CSE 303 Final Exam

December 11, 2007

Name ___________Sample Solution___________

The exam is closed book, except that you may have a single page of hand-written notes for reference.

If you have questions during the exam, raise your hand and someone will come to help. Stay seated.

Please wait to turn the page until everyone has their exam and you have been told to begin.

1	/ 6
2	/ 5
3	/ 14
4	/ 10
5	/ 5
6	/ 10
7	/ 12
8	/ 12
9	/ 12
10	/ 14
Total	/ 100

Question 1. (6 points) As we saw in class, it is possible to implement true object-oriented programming in C, even though it requires writing out some programming details that are handled automatically in C++ or Java. Suppose we have the following partial C++ class specification:

```
class Example {
  public:
     void do_something(int k);

  // remaining methods and data omitted
};
```

In a C implementation, the function do_something would have the following declaration:

```
void Example_do_something(Example * this, int k);
```

What is the purpose of the additional "this" parameter? Is it really needed? If so, why? (Be brief – one or two sentences are probably enough to answer the question.)

The primary reason is so that when the function (method) is called it can access the fields of the object associated with the call. Without this the function has no way to locate the appropriate instance of the Example "class" that it should act on.

Question 2. (5 points) Suppose we want to write a function mumble that has a function pointer as a parameter. In the prototype for function mumble,

void mumble (______________________________);

what should we put in place of the blank line to declare that parameter f is a pointer to a function which, when called, accepts two integer arguments and returns a double as a result? (circle)

(a) double * (f) (int, int)

(b) double f (int, int)

(c) double (*f) (int *, int *)

(d) double (*f) (int, int) [circled]

(e) void (*f) (int, int, double)

Question 3. (14 points) gcc is broken! More to the point, someone deleted gcc from the system directories. But the C preprocessor, compiler, and loader are all still there. You have to get a small project done by tomorrow morning and gcc won't be fixed by then. But with a small bit of work, it should be easy enough to write a makefile that will compile and build your program. Here are the details of what needs to be done.

The program consists of three source files: main.c, f.c, and f.h. The header file f.h is included by both f.c and main.c. The makefile should contain separate targets to build main.o and f.o from the respective .c files and should contain a default target to build an executable program named mumble from these compiled files.

The difference is that we can't use the gcc command to compile the files. Instead we can use cpp (the C preprocessor), cc1 (the C compiler), and ld (the loader) individually. These commands work as follows:

cpp infile outfile	read from infile, perform preprocessing, and write the results to outfile (infile is a .c file, outfile can have any name, but it should end with an extension .x so as not to be confused with other files)
cc1 infile outfile	read a processed C program generated by cpp and compile it, writing the result to outfile (which normally has a .o filename)
ld -o outfile files	read multiple .o files and produce an executable file named outfile

Write a makefile on the next page to build the executable program mumble from main.c, f.c, and f.h. The makefile should update any out-of-date files when it runs, but should not do any unnecessary work (i.e., it should not recompile files that are already up to date, for example – but if you do need to recompile a file, just call both cpp and cc1 and don't worry about whether or not the preprocessor output is different than it was before). Also don't worry about additional compiler or loader options like -g; those are not the point of this question.

(write your answer on the next page)

Question 3. (cont) Your makefile here.

```
mumble: main.o f.o
      ld -o mumble main.o foo.o

main.o: main.c f.h
      cpp main.c main.x
      cc1 main.x main.o

f.o: f.c f.h
      cpp f.c f.x
      cc1 f.x f.o
```

Question 4. (10 points) The dreaded C++ "what does this print?" question. Suppose we have the following C++ class declarations and implementations:

```
#include <iostream>
using namespace std;

class B {
public:
  virtual void p();
  virtual void q();
};

void B::p() { q(); }
void B::q() { cout << "B::q" << endl; }

class D: public B {
  virtual void q();
};

void D::q() { cout << "D::q" << endl; }

int main() {
  B * ptr = new D();
  ptr->p();
  return 0;
}
```

(a) What output is produced when this program is executed?

D::q

(b) What output is produced if we run the same program, except with all of the occurrences of the word "virtual" deleted from the code?

B::q

Question 5. (5 points) What output is produced when the following C program is executed?

```
#include <stdio.h>

#define PROD(x,y) (x*y)

int main() {
  int a=2;
  int b=3;
  int c=4;
  printf("%d\n", PROD(a+b,c));
  return 0;
}
```

Output: ________**14**__________

Question 6. (10 points) Your program has just crashed with a segmentation fault. Give a concise step-by-step explanation of how you could use gdb to figure out where the problem is and what the cause is (bad pointer value, stack overflow, etc.).

