Name:_____

CSE 303, Winter 2007, Final Examination
15 March 2007

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, closed-note, except for two 8.5x11in pieces of paper (both sides).
- **Please stop promptly at 10:20.**
- You can rip apart the pages, but please write your name on each page.
- There are **100 points** total, distributed **unevenly** among 8 questions (many of which have multiple parts).
- When writing code, style matters, but don't worry about indentation.

Question	Max	Grade
1	15	
2	15	
3	10	
4	15	
5	10	
6	20	
7	5	
8	10	
Total	100	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit.**
- The questions are not necessarily in order of difficulty. **Skip around.**
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**15** points) Consider the following C program:

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

#define MAX_SIZE 1024
#define MAX_BUF  256

typedef struct item {
  char product[MAX_BUF];
  int price;
} Item;


void add_element(Item* inventory[], int* total, Item* item) {

  if ( (*total) < MAX_SIZE) {
    inventory[*total] = item;
    (*total)++;
  }

}

int main() {

  Item *inventory[MAX_SIZE];
  int total = 0;

  Item *p = (Item*)malloc(sizeof(Item));
  if ( !p ) {
    printf("Out of memory\n");
    return 1;
  }
  strncpy(p->product,"Product",MAX_BUF);
  p->product[MAX_BUF-1]='\0';
  p->price = 30;

  add_element(inventory,&total,p);
  free(p);

  // ... more code ...

  return 0;
}
```

(a) (**5** points) There is a bug in the above program related to memory management. Explain what the problem is.

(b) (**5** points) Fix the above program by editing **only** the function `main`. Modify directly the source code above.

(c) (**5** points) Fix the above program by modifying function `add_element`. Feel free to either re-write this function below or modify the source code above.

Solution:

(a) The function `add_element` copies the value of the pointer to `Item` that it is given as argument directly into the `inventory` array. After `add_element` returns both `p` and `inventory[total-1]` point to the same memory location. When `main` frees this chunk of memory (`free(p)`), this creates a dangling pointer in the `inventory` array.

(b) To fix the program, we must remove the call to `free` in `main`.

(c)
```
void add_element(Item* inventory[], int* total, Item* item) {

  if ( (*total) < MAX_SIZE) {

    Item* new_item = (Item*)malloc(sizeof(Item));
    if ( ! new_item ) {
      printf("Out of memory\n");
      return;
    }
    strncpy(new_item->product,item->product,MAX_BUF);
    new_item->product[MAX_BUF-1]='\0';
    new_item->price = item->price;

    inventory[*total] = new_item;
    (*total)++;
  }

}
```

2. (**15** points) Consider a software system composed of the following set of files:

```
% ------------------------------
% Content of A.h
% ------------------------------
#ifndef A_H
#define A_H

typedef struct s_a {
  int a;
} A;

void printA(A element);

#endif
```

```
% ------------------------------
% Content of A.c
% ------------------------------
#include <stdio.h>
#include "A.h"

void printA(A element) {
  printf("Element A {%d}\n",element.a);
}
```

```
% ------------------------------
% Content of B.h
% ------------------------------
#ifndef B_H
#define B_H

typedef struct s_b {
  int b;
} B;

void printB(B element);

#endif
```

```
% ------------------------------
% Content of B.c
% ------------------------------
#include <stdio.h>
#include "B.h"

void printB(B element) {
  printf("Element B {%d}\n",element.b);
}
```

```
% ------------------------------
% Content of C.h
% ------------------------------
#ifndef C_H
#define C_H

#include "A.h"
#include "B.h"

typedef struct s_c {
  A element_a;
  B element_b;
} C;

void printC(C element);

#endif
```

```
% ------------------------------
% Content of C.c
% ------------------------------
#include <stdio.h>
#include "C.h"

void printC(C element) {
  printf("Element C {\n");
  printA(element.element_a);
  printB(element.element_b);
  printf("}\n");
}
```

```
% ------------------------------
% Content of main.c
% ------------------------------
#include "A.h"
#include "B.h"
#include "C.h"

int main() {

  A item1;
  item1.a = 3;

  B item2;
  item2.b = 4;

  C item3;
  item3.element_a = item1;
  item3.element_b = item2;

  printC(item3);

  return 0;

}
```

(a) (**10** points) The Makefile below builds the software system shown above, but it contains several errors. Fix this Makefile by correcting the rules that have errors and by adding any possibly missing rules.

