CSE 374 Final Exam Solutions, March 20, 2014

Name: ________KEY____

Write your name in the space provided above.
Do not write your ID number or any other confidential information on this page.

Please wait to turn the page until everyone is told to begin.

While you are waiting, please read the following information:

- There are 9 questions on 13 pages worth a total of 100 points. Please budget your time so you get to all the questions. Keep your answers brief and to the point.
- Some question pages may be detached for your convenience (a separate page is provided for your answer in these cases). A stapler is available at the instructor podium if your entire exam falls apart.
- The exam is closed book, closed notes, closed electronics, closed Internet, closed neighbor, closed telepathy, etc., with the exception of the one instructor-provided double-sided page of notes.
- Many of the questions have short solutions, even if the question is somewhat long. Don't be alarmed.
- If you don't remember the exact syntax of some command or the format of a command's output, make the best attempt you can. We will make allowances when grading. Write legibly.
- Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin.

Score: ____________ / 100

#	Score	Points	Topic
1.	__________	/ 8	debugging, `gdb` (short answer)
2.	__________	/ 14	automating compilation, dependencies, `make` (draw a graph, write a `makefile`)
3.	__________	/ 8	version control, `svn` (matching)
4.	__________	/ 12	C programming, pointers vs. what's being pointed at (write a small amount of C code)
5.	__________	/ 8	C program output (pointers as parameters) (similar to midterm, what does this program output?)
6.	__________	/ 8	C program evaluation, linked lists and strings, `free` (matching)
7.	__________	/ 18	C programming, linked lists and strings, `malloc` (write C code)
8.	__________	/ 12	C programming, memory management (write C code)
9.	__________	/ 12	C programming, memory management (write C code)

Question 1. (8 points) (debugging, gdb)

You are trying to fix a C program that is crashing for some reason. You have narrowed the problem down to a set of functions that implement a list of strings. The four functions involved are:

```
init();             // must call first before any others
add(char *s);       // add s to the list
delete(char *s);    // delete s from the list if present
size();             // return number of strings in the list
```

The list package requires that the function `init()` be called before any of the other three functions can be used successfully. Your guess is that somehow one of the other functions is being called first, but your boss wants you to prove your guess is right before spending any more time on the problem.

Explain how you would use a debugger like `gdb` to discover whether one of the other functions is called before `init()`. You may not modify the source code (i.e., you can't insert print statements or anything like that – you have to work with the source code as it exists); however, you may recompile existing code if needed.

- Recompile and link with –g flag to ensure debugging information is available
- Open program with gdb and set breakpoints at the beginning of each function
- Run the program within gdb. If the first breakpoint is reached and it is at the beginning of a different function than `init()`, you have confirmed your guess

Question 2. (14 points) (automating compilation, dependencies, make)

Suppose we have the following collection of C header and implementation files:

```
--------------------
thingie.h
--------------------
#ifndef THINGIE_H
#define THINGIE_H
. . .
#endif
```

```
--------------------
doodad.h
--------------------
#ifndef DOODAD_H
#define DOODAD_H
. . .
#endif
```

```
--------------------
thingie.c
--------------------
#include "thingie.h"
. . .
```

```
--------------------
doodad.c
--------------------
#include "doodad.h"
. . .
```

```
--------------------
whatsit.c
--------------------
#include "thingie.h"
#include "doodad.h"

int main() {
  . . .
}
```

These source files are to be used to build an executable program file named `whatsit`, whose `main` function is in the source file `whatsit.c` and which uses all of the functions defined in all of the above source files.

Answer the questions (2 parts) on the next page using the above information. (Suggestion: sketch your answer to part (a) below before you make a clean copy of it on the next page.)