The main idea was to describe how you would run the program under gdb, then look around after the crash to see what went wrong. In more detail:

1) Compile the program using the –g option if not done already, then start gdb with the executable file as its argument.

2) Enter the gdb run command with any desired arguments to start the program.

3) When the program crashes, use the error message and commands like bt (backtrace) and list to find the location of the crash in the source program.

4) Use commands like print or display to find the values of program variables and other expressions.

Question 7. (12 points) Concurrency. Consider the following code, which maintains a linked list of integer values.

```
struct int_node {           // nodes for an integer list:
  int val;                  //   value in this node
  struct int_node * next;   //   next node in list or NULL
};

struct int_node * list = NULL;  // list of nodes; NULL if empty

/* place new_node on the front of the list */

void add_node(struct int_node * new_node) {
  atomic {
  new_node->next = list;
  list = new_node;
  }
}

/* delete first node on the list and return it; return NULL if the list is empty */

struct int_node * get_node() {
  atomic {
  struct int_node * temp = list;
  if (list == NULL) return NULL;
  list = list->next;
  }  // could also appear after the return statement – doesn't matter
  return temp;
}
```

Suppose that this code is used in a program where there are two threads t1 and t2, both of which are adding and deleting nodes from the (shared) list by calling the add_node and get_node functions.

(a) This code is not thread-safe. Give a brief explanation of what can go wrong if the two threads are both using these functions concurrently. (You should assume that neither thread modifies a node while it is stored on the list – the problem is elsewhere.)

There were many good explanations in the answers. The common theme is that if more than one thread is executing add_node and get_node at the same time, they can see inconsistent values of variable list when the list is partially updated, leading to updates that effectively vanish, or to a mal-formed list, etc.

(b) Good news! It turns out that the UW CSE research program on concurrency has been successful, and the new C standard has added an atomic statement. A sequence of statements s1, s2, …, sn will be executed atomically if we surround them with atomic{ … } as follows: atomic{ s1, s2, …, sn }.

Add atomic statements to the above code so that the resulting code is thread-safe. **See above.**

Question 8. (12 points) The nodes in a binary tree containing string values can be defined as follows in a C program:

```
struct tnode {            // node for a binary tree of strings:
  char * str;             //   string in this node (on the heap)
  struct tnode *  left;   //   left subtree; NULL if empty
  struct tnode * right;   //   right subtree; NULL if empty
};
```

We assume that the string values pointed to by str are dynamically allocated on the heap, as are the tree nodes (tnodes) themselves.

For this problem, complete the following function so that it correctly frees all of the dynamic storage allocated to the tree that is its argument. You should add appropriate statements before or after the lines of code in the function below; you ***may not*** alter the existing code or add additional code or functions to traverse the tree structure. Hint: be sure to free both the nodes and the strings, but be careful to avoid memory leaks, dangling pointers, multiple frees, and similar problems.

```
/* free all nodes and strings in the tree with root t */
void delete_tree(struct tnode * t) {


  if (t == NULL) {

    return;

  } else {

    free(t->str);  // could appear anywhere before the free(t) statement

    delete_tree(t->left);

    delete_tree(t->right);

    free(t);

  }

  return;

}
```

Question 9. (12 points) One of your friends is looking for some advice about how to use version control systems; subversion in particular.

(a) The two commands svn commit and svn update both sound like they do something useful involving files in the repository and in the local working copy. What do they do and how do they differ? (Be brief)

svn commit stores local changes in the repository. svn update brings the local copy up to date to reflect any changes in the repository since the last time the two were synchronized.

(b) In a programming project, there are many files, some of which should be stored in a repository like subversion, and some of which should not. For each of the following, circle yes if it should be stored in the project repository; circle no if it should not normally be stored.

(yes) no C/C++/Java source files (.c, .cc, .java)

yes (no) compiler-produced object code files (.o)

yes (no) Java compiler-produced class files (.class)

(yes) no header files (.h)

(yes) no makefiles

yes (no) executable files (a.out or executables with other names)

yes (no) emacs backup files (files with names ending in ~, like foo.c~)

Question 10. (14 points) A little storage management.

