Your login name: cs61c-________

Problem 1

Recall the definition of root in project 1. (The declaration of struct entryNode appears below.)

```
struct entryNode * root;
```

Give the type of each of the following expressions. The answer may be “error”. Assume that root and the fields that it points to have all been initialized with reasonable values.

`(root->name)[3]`	char if the name contains at least three characters
`&(root->parent)`	struct entryNode **
`root.next`	error (should be “->”, not “.”)
`root->contents`	error (should be root->entry.contents)

Definition of struct entryNode

```
struct entryNode {
   char * name;
   struct entryNode * next;            /* sibling */
   int isDirectory;
   struct entryNode * parent;
   union {
      char * contents;                 /* for a text file */
      struct entryNode * entryList;    /* for a directory */
   } entry;
};
```

[Note: we didn’t cover union in fall08 CS 61CL.]

Problem 2

Add the two 2’s complement signed 8-bit values given below, and express your answer in decimal.

```
     01100110
+    10010100
-------------
   11111010 = -6
```

Problem 3

Complete the framework below to complete the implementation of the appendFiles function for project 1. appendFiles takes four arguments:

- a pointer to the working directory;
- the name of a text file in the working directory;
- the name of another text file in the working directory;
- the name of a text file to create, which will contain the concatenation of the contents of the first and the second files.

You may assume that the functions located and inserted have already been defined. Here are their headers (the declaration of struct entryNode appears with an earlier problem on this exam):

```
/* The list of entries is sorted alphabetically by name.
   Return the result of inserting the new entry into the list. */

struct entryNode * inserted (struct entryNode * newFile,
                             struct entryNode * list);

/* Return a pointer to the entry with the given name in the given list,
   or NULL if no such entry exists. */

struct entryNode * located (char * name, struct entryNode * list);

void appendFiles (struct entryNode * wd,
                  char * fileName1, char * fileName2, char * newName) {
   struct entryNode * file1 = located (fileName1, wd->entry.entryList);
   struct entryNode * file2 = located (fileName2, wd->entry.entryList);
   if (file1 == NULL) {
      printf ("append: file %s not found\n", fileName1);
   } else if (file2 == NULL) {
      printf ("append: file %s not found\n", fileName2);
   } else if (file1->isDirectory) {
      printf ("append: can't append directory %s\n", fileName1);
   } else if (file2->isDirectory) {
      printf ("append: can't append directory %s\n", fileName2);
   } else if (located (newName, wd->entry.entryList)) {
      printf ("append: can't overwrite file %s\n", newName);
   } else {
      struct entryNode * newFile;

      // YOU PROVIDE THIS CODE.

      wd->entry.entryList = inserted (newFile, wd->entry.entryList);
   }
}
```

```
// 2 points for malloc on node
newFile = (struct entryNode *) malloc (sizeof (struct entryNode));

// 2 points for malloc on name
newFile->name = (char *) malloc (strlen(newName)+1);

// 2 points for strcpy
strcpy (newFile->name, newName);

// 3 points for malloc on contents
newFile->entry.contents =
   (char *) malloc (strlen(file1->entry.contents)
   + strlen(file2->entry.contents) + 1);

// 3 points for contents copy
strcpy (newFile->entry.contents, file1->entry.contents);
strcpy (newFile->entry.contents + strlen(file1->entry.contents),
   file2->entry.contents);

// 2 points for the rest of newFile
newFile->isDirectory = 0;
newFile->parent = wd;
```

Problem 4

Part a

Given the following definition,

```
struct node {
   char name[12];
   int value;
};
```

what is sizeof (struct node)? _____

Assume that the sizes of chars and ints are the same as on the 271 Soda computers.

1 point, all or none.
The answer is 16 bytes.

Problem 5

The following program includes the buggy swap function encountered in a pre-lecture quiz. Some students observed that this function "worked" because of values accidentally in memory.

```
#include <stdio.h>

void swap (int *a, int *b) {
   int *temp;
   *temp = *a;
   *a = *b;
   *b = *temp;
}

int main ( ) {
   int x, y, z;
   x = 2;
   y = 3;

   f ( ____________________________ );     /* Supply the argument(s) to f. */
   swap (&x, &y);
   printf ("The values of x and y are now %d and %d.\n", x, y);
   return 0;
}
```

In the space below, supply the definition of a function f, and supply a call to f in the blank above, that will *guarantee* that the program will *not* "work", that is, it will crash when the uninitialized temp pointer is dereferenced. Also explain why your call guarantees that swap will crash.

