! Good job if you solved it!