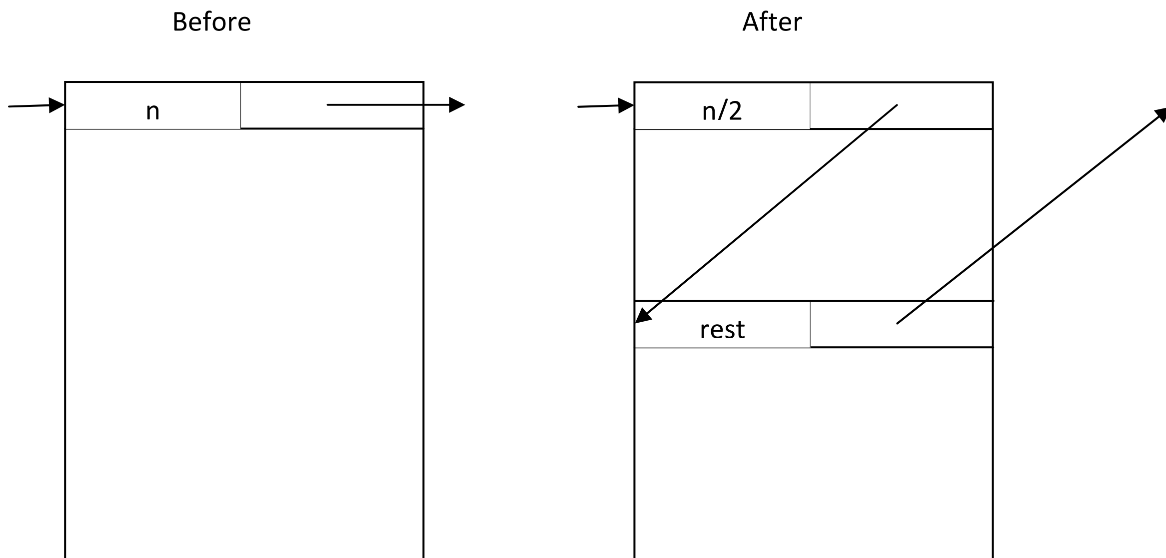
A typical way to represent the free list in the getmem/freemem storage allocator is as a linked list of blocks that begin with the following struct:

```
struct free_block {
  int size;                    // number of bytes in this block,
                               //      including this header
  struct free_block * next;   // next block on the free list
};
```

Although different people used different conventions in the actual project, for the purposes of this problem, assume that the size in a free_block header includes both the free_block struct itself plus the rest of the block, and that it is the total number of bytes.

One of the operations needed in the storage allocator is to split a free block into two smaller blocks so that one of the resulting blocks can be used to satisfy a storage request, while the other block remains on the free list. For this problem you should implement a similar, but not identical, function.

Implement the function split, on the next page, so it takes a free block and splits it into two halves, leaving both resulting blocks on the free list. In pictures, the input is a block like the one on the left; the result should look like the picture on the right.



In other words, the size in the original header should be half the size it was to begin with, its next pointer should point to the new header in the middle of the block, and the new header should point to the block that was previously next on the list. The original block is aligned on an 8-byte boundary; both resulting blocks should also be aligned on 8-byte boundaries.

You may assume that n (the original size) is a large enough multiple of 8 so that both resulting blocks will be large enough to contain a header plus significant amounts of additional storage.

Question 10. (cont) Complete the definition of function split below. The definition of the free_block struct is repeated here for convenience.

```
struct free_block {
  int size;                  // number of bytes in this block,
                             //     including this header
  struct free_block * next;  // next block on the free list
};

/*
 * Split the block with header *p into two blocks, both half the size of the
 * original block.  Set p->next to point to the new block and set the new block's
 * next field to point to the original value from p->next.  Return a pointer to
 * the original block as the function's value.
 */
struct free_block * split(struct free_block *p) {

  // calculate sizes - first block is largest multiple of 8 not greater than n/2
  // second block is whatever is left [ok if the sizes are reversed, not ok to
  // have a memory leak of part of an 8-byte block or to lose 8-byte alignment.]

  int size1 = (p->size / 2) / 8 * 8;
  int size2 = p->size - size1;

  // calculate location of new block header

  int ip = (int) p;
  ip += size1;
  struct free_block *p2 = (struct free_block *) ip;

  // set fields in new block (must do before altering original block)

  p2->size = size2;
  p2->next = p->next;

  // adjust original block size and set its next pointer to refer to the new block

  p->size = size1;
  p->next = p2;

  // return pointer to original block as the result

  return p;

}
```