Their solution has to have a local variable in the proper orientation that gets initialized to 0, for example:

```
   void f (int a, int* b) {
      int c;
      c = 0;
   }
```

3 points in all:

1 for the code;
1 for saying something reasonable about a position on the stack
1 for saying something reasonable about a value sure to produce a crash when dereferenced

Problem 6

Write a C function named copyStrArray that, given an integer count and an array strArray that contains count strings, returns a pointer to a complete ("deep") copy of the array. (In Java terminology, this would be a "clone".) For example, the program segment

```
int main (int argc, char **argv) {
   char **ptr;
   ptr = copyStrArray (argc, argv);
      ...
```

would place in ptr a pointer to a copy of argv, the command-line argument structure.

You may use brackets and other array notation in your solution. You may assume that there is sufficient free memory in which to build the copied structure. Make no assumptions about the size of a pointer or a char. Include all necessary casts, and allocate only as much memory as necessary. You may use any function in the stdio, stdlib, or string libraries.

```
char **copyStrArray (int count, char **strArray) {
   char **copy;
   char **temp;
   int k;
   copy = (char **) malloc (count*sizeof(char *));
   temp = copy;
   for (k=0; k<count; k++) {
      (*temp) = (char *) malloc (((strlen(*strArray)+1)*sizeof(char));
      strcpy (*temp, *strArray);
      temp++;
      strArray++;
   }
   return copy;
}
```

If only one malloc, deduct 2; if no mallocs, deduct 4. Otherwise:
(1.5 points) malloc'ing the 1-d array of pointers (with the correct length and cast);
(1.5 points) malloc'ing the space for each string (with the correct length and cast);
(1.5 points) copying the string into the right place
(1.5 points) everything else: traversing the 1-d array of pointers; loop control, declarations, return value.
This probably works out to -1/2 per small error.

Problem 7

Consider the following C program.

```
#include <string.h>

int main ( ) {
   char oneName[10] = "xxxxxxxxx";
   char* anotherName;

   anotherName = oneName;
```

```
        strcpy (anotherName, oneName);
        oneName[1] = 'O';
        return 0;
    }
```

Draw the box-and-pointer diagram that represents the oneName and anotherName arrays and their contents after executing this segment.

```
oneName ----> |"xOxxxxxxx"| followed by null char
               ^
               |
anotherName ---+
```

Anticipated errors: no sharing; only part sharing, no arrows, off-by-one indexing.

two copies of string => -3
anotherName nil => -3
anotherName points to last char of oneName string => -2
anotherName points to oneName, which points to char => -2
off by one on char assignment => -1
off by one on count of chars => -1
didn't show trailing null => -1

Problem 8

Write a C function named resultOfInsert that, given a string s, a character c, and a position k between 0 and strlen(s), inclusive, returns the string that results from inserting c into s at position k. For example,

```
resultOfInsert ("abcd", '_', 2)
```

should return the string "ab_cd". The insertion should not change the argument string. You may use any functions declared in <string.h>. You don't need to do any error checking.

```
char *resultOfInsert (char *s, char c, int k) {
```

malloc call is worth 2 points, 1 for trying it and 1 for getting it right.
everything else is worth 3 points: -1 for each error, like off-by-one, forgetting terminating 0, using = for ==, forgetting to return something

Solution code:

```
char *resultOfInsert (char *s, char c, int pos) {
   char *rtn = (char *) malloc (strlen(s)+2);
   int k;
   for (k=0; k<pos; k++) {
      rtn[k] = s[k];
   }
   rtn[pos] = c;
   for (k=pos; k<=strlen(s); k++) {
      rtn[k+1] = s[k];
   }
   return rtn;
}
```

Another solution:
```
   for (k=0, j=0; k<=strlen(s); k++, j++) {
      if (k==pos) {
         rtn[j] = c;
         j++;
      }
      rtn[j] = s[k];
   }
```

Problem 9

Consider the following C program.

```
#include <stdio.h>

struct foo {
   int *ip;
   int x;
   struct foo *p;
   int a[3];
};

int w[10] = {5, 22, 781, 0, 104, 25, 18, 24, 14, 30};

struct foo foo1 = {&w[4], 99, NULL, {11, 12, 13}};

struct foo foo2 = {w, 87, &foo1, {77, 78, 79}};

int main ( ) {
   foo1.p = &foo2;
   printf("%d\n", foo1.x);
   printf("%d\n", foo2->x);
   printf("%d\n", foo2.p->x);
   printf("%d\n", foo2.p.x);
   printf("%d\n", foo1.a[1]);
   printf("%d\n", foo1.a);
   printf("%d\n", foo1.ip[2]);
   printf("%d\n", *foo1.ip);
   printf("%d\n", *(foo1.ip + 4));
   printf("%d\n", *(foo1.a + 1));
   printf("%d\n", foo1.a + 1);
}
```

