CSE 303 Final Exam

December 9, 2008

Name _____Sample Solution_____

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

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	/ 8
2	/ 8
3	/ 14
4	/ 14
5	/ 17
6	/ 14
7	/ 10
8	/ 15
Total	/ 100

Question 1. (8 points) Several terms were introduced to talk about how thoroughly a set of tests checked a program. One term was *statement coverage*, which was a measure of the percentage of statements in the code executed by a set of tests. Two other measures were *branch coverage* and *path coverage*. Briefly describe what those two terms mean and how they differ.

Branch coverage = how thoroughly the tests cause each branch in the program to be executed in both directions.

Path coverage = how thoroughly the tests exercise each possible execution path through the program.

Path coverage is a superset of branch coverage in the sense that it measures all possible combinations of branches through the code instead of just measuring whether each branch is executed in both directions.

Question 2. (8 points) C++ includes something called a copy constructor, which is not the same as assignment (operator=). What is the essential difference between the copy constructor and the assignment operator?

A copy constructor stores a value in an uninitialized object, while assignment replaces the existing value of a previously-initialized object. Assignment might have to dispose of resources belonging to the object or otherwise clean up the existing state before changing it, while a copy constructor cannot and should not do this, since the object does not have a well-defined state when the copy constructor begins execution.

A number of answers said specific things about the behavior of the copy constructor or assignment, for instance a common one was that "assignment copies the pointer". While it's true that assignment in Java is just pointer assignment, in C++ both assignment and the copy constructor can perform any computation, depending on the code. The essential difference is whether we are initializing a new object, or changing the value of one that is already initialized. (And, even then, it is possible that an assignment operator could actually do nothing at all – although that is unlikely to be useful.)

Question 3. (14 points) To demonstrate `make` we used a little program that had several functions to print messages, and a main program that called them. The program had header files `speak.h` and `shout.h`, and implementation files `speak.c, shout.c,` and `main.c`.

In this question, we want to modify the program by adding two new files, `babble.h` and `babble.c`, which are as follows:

`babble.h`

```
#ifndef BABBLE_H
#define BABBLE_H

/* utter string s n times */
void babble(char * s, int n);

#endif
```

`babble.c`

```
#include "babble.h"
#include "speak.h"

/* utter string s n times */
void babble(char * s, int n) {
  int k;
  for (k = 0; k < n; k++) {
    speak(s);
  }
}
```

We also will modify `main.c` to `#include "babble.h"` in addition to the other files it already includes, and to call `babble` with appropriate arguments.

(Question continued next page; you can remove this page for reference if it makes things easier.)

Question 3. (cont.) After the new `babble.c` and `babble.h` files are created, and `main.c` is modified to use them, we need incorporate these changes into our project.

(a) Here is the original Makefile for the project. Indicate the changes that need to be made to handle these additions and modifications properly.

(Changes shown in green. Basically, we need to add dependencies on the new files, and add a command to compile it.)

```
talk: main.o speak.o shout.o babble.o
	gcc -Wall -g -o talk main.o speak.o shout.o babble.o

speak.o: speak.c speak.h
	gcc -Wall -g -c speak.c

shout.o: shout.c shout.h speak.h
	gcc -Wall -g -c shout.c

main.o: main.c speak.h shout.h babble.h
	gcc -Wall -g -c main.c

babble.o: babble.c babble.h speak.h
	gcc –Wall –g –c babble.c
```

(b) Assuming that this project was originally checked out from a svn repository, what svn commands need to be issued to bring the repository up to date and incorporate all of the changes made to include `babble` in the code?

Need to add the new files to the project, then commit the changes (including a commit message):

```
svn add babble.h babble.c
svn commit –m “add babbling”
```

Question 4. (14 points) The dreaded C++ "what does this print?" question. Suppose we have the following C++ classes and main program:

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

class A {
public:
  virtual void p() { cout << "A::p" << endl;  q(); }
          void q() { cout << "A::q" << endl;  r(); }
  virtual void r() { cout << "A::r" << endl; }
};

class B: public A {
public:
          void q() { cout << "B::q" << endl; r(); }
  virtual void r() { cout << "B::r" << endl; }
};

int main() {
  cout << "first part:" << endl;
  A* x = new B();
  x->p();

  cout << "second part:" << endl;
  B* y = new B();
  y->p();
  y->q();

  delete x;
  delete y;
  return 0;
}
```

(In the above code, the function implementations are included as part of the classes, instead of being written as separate functions elsewhere. This is legal C++ and compiles and executes without errors.) What output is produced when this program is executed?

```
first part:
A::p
A::q
B::r
second part:
A::p
A::q
B::r
B::q
B::r
```

Question 5. (17 points) Your partner Ben Bitblitter decided that the best way to handle combining adjacent blocks on the free list of the memory allocator was first to put blocks on the list in order of ascending addresses, but not combine them with adjacent blocks right away. Then he decided to write a separate function to go through the free list and combine any adjacent blocks that are there. Since this function might not get called every time the free list is changed, it needs to deal with the possibility that several successive blocks on the free list are, in fact, adjacent, and that there may be more than one such group of adjacent blocks on the free list. Unfortunately, he dropped the class before implementing his brilliant idea, and it's up to you to get it to work.

The free list is a single linked list with nodes defined by the following structure:

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

The global variable free_list stores a pointer to the free list:

```
struct free_block * free_list;   // free list blocks; NULL if none
```

The blocks on the free list are stored in ascending order by block address.

Complete the definition of function `merge_free_blocks` on the next page so it scans the free list and combines all adjacent blocks that it can into larger blocks.

(Hint: Take a moment to think about what needs to be done before writing code. In particular, you only need to figure out how to recognize a pair of adjacent blocks and combine them, then repeat that if the combined block is itself adjacent to the next block on the free list. That may cut down on the number of separate cases that need to be handled in the code.)

(write your code on the next page)

Question 5. (cont.) Reminders: A free list node begins as follows:

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

The head of the free list is this global variable:

```
struct free_block * free_list;   // free list blocks; NULL if none
```

Write your code below.

