CSE 333 Midterm Exam **Sample Solution** 5/10/13

Question 1. (18 points) Consider these two C files:

a.c

```
void f(int p);

int main() {
  f(17);
  return 0;
}
```

b.c

```
void f(char *p) {
  *p = 'x';
}
```

(a) Why is the program made from `a.c` and `b.c` incorrect? What would you expect to happen if it is executed?

The declaration of `f` in `a.c` is inconsistent with the actual function definition in `b.c`. Function `f` expects a valid pointer as its argument but is passed 17. It will treat this `int` as a pointer and will probably segfault when it tries to write 'x' to memory location 17.

(b) Will `gcc -Wall -c a.c` and `gcc -Wall -c b.c` give an error or will they successfully produce `a.o` and `b.o` without complaint?

Both will compile without error. (The code in `a.o` will assume that `f` has an integer parameter, but that does not prevent it from compiling successfully.)

(c) Will `gcc -Wall a.c b.c` give an error or will it successfully produce `a.out` without complaint?

The code will compile and link without problems. The linker does not have type information and, since `f` exists, files `a.o` and `b.o` can be combined to produce an executable program.

(d) How would you use standard C coding practices (using an extra file) to avoid the problems with this program (or at least detect them properly)? Give the contents of that extra file below and explain what modifications should be made to a.c and/or b.c, if any.

Put the following code in file `b.h`:

```
#ifndef B_H
#define B_H
void f(char * p);
#endif
```

Add the line

```
#include "b.h"
```

at the beginning of files `a.c` and `b.c`. Remove the declaration

```
void f(int p);
```

from the beginning of `a.c`.

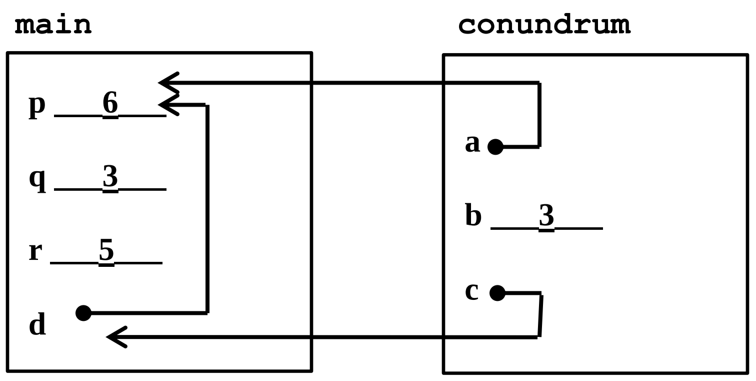
Question 2. (20 points) Consider the following rather twisted C program, which does compile and execute with no warnings or errors:

```
#include <stdio.h>

int *conundrum(int *a, int b, int **c) {
  **c = b + *a;
  *c = a;
  **c = *a * b;
  // HERE
  return a;
}

int main() {
  int p = 2, q = 3, r = 7;
  int *d = &r;
  d = conundrum(&p, q, &d);
  printf("%d %d %d %d\n", p, q, r, *d);
  return 0;
}
```

(a) Draw a boxes 'n arrows diagram showing state of memory when control reaches the comment containing `HERE`, right before executing the `return` statement in function `conundrum`. Your diagram should have two boxes showing the stack frames for functions `main` and `conundrum`. The stack frames should include values of integer variables and an arrow from each pointer to the location that it references. Then answer part (b) at the bottom of the page.



(Many people drew a single diagram of the stack showing the frame for `main` above the frame for `conundrum`. That is also a good solution and received full credit if done properly.)

(b) What output does this program produce when it is executed?

6 3 5 6

(We gave generous partial credit for incorrect answers to this part if they were consistent with the diagram in part (a)).

Question 3. (22 points) The nodes in a linked list of C strings can be defined as follows:

```
typedef struct snode {
  char * str;          // this node's heap-allocated string
  struct snode * next; // next node in the list or NULL
} Snode;
```

Complete the definition of function `Clone` below so that it returns (a pointer to) an exact duplicate of the list that is its argument, including duplicates of all the nodes and strings in the original list. You may use `strcpy` instead of `strncpy`, and may assume that all strings in the original list are properly `\0`-terminated. You may assume that `malloc` will always successfully allocate data when it is called. Also assume that all necessary library header files have already been `#include`d. Hint: `strcpy(dst,src)` copies the string `src` to the string `dst`.

```
// return a clone of the linked list with first node lst
// (which may be NULL)
Snode * Clone(Snode * lst) {

  // return NULL if at end of list
  if (lst == NULL) {
    return NULL;
  }

  // create a new node that is a clone of
  // the first node in the list
  Snode * dupnode = (Snode *)malloc(sizeof(Snode));
  dupnode->str = (char *)malloc(1+strlen(lst->str));
  strcpy(dupnode->str, lst->str);

  // next field of new node is clone of rest of the list
  dupnode->next = Clone(lst->next);

  // return new node, which is head of the cloned list
  return dupnode;

}
```

(Many solutions used a loop to advance through the original list creating a clone of each node along the way. If done properly that was also a good solution, and has the advantage of not needing stack space proportional to the length of the list for the recursive function calls.)