```
CC = gcc
CFLAGS = -Wall -g
LDFLAGS =

PROGRAMS = main

all: $(PROGRAMS)

A.o: A.c A.h
  $(CC) $(CFLAGS) -c $<

main.o: main.c
  $(CC) $(CFLAGS) -c $<

B.c: B.h
  $(CC) $(CFLAGS) -c $<

main: main.o A.o B.o C.o
  $(CC) $(LDFLAGS) -o $@ $^

clean:
  rm -f *.o $(PROGRAMS)
```

Reminder:

- `$@` designates the current target
- `$^` designates all prerequisites
- `$<` designates the left-most prerequisite

If you write any additional rules, you do not need to use the above symbols.

(b) (**5** points) What would happen if we removed **all** the following preprocessor directives **at the same time** from the header files and typed `make`?

From `A.h` remove	From `B.h` remove	From `C.h` remove

`#ifndef A_H`
`#define A_H`
`#endif`
	`#ifndef B_H`
	`#define B_H`
	`#endif`
		`#ifndef C_H`
		`#define C_H`
		`#endif`

Solution:

(a) Note that the order of the rules does not matter, except for `all`, which must appear as the first rule.

```
CC = gcc
CFLAGS = -Wall -g
LDFLAGS =

PROGRAMS = main

all: $(PROGRAMS)

A.o: A.c A.h
        $(CC) $(CFLAGS) -c $<

main.o: main.c A.h B.h C.h
        $(CC) $(CFLAGS) -c $<

B.o: B.c B.h
        $(CC) $(CFLAGS) -c $<

main: main.o A.o B.o C.o
        $(CC) $(LDFLAGS) -o $@ $^

C.o: C.c C.h B.h A.h
        $(CC) $(CFLAGS) -c $<

clean:
        rm -f *.o $(PROGRAMS)
```

(b) `main.c` includes `A.h`, `B.h`, and `C.h`. But `C.h` also includes `A.h` and `B.h`. Without the preprocessor directives, the content of `A.h` and `B.h` would get added twice into the pre-processed `main.c` file. This would cause a compiler error since the structures have been defined twice.

3. (**10** points) For each statement below about CVS, indicate if it is true or false.

   (a) (**2** points) A team of software developers is using CVS to manage their source code. When a team member modifies a file and commits his or her changes, these changes propagate automatically to all other team members, without them performing any operations.

   true false

   (b) (**2** points) When using a version control system such as CVS, it is possible to recover old versions of previously modified and committed files.

   true false

   (c) (**2** points) Alice and Bob edit the same source file at the same time. Alice commits her changes first. When Bob updates his local copy before committing his own changes, CVS will try to merge the changes made by Bob and Alice.

   true false

   (d) (**2** points) Chuck creates a new directory called `mydirectory` and adds it to CVS with the command: `cvs add mydirectory`. From that point on, all the files that Chuck creates in `mydirectory` will automatically be added to the CVS repository.

   true false

   (e) (**2** points) Chuck creates a new directory called `mydirectory`, **adds some files to that directory** and adds the directory to CVS with the command: `cvs add mydirectory`. All files in `mydirectory` are automatically added to CVS as well.

   true false

Solution:

(a) false

(b) true

(c) true

(d) false

(e) false

4. (**15** points) Your friend Donna accidentally erased the specification for one of her functions. That function, called `validate` checks if the content of an array of integers is valid or not.

   (a) (**5** points) Given the current implementation of function validate below, help Donna recover her specification for the function:

```
/**
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 *
 */
bool validate(int *array, int size) {

  assert(array);

  if ( size == 0 ) {
    return true;
  }

  int i;
  for ( i = 0; i < size; i++) {
    if ( (array[i] % 2) || (array[i] < 2) || (array[i] > 20)) {
      return false;
    }
  }

  return true;
}
```

(b) (**5** points) List the *smallest possible* number of *white-box* test cases for function `validate` that still *achieve full statement coverage*. For each test case, only show the arguments that you are passing to the function as shown in the example below:

```
Example test case: int array[] = { 1, 2, 3, 4, 5 }; int size = 5;
(this test case does not count)
```

(c) (**5** points) List what could constitute a good set of 10 *black-box* test cases for function `validate`.

Solution:

(a)
```
/**
 * Validates the content of an array
 *
 * Precondition
 * Argument array must point to an array of integers
 * (it cannot be null nor dangling)
 * The size of the array must be equal to the value
 * of argument size.
 *
 * Postcondition
 * The array given as argument is declared to be
 * valid if it contains only even integers
 * between 2 and 20.
 * An empty array is considered valid.
 *
 * @param array the array to validate
 * @param size the size of the array
 * @return true if the array is valid and false otherwise
 */
```

(b) Three test cases are necessary to achieve full statement coverage: the case where the array is empty, the case where the array contains only valid numbers, and the case where the array contains one or more invalid numbers. Below are three possible test cases:

