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.

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 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 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 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`

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

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`.

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`).

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?

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.

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.