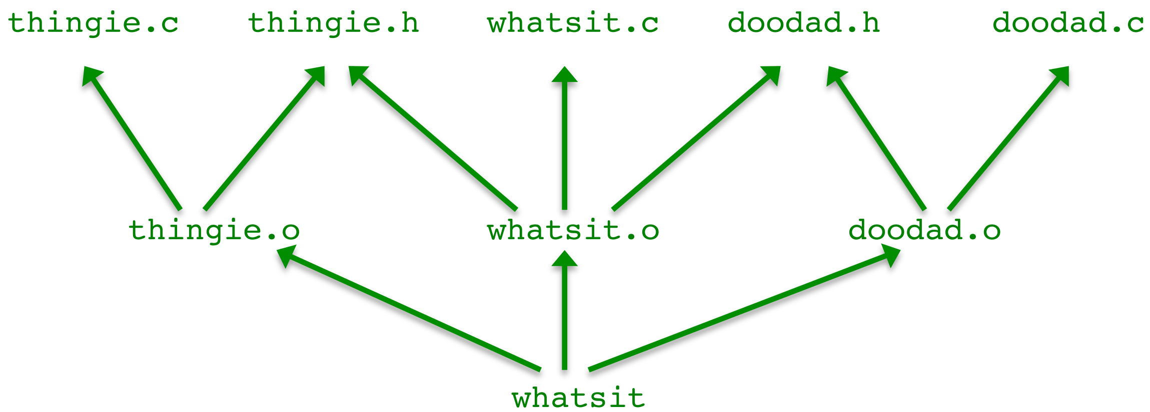
Please write your answer on the next page.

You may detach this page from the exam if that is convenient.

Question 2. (continued)

Part (a) Recall that we can specify the **dependencies** between files in a program using a graph, where there is an arrow drawn from each file name to the file(s) it depends on. For example, the drawing to the left shows how we would diagram an executable program named `foo` that depends on (is built from) `foo.o`, which in turn depends on `foo.c`.

In the space below, draw a **graph** (diagram) showing the dependencies between the executable program `whatsit` and all of the source (`.c`), header (`.h`), and compiled (`.o`) files involved in building it, based on the file information given on the previous page.



Part (b) Write the contents of a **`Makefile`** whose default target builds the program `whatsit`, and which only recompiles individual files as needed. Your `Makefile` should reflect the dependency graph you drew in part (a). Compile with flags for all warnings and for including debugging information to use with `gdb`.

```
whatsit: whatsit.o thingie.o doodad.o
     gcc –Wall –g –o whatsit whatsit.o thingie.o doodad.o

whatsit.o: whatsit.c thingie.h doodad.h
     gcc –Wall –g –c whatsit.c

thingie.o: thingie.c thingie.h
     gcc –Wall –g –c thingie.c

doodad.o: doodad.c doodad.h
     gcc –Wall –g –c doodad.c
```

Question 3. (8 points) (version control, svn)

Match each svn command with the best description

Command	Answer	Description
A. Update	E	Performed once by one developer to set up the project in the repository initially
B. Add/Delete	D	Uploads changed or new files to the repository, merging changes into the repository if needed
C. Checkout	A	Downloads changed or new files in a project to the local "working directory", merging changes into local files if needed
D. Commit	B	Schedules files to be included or removed from the project repository
E. Import	C	Sets up a local "working directory" and downloads an entire copy of the project from the repository to it

Question 4. (12 points) (C programming, pointers vs. what's being pointed at)

Write a C function `are_same` that has two parameters of type `int*` and returns:

- 0 if the two parameters point to locations holding different `int` values,
- 1 if the two parameters point to different locations holding the same `int` values,
- 2 if the two parameters point to the same location

```
int are_same(int *x, int *y) {

     if (x == y) {
          return 2;
     }
     else if ((*x) == (*y)) {
          return 1;
     }
     else {
          return 0;
     }
```

Don't deduct points for extra casts.

```
}
```

Question 5. (8 points) (C program output, pointers as parameters)

Consider the following C program:

```
#include <stdio.h>

void mysterious(int *a, int *b, int *c) {
     *a = *c;
     *b = *b + *a;
     *c = *a - *b;
}


int main() {
     int w = 5;
     int x = 1;
     int y = 3;
     int z = 2;
     mysterious(&x, &y, &w);
     printf("%d %d %d %d\n", w, x, y, z);
     mysterious(&w, &w, &z);
     printf("%d %d %d %d\n", w, x, y, z);
     return 0;
}
```