```
Test case 1: int array[] = { }; int size = 0;
Test case 2: int array[] = { 2, 4, 6, 8 }; int size = 4;
Test case 3: int array[] = { 1, 3, 5 }; int size = 3;
```

(c) Many test-cases are possible. We accepted any set of black-box test cases where each test contained inputs from a different equivalence class. Below are 10 examples of valid test cases:

```
Test case 1: int array[] = { }; int size = 0;
Test case 2: int array[] = { 2 }; int size = 1;
Test case 3: int array[] = { 1 }; int size = 1;
Test case 4: int array[] = { 20 }; int size = 1;
Test case 5: int array[] = { 21 }; int size = 1;
Test case 6: int array[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 }; int size = 10;
Test case 7: int array[] = { 1, 3, 5 }; int size = 3;
Test case 8: int array[] = { 2, 4, 6, 8, 9, 10, 12, 14, 16, 18, 20 }; int size = 11;
Test case 9: int array[] = { 2, 4, 6, 8, 21, 10, 12, 14, 16, 18, 20 }; int size = 11;
Test case 10: int array[] = { 2, 4, 6, 8, 1, 10, 12, 14, 16, 18, 20 }; int size = 11;
```

5. (**10** points) Consider the following C++ program.

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

class Vehicle {

public:
  Vehicle(string owner) : _owner(owner) {}
  virtual ~Vehicle() {}
  virtual void print() = 0;

protected:
  string _owner;

};


class Car : public Vehicle {

public:

  Car(string owner, string manufacturer, string model)
  : Vehicle(owner),
    _manufacturer(manufacturer),
    _model(model) {
  }

  virtual ~Car() {}

  virtual void print() {
    cout << "owner = " << _owner << ", "
         << "manufacturer = " << _manufacturer << ", "
         << "model = " << _model << endl;
  }

private:
  string _manufacturer;
  string _model;

};


void function1() {

  Car car1("Alice","Honda","Accord");                        // (a) Legal    (b) Illegal

  Vehicle *vehicle1 = new Vehicle("Bob");                    // (a) Legal    (b) Illegal

  Car *car2 = new Car("Donna","BMW","335");                  // (a) Legal    (b) Illegal

  Vehicle *vehicle2 = new Car("Chuck","Toyota","Rav4");      // (a) Legal    (b) Illegal

  Car *car3 = new Vehicle("Erik","Mitsubishi","Lancer");     // (a) Legal    (b) Illegal
}
```

```
void function2() {

  Car *car = new Car("Eugene","Dodge","RAM");

  Vehicle *vehicle = dynamic_cast<Vehicle*>(car);
  if ( vehicle ) {
    cout << "(1) OK" << endl;
    vehicle->print();
  } else {
    cout << "(1) ERROR" << endl;
  }

  Car *car2 = dynamic_cast<Car*>(vehicle);
  if ( car2 ) {
    cout << "(2) OK" << endl;
    car2->print();
  } else {
    cout << "(2) ERROR 3 " << endl;
  }

}

int main() {
  function1();
  function2();
  return 0;
}
```

(a) (**5** points) Examine the statements in `function1` above. For each statement, indicate if it is legal or illegal by circling the corresponding comment. A statement is illegal if it results in a compile-time error or a runtime error.

(b) (**5** points) Examine `function2`. What will this function print if we execute the program? (Warning: notice the calls to member function `print`).

Solution:

(a)
```
(a) Legal
(b) Illegal
(a) Legal
(a) Legal
(b) Illegal
```

(b)
```
(1) OK
owner = Eugene, manufacturer = Dodge, model = RAM
(2) OK
owner = Eugene, manufacturer = Dodge, model = RAM
```

