15-122 : Principles of Imperative Computation

Fall 2013

Midterm 2 Review (Topics in C)

Exam Date: Tuesday, Nov. 12, 2013

1 Handout

I've put together a basic handout for you, to get some practice in writing C code. To get to the handout, you should do the following. First, ssh into your Andrew space, and navigate to a folder where you want to do your practice. Once there, do the following (don't ommit the period!):

```
cp /afs/andrew/usr23/eszabowe/15122_Review.tar .
tar -xvf 15122_Review.tar
cd 15122_Exam_Review
```

You should now have 4 directories in your space: (1) basics, (2) usage, (3) memory, (4) lib. There are 4 example C files, 2 in basics, 1 each in usage and memory. You should go through each, and attempt to finish the files as specified, without consulting any external sources, or compiling. You can get the solutions by doing:

```
cp /afs/andrew/usr23/eszabowe/15122_Exam_Review_Soln.tar .
tar -xvf 15122_Exam_Review_Soln.tar
cd 15122_Exam_Review_Soln
```

2 Basics (Syntax)

Much of the syntax remains the same, between C and C0. To make sure you remember it all, try to fill in ex1.c and ex2.c in the basics folder.

```
#include <_____> // standard C library
#include <_____> // C terminal I/O

__________ { // a structure named ex1
  ______ x; // an integer called x
  ______ y; // an integer called y
  ______ L; // a pointer to a structure of type ex1, called L
  ______ R; // a pointer to a structure of type ex1, called R
};

________ // a typedef which lets us call instances of ex1 by "ex1"

int main () {
  ex1 runner;
  int sum;
  ______; // assign the field of x on runner to be 5
  ______; // assign the field of y on runner to be 7
  ______; // assign sum to be the sum of x and y
  ______; // print the following string: "Runner runs to #",
          // where # is sum
  return sum;
}
```

```
#include "../lib/exam_prep.h" // Do not modify this

// Assume the following has been executed:
/*

  typedef struct {
    bool bit1;
    bool bit2;
    bool allow;
    char *str;
  } ex2;

 */

int main () {
  ex2 *walks = prep_helper1();
  ______; // assign the allow field of walks to be [bit1 or bit2]
  selective_print(walks);
  free(walks);

  ex2 *walks2 = prep_helper2();
  ______; // assign the allow field of walks2 to be [bit1 and bit2]
  selective_print(walks);
  free(walks2);

  return 0;
}

void selective_print (ex2 *input) {
  if (_____) // if allow is asserted within input, then
    ______; // print the str field from input
}
```

3 Usage (More Syntax)

There are a couple of new things in C, among them the switch statement, and the address of operator. Fill in the blanks in ex3.c, to get some practice with switches.

```
#include "../lib/exam_prep.h" // DO NOT MODIFY

int main () {
  char some_char = gen_char();
  int x;
  ______ { // case on some_char
    ________; // if 'a', set x to 0
    ________; // if 'b' set x to 1
    ________; // if 'c' set x to 2
    ________; // otherwise, set x to 10
  }
  prep_output_3(x);

  // Assume we have:

  /*

typedef struct {
  bool p;
  char *str;
} ex3;

   */

  ex3 *gen = prep_helper3();
  __________; // using the ternary operator (?),
              // print gen->str if p is true.
              // otherwise, print "Nope! You can't make me."
  free(gen);
  return 0;
}
```

4 Memory (Management)

A major distinction between C and C0 is that C0 is garbage collected, while C is not. This means that every bit of memory in C allocated on the heap (via a call to one of the malloc functions) must be explicitly freed, else it will not be reclaimed. If a program does not free its memory, we say that it has a *memory leak*, and this is a ***Very Bad Thing***^{TM}. To refresh yourself on malloc/calloc, and freeing memory, fill in the blanks in ex4.c.

```
#include "../lib/exam_prep.h"

typedef struct tree {
  int d;
  struct tree *L;
  struct tree *R;
} tree;

int main () {
  tree *root = ______; // allocate the tree using a malloc function
  tree *l    = ______; // allocate left child with malloc function
  tree *r    = ______; // allocate right child with malloc function
  _____;
  _____; // set data to be 0 for left child, 1 for root, 2 for right
  _____;

  _____; // free the memory (this may take multiple lines)
  return 0;
}



/**
 * Frees a tree recursively. We assume that every node in the
 * tree has been allocated via a call to malloc, and the memory
 * must now be returned.
 * REQUIRES: R is a valid binary tree (enforced by data structure)
 * ENSURES : tree_free(R) will free all of the memory of the left
 *           and right subtrees of R, and then the memory associated
 *           with the root node itself.
 */

void tree_free (tree *R) {
  ... // fill in the recursive code here to free R
}
```