What output does this program produce when it is executed? (It does execute successfully.) It may be useful to draw diagrams showing variables and pointers to help answer the question and help us award partial credit if needed.

```
-3 5 8 2
4 5 8 0
```

Question 6. (8 points) (C programming, linked lists and strings, `free`)

Consider the following definition of a node of a linked list of nodes containing strings in C:

```
struct node {
     char *s;
     struct node *next;
}
```

and three functions that allegedly deallocate (`free`) the space occupied by a list:

```
void free_list_1(struct node *lst) {
     if (lst == NULL)
          return;
     free(lst);
}
```

A

```
void free_list_2(struct node *lst) {
     if (lst == NULL)
          return;
     free(lst->s);
     free_list_2(lst->next);
     free(lst);
}
```

```
void free_list_3(struct node *lst) {
     if (lst == NULL)
          return;
     free(lst);
     free(lst->s);
     free_list_3(lst->next);
}
```

B

In the box provided next to each `free_list_#` function above, write zero or more of the following letters (A, B) if the corresponding description provided below matches what is going on in that function.

A. Causes a memory *leak* because it only frees the first `node struct` in the list, without freeing any of the strings or remaining `nodes`.

B. Uses a pointer to a `struct` after it has already been freed, which is considered a *dangling pointer*. Attempts to access other pointers inside the freed `struct`.

Question 7. (18 points) (C programming, linked lists and strings, `malloc`)

A classic data structure is a *linked list*. For this problem, we will work with a linked list whose nodes contain strings, or, more precisely, the data in each node is a pointer to a '`\0`'-terminated C string. Assume a list node is defined as follows:

```
struct list_node {
     char *str;
     struct list_node *next;
};
```

You may assume the list does not contain any duplicates and the strings are not sorted in any particular order. You are to write code to insert a new string `s` into a linked list. You may insert the copy of `s` anywhere in the list if you need to add it. The list might be empty initially, in which case you should create a single node pointing to a copy of `s` and return a pointer to that new node as a result. If the string `s` already appears in the list, the function should return a pointer to the original, unmodified list.

Example: The following statement would add `"xyzzy"` to the list whose first node is `words` if `"xyzzy"` is not already in the list;

```
words = insert("xyzzy", words);
```

Write your code in two steps in Parts (a) and (b) below and on the next page:

Part (a) (8 points) Complete the definition of the following function `new_node` so that it returns a pointer to a newly heap-allocated `list_node` that references a newly heap-allocated copy of the given string. Some useful `#includes` are provided for you.

```
#include <string.h>
#include <stdlib.h>

struct list_node *new_node(char *s) {

     struct list_node *answer;
     answer = (struct list_node *) malloc(sizeof(struct
     list_node));
     answer->str = (char *) malloc(strlen(s) + 1);
     strcpy(answer->str, s);
     answer->next = NULL;
     return answer;

}
```

Question 7 (continued).

Part (b) (10 points) Now, implement the following function to insert a new heap-allocated copy of a string s into linked list l. **(If a new node is needed to hold s, use the new_node function from part (a) to create it.)** If string s already occurs in the linked list, the function should not change the list. The function should return a pointer to the head of the (possibly modified) list l. You may assume that all strings are properly \0-terminated and you do not need to worry about overrun errors (i.e., it's ok to use strcmp instead of strncmp, etc.).

```
/* insert a copy of string s into the linked list l        */
/* and return a pointer to l.  If s already appears in l,  */
/* do not change l and just return a pointer to it.        */

struct list_node *insert(char *s, struct list_node *l) {

     if (l == NULL) {
          l = new_node(s);
          return l;
     }

     struct list_node *temp = l;
     int found = 0;

     while (temp != NULL) {
          if (strcmp(temp->str, s) == 0) {
               found = 1;
          }
          temp = temp->next;
     }

     if (found) {
          return l;
     }
     else {
          temp->next = new_node(s);
          return l;
     }

}
```

Question 8. (12 points) (C programming, memory management)

For this problem, assume that the following `struct` defines the layout of the header part of each free list node (**note that the `size` field here is slightly different from HW6**):