Question 4. (22 points) For this problem consider the C++ class `Vec` on the next page. This class is supposed to implement a simple vector of integers. All of the code is written in the class declaration instead of in separate `.h` and `.cc` files to get it to fit on one page for the exam.

The class has a constructor, copy constructor, assignment, methods to set and get individual elements from the vector, and a destructor. The assignment and copy constructor operations are supposed to make a complete copy (clone) of their argument.

The class compiles without any errors or warnings, but when it is used in a program it generally segfaults, and, even if it doesn't crash, valgrind reports all sorts of memory management problems.

Mark the code on the following page to identify the problems, and write in corrections so the class will work as intended. Keep your notes brief and to the point.

You should ignore possible problems with invalid index values or length arguments – i.e., assume the length provided to the constructor is positive and that the index arguments to `get` and `set` are within bounds (i.e., `0 <= index < len_`). You can also assume that heap allocation (`new`) always succeeds. (There are enough other problems in the code without worrying about these possibilities. ☺)

You can use the space below if you need extra room to write explanations or corrections, but *please* help the graders by making it easy to read your changes and figure out where they fit.

The main problem with the code is that the copy constructor and assignment operator do not copy the underlying array. That creates problems when the destructor executes a `delete [] v_` operation, which could easily be a double delete of a dangling pointer if the array was previously shared with another `Vec` object. (If the intent really was to have multiple `Vec`s share a single array, additional reference-counting code or something similar would be needed to delete each array when the last `Vec` object with a pointer to it was deleted.)

There are two other problems. First, the assignment operator does not check for self-assignment and will accidentally delete the underlying array before copying it if a `Vec` object is assigned to itself. Second, the `delete` operations omitted the brackets (`[]`) needed to delete arrays.

Corrections shown in green on the next page. Even better would be to create a `copyFrom` function to copy instance variables from another `Vec` object and call it from both the copy constructor and assignment operators. But for the exam that would not be expected.

Question 4. (cont.) Find the bugs in the C++ code below, give a ***very*** brief description of the problems, and correct the code so it works properly.

```
class Vec {  // a vector of integers
 public:

  // initialize new Vec with n elements all set to 0. Assume n>0.
  Vec(int n) {
    v_ = new int[n];
    len_ = n;
    for (int i=0; i<len_; i++) v_[i] = 0;
  }

  // copy constructor - initialize *this to be a clone of other
  Vec(const Vec &other) {
    ~~v_ = other.v_;~~                                 // delete
    len_ = other.len_;
    v_ = new int[len_];                               // add
    for (int i=0; i<len_; i++) v_[i] = other.v_[i];   // add
  }

  // destructor - free resources
  ~Vec() { delete [] v_; }                            // fix

  // replace the contents of *this with a clone of rhs
  Vec &operator=(const Vec &rhs) {
    if (this == &other)                               // add
      return *this;                                   // add
    delete [] v_;                                     // fix
    ~~v_ = rhs.v_;~~                                   // delete
    len_ = rhs.len_;
    v_ = new int[len_];                               // add
    for (int i=0; i<len_; i++) v_[i] = other.v_[i];   // add
    return *this;
  }

  // get/set functions. Assume that 0<=index<len_ (i.e., for this
  // question don't worry about index out of bounds problems)
  int  get(int index) const  { return v_[index]; }
  void set(int index, int n) { v_[index] = n; }

 private:
  int* v_;     // array of int values allocated on the heap
  int len_;    // number of ints in v_
};
```

Question 5. (18 points) A few short questions to wrap up. For each question circle the correct choice. You do not need to justify your answers.

(a) If `a` is declared as `char a[5]`, then `a[3]==*(a+3)` is always true. (circle)

[✓ True] False

(b) If `a` is declared as `int32_t a[5]`, then `a[3]==*(a+12)` is always true. (Recall that a `int32_t` value occupies 4 bytes.) (circle)

True [✓ False] **(`*(a+12)` is the same as `a[12]`)**

(c) If we execute program `xyzzy` using the command line `./xyzzy one two`, then the value of `argv[0]` in the main program will be the string `one`. (circle)

True [✓ False] **(`argv[0]` is `./xyzzy`)**

(d) If on our Linux system a program contains a pointer variable declared as `int *p` and in the debugger we see that the value of `p` is `0x7fffffffe318`, we can conclude that: (circle the best choice)

(i) `p` refers to a variable allocated on the heap

[✓ (ii) `p`] refers to a local variable in some function's stack frame

(iii) `p` refers to constant data or to a location in the program's x86 machine code

(iv) `p` is `NULL`

(v) we cannot conclude anything about the data referenced by `p`

(e) The system-level `read` function normally returns the number of bytes read. If it returns the value -1 (error), then the program should not attempt any further I/O operations on that stream because it would be a fatal error to do so. (circle)

True [✓ False] **(The error might signal an incomplete read)**

(f) The `stat` system function returns information about a file. That information includes a field `st_mode` that, among other things, describes the type of the file. All of these files are classified as "regular files" as opposed to some other type: ascii text, Word .docx files, jpeg picture files, compiler produced `.o` files, and executable files like `a.out`. (circle)

[✓ True] False