Programming for Performance

Lab Session 1: Catch the Bugs.

1. This problem tests your understanding of memory bugs. Each of the code sequences below may or may not contain memory bugs. The code all compiles without warnings or errors. If you think there is a bug, please circle **YES** and indicate the type of bug from the list below of memory bugs. Otherwise, if you think there are no memory bugs in the code, please circle **NO**.

Bugs:
1. Potential buffer overflow error
2. Memory leak
3. Potential for dereferencing a bad pointer
4. Incorrect use of free
5. Incorrect use of realloc
6. Misaligned access to memory
7. Other memory bug

Part A

```
/*
* strndup - An attempt to write a safe version of strdup
*
* Note: For this problem, assume that if the function returns a
* non-NULL pointer to dest, then the caller eventually frees the dest
      buffer.
*/
char *strndup(char *src, int max)
{
      char *dest;
      int i;

      if (!src || max <= 0)
            return NULL;
      dest = malloc(max+1);
      for (i=0; i < max && src[i] != 0; i++)
            dest[i] = src[i];
      dest[i] = 0;
      return dest;
}
```

NO **YES** Type of bug:_____

Part B

```
/* Note: For this problem, asssume that if the function returns a non-
NULL
* pointer to node, then the caller eventually frees node. */
struct Node {
      int data;
```

```
        struct Node *next;
};
struct List {
        struct Node *head;
};
struct Node *push(struct List *list, int data)
{
        struct Node *node = (struct Node *)malloc(sizeof(struct Node));
        if (!(list && node))
                return NULL;
        node->data = data;
        node->next = list->head;
        list->head = node;
        return node;
}
```

NO **YES** Type of bug: _____

Part C

```
/* print_shortest - prints the shortest of two strings */
void print_shortest(char *str1, char *str2)
{
        printf("The shortest string is %s\n", shortest(str1, str2));
}
char *shortest(char *str1, char *str2)
{
        char *equal = "equal";
        int len1 = strlen(str1);
        int len2 = strlen(str2);
        if (len1 == len2)
                return equal;
        else
                return (len1 < len2 ? str1 : str2);
}
```

NO **YES** Type of bug: _____

2.

```
/*
 * This program has various memory related problems that provide a good way
 * to show off the various abilities of valgrind.  To run it:
 *
 *   valgrind <optional valgrind options> ./valgrind-tests <test number>
 *
 * where <test number> is between 1 and 9, inclusive.  Suggested
 * valgrind options to run with are
 *
 *   --logfile=valgrind.output --num-callers=6 --leak-check=yes
```

```
 */

#include <cassert>
#include <iostream>

using namespace std;

void
test_1()
{
  // This test provides an example of using uninitialised memory
  int i;
  printf("%d\n", i);               // Error, i hasn't been initialized

  int * num = (int*)malloc(sizeof(int));
  cout << *num << endl;            // Error, *num hasn't been initialized
  free(num);
}

void
test_2()
{
  // This test provides an example of reading/writing memory after it
  // has been free'd
  int * i = new int;
  delete i;
  *i = 4;                          // Error, i was already freed
}

void
test_3()
{
  // This test provides an example of reading/writing off the end of
  // malloc'd blocks
  int * i = (int*)malloc(sizeof(int)*10);
  i[10] = 13;                      // Error, wrote past the end of the block
  cout << i[-1] << endl;           // Error, read from before start of the
block
  free(i);
}

void
test_4()
{
  // This test provides an example of reading/writing inappropriate
  // areas on the stack.  Note that valgrind only catches errors below
  // the stack (so in this example, we have to pass a negative index
  // to ptr or valgrind won't catch the problem)
  int i;
  int * ptr = &i;
  ptr[-8] = 7;                     // Error, writing to a bad location on
stack
  i = ptr[-15];                    // Error, reading from a bad stack
location
}

void
```

```
test_5()
{
  // This test provides an example of memory leaks -- where pointers
  // to malloc'd blocks are not freed
  int    * i = new int;
  static double * j = new double;
  i = NULL;
  // Note that neither i or j were freed here, although j being static means
  // that it will be considered still reachable instead of definitely lost
}

void
test_6()
{
  // This test provides an example of mismatched use of
  // malloc/new/new [] vs free/delete/delete []
  int * i = new int;
  free(i);                          // Error, new/free mismatch
  double * j = new double[50];
  delete j;                         // Error, new[],delete mismatch
}

void
test_7()
{
  // This test provides an example of overlapping src and dst
  // pointers in memcpy() and related functions
  char big_buf[1000];
  char * ptr_1 = &big_buf[0];
  char * ptr_2 = &big_buf[400];
  memcpy(ptr_1, ptr_2, 500);        // Error, dst region overlaps src region
}

void
test_8()
{
  // This test provides an example of doubly freed memory
  int * i = new int;
  delete i;
  delete i;                         // Error, i delete'd twice
}

void
test_9()
{
  // This test provides an example of passing unaddressable bytes to a
  // system call.  Note that the file descriptors for standard input
  // (stdin) and standard output (stdout) are 0 and 1 respectively,
  // which is used in the read(2) and write(2) system calls (see the
  // respective man pages for more information).
  char * buf = new char[50];
  printf("Please type a bunch of characters and hit enter.\n");
  read(0, buf, 1000);               // Error, read data overflows buffer
  write(1, buf, 1000);              // Error, data comes from past end of
buffer
  delete[] buf;
}
```

```
int
main(int argc, char**argv)
{
  if (argc!=2) {
    cerr << "Syntax:" << endl;
    cerr << "  " << argv[0] << " <test-number>" << endl;
    return -1;
  }
  int test_number = atoi(argv[1]);

  switch (test_number) {
    case 1: test_1(); break;
    case 2: test_2(); break;
    case 3: test_3(); break;
    case 4: test_4(); break;
    case 5: test_5(); break;
    case 6: test_6(); break;
    case 7: test_7(); break;
    case 8: test_8(); break;
    case 9: test_9(); break;
    default: cout << "No test or invalid test specified (only 1--9 are
valid)."
                  << endl;
             return -1;
  }

  return 0;
}
```