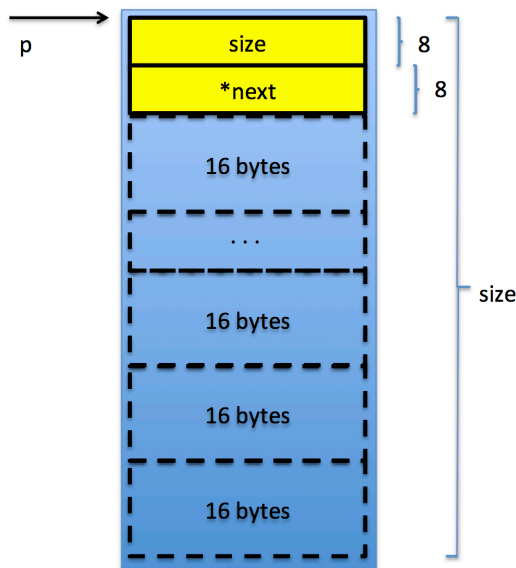
```
struct free_node {
     uintptr_t size;           // number of bytes in this node,
                               //   including this header

     struct free_node *next;   // next node on the free_list,
                               //   or NULL if no more nodes
};
```

Assume as in HW6 we are dealing with sizes in multiples of 16 bytes. In a properly formed free list, successive nodes should occupy **increasing** memory addresses. If some node on the list has a `next` pointer that is not `NULL` and contains a lower memory address, then something is wrong with the list.

Complete the definition of function `looks_ok` on the next page so that it returns *true* (`1`) if the successive nodes on list `p` are stored at increasing addresses, and returns *false* (`0`) if some node has a successor with an address that is less than the address of the node itself.

Use the following diagram of a block on the free list for Questions 8 and 9:



Please write your answer on the next page.

You may detach this page from the exam if that is convenient.

Question 8 (continued). Free list node definition repeated for reference:

```
struct free_node {
     uintptr_t size;           // number of bytes in this node,
                               //   including this header

     struct free_node *next;   // next node on the free_list,
                               //   or NULL if no more nodes
};
```

Write your code for this function below:

```
/* return true (1) if the node addresses on list p are */
/* strictly increasing, otherwise return false (0)     */

int looks_ok(struct free_node *p) {

     /* an empty free_list is ok */
     if (p == NULL) {
          return 1;
     }

     /* non-empty free_list; check p against p->next */
     /* as long as p->next != NULL */
     while (p->next != NULL) {
          if ((uintptr_t) p >= (uintptr_t) p->next)) {
               /* p->next has a smaller address – bad! */
               return 0;
          }
          p = p->next;
     }
     /* by now have reached the end without problems */
     return 1;
```

Allow either > or >= in comparing addresses
Allow recursive solutions if correct

```
}
```

Question 9. (12 points) (C programming, memory management)

In this problem, we would like to write a function to help us analyze the memory manager free list. You should assume that the following `struct` gives the layout of the header part of each free list node (**note that the `size` field here is slightly different from HW6**):

```
struct free_node {
     uintptr_t size;          // number of bytes in this node,
                              //   including this header

     struct free_node *next;  // next node on the free_list,
                              //   or NULL if no more nodes
};
```

Complete the following function so it scans the entire free list and prints the maximum address **occupied** by any part of any block on the free list. (Note that the maximum address occupied by a block is not the address of the header node or the beginning / ending boundary of the block.) For this problem, **assume that the free list blocks might not be sorted properly**, so the maximum address might be found in a block anywhere on the list. If the free list is empty, the function should print `0x00000000` for the maximum address.

Hint: "`%p`" is a suitable `printf` format string to print a `uintptr_t` value using 8 hex digits with "`0x`" in front.

```
void print_max_address(struct free_node *free_list) {

     uintptr_t max_addr;
     struct free_node *p;
     uintptr_t end_addr;

     max_addr = 0;
     p = free_list;

     while (p != NULL) {
          end_addr = ((uintptr_t)p) + p->size – 1;
          if (end_addr > max_addr) {
               max_addr = end_addr;
          }
          p = p->next;
     }
     printf("%p", max_addr);


}
```