6. (**20** points) Consider the following C++ program:

```
// ----------------------------
// Content of StringList.h
// ----------------------------
#ifndef STRINGLIST_H
#define STRINGLIST_H
#include <iostream>
#include <cassert>

#define BUF_SIZE 1024
using namespace std;

typedef struct node {
  char original[BUF_SIZE];
  struct node *next;
} Node;

class StringList {

public:
  StringList();
  StringList(const StringList& old);
  ~StringList();
  void insert (const char *original);
  void print() const;
private:
  Node* _head;

};
#endif

// ----------------------------
// Content of main.cc
// ----------------------------
#include "StringList.h"

using namespace std;

int main() {

  StringList *list1 = new StringList();
  list1->insert("xxxx");
  list1->insert("aaaa");

  StringList *list2 = new StringList(*list1);
  list2->insert("cccc");

  list1->print();     // Line 14
  list2->print();     // Line 15

  delete list1;
  delete list2;

  return 0;

}
```

```
// ----------------------------
// Content of StringList.cc
// ----------------------------
#include "StringList.h"

StringList::StringList() : _head(NULL) {}

StringList::StringList(const StringList& old) {
  _head = old._head;
}

StringList::~StringList() {
  while (_head) {
    Node *next = _head->next;
    delete _head;
    _head = next;
  }
}

void
StringList::insert (const char *original) {

  assert( original );
  assert( strlen(original) < BUF_SIZE );

  Node *node = new Node();
  if ( !node ) {
    cerr << "Out of memmory\n";
    return;
  }
  strncpy(node->original,original,BUF_SIZE);
  node->original[BUF_SIZE-1] = '\0';
  node->next = NULL;

  node->next = _head;
  _head = node;

}

void
StringList::print () const {
  Node *current = _head;
  cout << "\nList content is:\n";
  while (current) {
    cout << current->original << endl;
    current = current->next;
  }
}
```

(a) (**5** points) What is the output of the above program at line 14? **Be very careful! This question is more tricky than it seems at first glance.**

(b) (**5** points) What is the output of the above program at line 15? **Be very careful! This question is also tricky.**

(c) (**5** points) What happens after line 15?

(d) (**5** points) Without writing any code, explain in plain English how you could fix this problem?

Solution:

(a)
```
List content is:
aaaa
xxxx
```

(b)
```
List content is:
cccc
aaaa
xxxx
```

(c) Because the copy constructor of class `StringList` makes only a shallow copy of the list, both `list1` and `list2` share a subset of elements in their lists. The line: `delete list1` causes the invocation of the destructor of `list1` which deletes all the elements in `list1`. This causes `list2` to have a dangling pointer in the middle of its list. `delete list2` causes the invocation of the destructor of `list2`, which tries to delete the previously shared `Node` instances for a second time. This causes the program to crash (segmentation fault).

(d) The copy constructor should make a deep copy of the list given as argument. It should copy all the elements in the list one by one. Note, that we were not looking just to avoid the crash but a proper bug fix that would also be robust against any other code we could add later on.

7. (**5** points) While testing a C program, called `main`, you discovered that for some inputs, the program enters an infinite loop somewhere inside function `X`. Function `X` contains 3 for loops and 2 while loops. Explain how you would use a debugger, such as gdb, to figure out where in function `X` the problem lies.

Solution:
We need to compile the program with the debugging information (option `-g`), and execute the program with the debugger (`gdb main`). Before starting the program, we can place a breakpoint at the beginning of each loop inside function `X`. We can do this by specifying the appropriate line numbers. We can then run the program on one of the inputs that causes the infinite loop. The execution will stop at the first breakpoint that we set. We can then continue execution until the next breakpoint. If the execution reaches and stops at the next breakpoint, we know that the previous loop is fine. If the execution never makes it to the next breakpoint (or the end of the program for the last breakpoint), then we know that the previous loop is faulty. We can then clear all breakpoints, leaving only the one at the faulty loop. By restarting the program, the execution will now stop at the beginning of the faulty loop. From that point on, we can step through the loop one line at the time. At each line, we can print and examine the current value of all local and global variables that are within scope. We can continue to do this until we find the problem.

8. (**10** points)

   (a) (**4** points) Why would anyone want to write a multi-threaded program?

   (b) (**4** points) What are some drawbacks of having multiple threads in a program?

   (c) (**2** points) Name (without explaining) two types of problems that can occur in a multi-threaded program that cannot occur in a program with a single thread.

Solution:

9. A multi-threaded program can potentially achieve a higher performance than a single-threaded one, because executing multiple threads in parallel can improve the utilization of system resources. For example, if one thread blocks while reading data off the disk, another thread can continue its execution and keep the CPU busy. Similarly, a multi-threaded program can benefit from multiple processors as different threads can run on different processors at the same time.

10. A multi-threaded program is more complex. Properly synchronizing threads by locking shared variables is difficult. Multi-threaded programs are also more difficult to debug. Finally, there is an overhead in running multiple threads.

11. Race conditions and deadlocks.