Part a

Predict the result of each of the printf lines. (Some may cause compiler errors.)

```
line 1: 99
line 2: compiler error
line 3: 99
line 4: compiler error
line 5: 12
line 6: wrong format; 133580
line 7: 18
line 8: 104
line 9: 14
line 10: 12
line 11: wrong format; 133584
```

Part b

List three different ways you could fill in the blank in the statement below to print the value 781 by extracting it from the data structures given in the above program.

```
printf("%d\n", ______ );
```

Do the same for the values 77 and 87.

```
Some ways to get 781: w[2], *(w+2), foo2.ip[2], *(foo2.ip+2)
Some ways to get 77: foo2.a[0], *(foo2.a), foo1.p->a[0]
Some ways to get 87: foo2.x, foo1.p->x, (*foo1.p).x
```

Problem 10

Consider the following C function, intended to append the characters in the second argument to the first argument (modifying the first argument in the process).

```
/* Precondition: s1 and s2 are correctly allocated C-style strings.

char* strcat (char *s1, char *s2) {
   char* ptr = s1;
   for ( ; *ptr != '\0'; ptr++) {
   }
   for ( ; *s2 != '\0'; s2++, ptr++) {
      *ptr = *s2;
   }
   return s1;
}
```

Part a

The function compiles without error messages. However, it contains two flaws that are likely to produce execution errors. Describe them.

The function may overfill the storage allocated to s1.
It doesn't put a terminating null character in the returned string.

Part b

Suppose that the flaws in the strcat function have been corrected. The program below, calling the revised version of strcat, prints the characters abcdef as expected.

```
void main ( ) {
   char str1[10] = "abc";
   char str2[10] = "def";
   printf ("The result of concatenating str1 and str2 is %s.\n",
   strcat (str1, str2));
}
```

Replace the arguments of the strcat call with equivalent arguments that use the & or the * operator.

Replacement call: strcat (__________ , __________)

strcat (&str1[0], &str2[0]);
or
strcat (&*str1, &*str2);
or
strcat (*&str1, *&str2);

Problem 11

What is printed by the following C program segment, run on the EECS instructional computers? (The notation 0x… specifies a hexadecimal constant; in a printf format string, %x specifies output in base 16.)

```
int *ptr;
ptr = 0x1050;
printf ("%x\n", ptr--);
```

```
      printf ("%x\n", ptr);
```

Answers: 1050, 104c
3 points: 1 for correct postincrement, 1 for correct base 16, 1 for correct pointer arithmetic.

Problem 12

Convert the eight-bit binary value 11110000 to:

a. hexadecimal.

F0

b. decimal, interpreting it as a unsigned value.

15*16 = 240

c. decimal, interpreting it as a two's complement signed value.

Complement and add 1 to find out what's it's the negative of. Answer = -16.

Problem 13

a. Describe the set of numbers that are represented by a 64-bit two's complement integer.

-2^63 to 2^63-1

b. Add the following 16-bit integers by hand, show the binary result and translate it to hexadecimal.

```
 0001011011100101
+0011101001101100
 0101000101010001 = 5151 base 16
```

c. Multiply the following 16-bit unsigned integers by hand, show the binary result and translate to hexadecimal

```
 0000000001101111
*0000000000001010
```

```
    11011110
+  1101111000
= 10001010110
```

Check: 1010 = 10 base 10; 1101111 = 6*16 + 15 = 111 base 10; 10 * 111 = 1110 = 456 hex.

Problem 14

Suppose you are given the following (correct) C declarations.

```
char foo = 'A', garply[] = "MIPS", bar = 'C';
char *phi = &foo, *beta = phi, *gamma = garply;
```

In the following, some of the statements are incorrect or illegal; cross out any such bad statements. Show in the spaces provided what the remaining print statements will print when the program is executed.

```
printf("%c", *beta);
```

beta is a char*, so *beta is a char; A is printed

```
printf("%c", phi);
```

phi is a char*, not a char; ERROR

```
printf("%c", *gamma);
```

gamma is a char*; M is printed

```
printf("%s", bar);
```

bar is a char; ERROR

```
      beta = garply;
```

beta is char *, which is assignment-compatible with garply

```
      printf("%c", *(garply + 2));
```

prints P

```
      printf("%s", &garply[1]);
```

prints IPS

```
      garply = phi;
```

can't assign to an array name; ERROR

```
      printf("%c", *beta);
```

prints M