```
/* Combine all groups of adjacent blocks on the free list into large blocks */
void merge_free_blocks() {

  // Strategy: Advance pointer p through the free list.  If p->next is adjacent
  // to p, combine p->next with p.  If p->next is not adjacent to p, advance p
  // to the next free list block.

  struct free_block * p = free_list;

  while (p != NULL) {

    if (p->next != NULL && ((int)p) + p->size == (int)(p->next)) {

      // combine with next block

      p->size += p->next->size;

      p->next = p->next->next;

    } else {

      // advance to next (non-adjacent) block

      p = p->next;

    }

}
```

Note: It is probably possible to omit the test for `p->next != NULL` in the `if` statement, but that relies on being able to safely cast `p->next` to an `int` and compare its value successfully with another integer value if it is `NULL`. Being somewhat paranoid, your instructor prefers not to rely on this, but we didn't deduct points if you did something similar.

Question 6. (14 points) Here is a simple fixed-size stack data structure, specified and implemented in a single C++ source file:

```
#define STACK_SIZE 1000

class IntStack {

public:

  // construct new empty stack
  IntStack() { size = 0; }

  // push n on stack if room and return true, otherwise return false
  int push(int n) {
    int result;
    if (size == STACK_SIZE) {
      result = 0;
    } else {
      items[size] = n;
      size++;
      result = 1;
    }
    return result;
  }

  // return top item on stack or 0 if stack is empty
  int pop() {
    int result;
    if (size > 0) {
      size--;
      result = items[size];
    } else {
      result = 0;
    }
    return result;
  }

private:
  int items[STACK_SIZE];  // stack items are in items[0..size-1] and
  int size;               //   items[size-1] is the "top" of the stack
};
```

(continued next page – do not remove this page from the exam)

Question 6 (cont.) Suppose we use an instance of this stack data structure in a concurrent program, and that the stack is accessed by more than one thread at a time. Individual statements or conditional tests are guaranteed to execute atomically, but execution may switch from one thread to another at any time between statements.

(a) The given code is not thread-safe. Give an example of how concurrent access by two threads to the same stack could result in an error.

Problems can occur if either push or pop is interrupted somewhere between checking `size` and finishing modifying the stack array and size. For instance, if two threads try to push something at the same time and execution switches from the first thread to the second one right after executing `items[size]=n` in the first thread, then both threads could wind up storing data in the same place on the stack but incrementing size twice. That would lose one of the items pushed on the stack, and leave an element of the stack initialized.

There are, of course, many possible answers to this question, but all of them involve switching threads in the middle of an operation that updates the stack, leaving `size` and/or the array in an inconsistent state.

(b) Explain how to best modify the given code so it can be executed safely and correctly by multiple threads. Either give an explanation here, or mark up the code on the previous page and summarize your changes below.

Use either something like the `atomic{...}` block discussed in class or locks to ensure that a thread cannot be interrupted between the start of either of the `if` statements and the last statement that changes either `items` or `size` in methods `push` and `pop`. The critical sections do not need to include the return statements or assignments to `result` that don't involve the stack, but there is no real harm if they do.

Question 7. (10 points) In both C and C++ code, the normal convention is that a header file named `foo.h` should begin with the preprocessor directives

```
#ifndef FOO_H
#define FOO_H
```

and end with the preprocessor directive

```
#endif
```

(a) Why is this done? (Be brief)

To ensure that the declarations and other things in the header file are processed only once during the compilation of any particular file.

(b) Given an example showing what can go wrong if these directives are omitted from all of the header files in a multi-file project.

Suppose that header file `foo.h` `#includes` another header `bar.h`, and that some other file `#includes` both `foo.h` and `bar.h`. If `bar.h` declares a function or a variable, then without the `#ifndef ... #endif` directives, the declaration will be processed twice, probably resulting in a compile-time error. With the directives, the contents of `bar.h` will be skipped the second time it is included, and the double-declaration error will not occur.

Question 8. (15 points) After having such a great time in CSE 303, you've been hired as a TA and you have office hours after the "pointers and dynamic allocation" project has been assigned. One of your students has come in with the following C code, which is supposed to add a new node to a linked list of C strings. The list does not need to be sorted, and can contain duplicates. But the code is rather badly broken – in fact, it won't even compile.

Show how to fix the following code so that `new_node` returns a properly initialized new node with a copy of the string, and `insert` adds the new node to the front of the list.

```
#include <stdlib.h>

#include <string.h>      // needed for strcpy, strlen

struct strnode {         /* node in a linked list of strings:          */
  char * str;            /*   dynamically allocated copy of the string */
  struct strnode * next; /*   next node in the list or NULL if none    */
};

struct strnode * str_list;    /* unordered list of strings */

// Major bugs in the following function are that code needs to be added
// to dynamically allocate both the node and an array for the copy of the
// string, then the entire string needs to be copied, not just the pointer.

/* return a new node pointing to a copy of string s and with next==NULL */
struct strnode * new_node(char * s) {
  struct strnode * node;
  node = (struct strnode *)malloc(sizeof(struct strnode));
  node->str = (char *)malloc((strlen(s)+1)*sizeof(char));
  node->str = s;
  strcpy(node->str, s);
  node->next = NULL;
  return node;
}

/* add a new node containing a copy of s to the front of the list */
void insert(char * s) {
  struct strnode * p = new_node(s);  // missing * in declaration of p
  p->next = str_list;     // need to reverse the order of these 2 statements
  str_list = p;
}
```