CMPUT 201 - Practical Programming Methodology
Final Exam - A2 - Fall 2000 - Friday December 15, 2000
University of Alberta, Department of Computing Science

Name:
Id:
Section: A2 (Jim Hoover)

Instructions

- Write your name and student id number in the box above.
- This is an **OPEN BOOK** exam. Time allowed: 120 minutes.
- Place all answers in the spaces provided on the question pages. **JUSTIFY** each answer appropriately.
- This exam counts 35% toward your final grade in this course. This exam is worth 50 points. The weight of each question is indicated in square brackets by the question number.
- This exam is not impossible, but the questions do not necessarily have obvious answers. Think about each question for couple of minutes before answering it.
- There should be 4 questions and 9 pages in this exam booklet. You are responsible for checking that your exam booklet is complete.

Question	Mark	Out Of
1		10
2		10
3		10
4		20
TOTAL		50

Question 1 [10 marks]: Circle T or F for each of the statements:

Part 1.1 [1 mark]: T F

All memory used in a program comes from either the heap or the runtime stack.

Part 1.2 [1 mark]: T F

In C, every array of characters with a NUL at the end is a string.

Part 1.3 [1 mark]: T F

In C, pointers to functions hold the address of the `frame pointer` on the execution stack.

Part 1.4 [1 mark]: T F

In C++ the value of a pointer to an object is the address of that object.

Part 1.5 [1 mark]: T F

In C++ a copy constructor is always called when passing an object to a function.

Part 1.6 [1 mark]: T F

When designing software, you should always decide as early as possible how your data structures and objects should be implemented.

Part 1.7 [1 mark]: T F

In C, it is safe to call `free` on a NULL pointer.

Part 1.8 [1 mark]: T F

In C, it is safe to call `free` on any pointer.

Part 1.9 [1 mark]: T F

In C++, if a function wants to change the value of an object `x` which belongs to the caller of the function, then `x` must be passed by reference.

Part 1.10 [1 mark]: T F

In C++, if a constant pointer is passed to a function, then the object pointed to by the pointer cannot be changed.

Question 2 [10 marks total]: Circle the **one** best answer from the available choices:

Part 2.1 [2 marks]: The implementation of an `operator=` for a class `X` is influenced by

A) whether `X` has value semantics
B) the possibility of self-assignment
C) assignment from objects not of class `X`
D) A, B, and C

Part 2.2 [2 marks]: The copy constructor of a class

A) always implements value semantics
B) always does a deep copy
C) always exists
D) can be virtual

Part 2.3 [2 marks]: Templates are used to

A) reuse code
B) implement inheritance
C) implement polymorphism
D) parameterize classes and code by class type

Part 2.4 [2 marks]: If class C is to be used as a base class (i.e. other classes will be derived from C), then

A) all methods of C are virtual
B) the destructor of C should be virtual
C) some virtual methods of C may be unimplemented
D) B and C

Part 2.5 [2 marks]: Consider the following `C` code fragment then select the correct statement.

```
char Y[10];
char *X=&Y[0];
```

A) the address of `X[0]` is the same as the address of `Y[0]`
B) the address of `Y[0]` is stored in `X`
C) `Y == X` is true
D) all of the above

Question 3 [10 marks total]:

Part 3.1 [5 marks]: Consider the following C program

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    char s[] = "outputs";

    for ( i=0; i <= strlen(s); i++ ) {
        fprintf(stderr, "Iteration %d, length is %d\n", i, strlen(s));
        s[i] = '#';
    }
}
```

When we ran it, we expected the last line of output to be

```
Iteration 7, length is 7
```

But instead we get the following output:

```
Iteration 0, length is 7
Iteration 1, length is 7
Iteration 2, length is 7
Iteration 3, length is 7
Iteration 4, length is 7
Iteration 5, length is 7
Iteration 6, length is 7
Iteration 7, length is 7
Iteration 8, length is 9
```

Explain why.

Part 3.2 [5 marks]: Consider the following C program

```
#include <stdio.h>
void set_to_hash(char *s) {
    int i;
    int len = strlen(s);

    for ( i=0; i <= len; i++ ) {
        s[i] = '#';
    }
}

void do_output(void) {
    char string1[] = "this is strange";
    char string2[] = "outputs";

    set_to_hash(string2);

    printf("string1 is '%s'\n", string1);
    printf("string2 is '%s'\n", string2);
}

int main(int argc, char *argv[]) {
    do_output();
    return 0;
}
```

When we run it, we get the following output:

```
string1 is 'this is strange'
string2 is '########this is strange'
```

Explain why.

Question 4 [20 marks total]: Recall from lectures, the Memory class we defined for representing memory.

```
#ifndef MEMORY_H
#define MEMORY_H
#include "util.h"

class Memory {
    int low_addr;
    int high_addr;              // invariant:  low_addr <= addr < high_addr
    unsigned char* data;        // invariant:  data[0] contains cell at low_addr
public:
    Memory() { low_addr = 0; high_addr = 32678; initialize(); };

    Memory(int v_low_addr, int v_high_addr) {
        low_addr = v_low_addr; high_addr = v_high_addr; initialize(); }

    virtual ~Memory() { clear(); }

    virtual int get_low_addr() const { return low_addr; }

    virtual int get_high_addr() const { return high_addr; }

// disable value semantics
private:
    // no copy constructor
    Memory(const Memory& m) { check_assert(0);  }

    // no assignment operator
    void operator=(const Memory& m) { check_assert(0);  }
public:

// public interface
    // accessors to individual memory cells
    virtual unsigned int get_cell(int addr) {
        check_assert( low_addr <= addr && addr < high_addr );
        return data[addr];
    }

    virtual void set_cell(int addr, unsigned int value) {
        check_assert( low_addr <= addr &&  addr < high_addr );
        data[addr] = value & 0xff;
    }

// internal utility functions
private:
    // initialize a newly created object to a consistent state
    // call only once on an object
    void initialize() {
        check_assert(0 <= low_addr && low_addr < high_addr);
        data = new unsigned char[high_addr-low_addr];
    }

    // reset a consistent object to well-defined initial
    // state with no allocated data members.
    void clear() { delete[] data; data = 0;  }
};
#endif
```

Part 4.1 [2 marks]: Why do constructors and destructors for the `Memory` class have to be very careful about when they call the method `Memory::clear`?

Part 4.2 [2 marks]: Why is it not a good idea to declare an array like: `Memory bunchOfMemories[100000];`

Part 4.3 [2 marks]: There is an error in the implementation of `get_cell` and `set_cell` that manifests itself when `low_addr` is not 0. What is the error and what is the fix?

Part 4.4 [14 marks]: Using `Memory` as a base class, define a new class `TrackedMemory` that keeps track of the address range of memory cells actually accessed. `TrackedMemory` will have two new interface methods `get_lowest_used` and `get_highest_used` that when called give the lowest and highest address that were actually ever used in a `get_cell` or `set_cell` call. An example program follows.

When you do the derivation, keep in mind the following issues:

1. This code for `Memory` is different than the one we used in lectures. In particular all of the public interface methods are virtual as recommended. You must not alter the code for `Memory`, unless absolutely necessary. In that case, make your changes directly on the code provided above and explain why they were necessary.

2. Consider what methods in `Memory` need to be extended in `TrackedMemory`. When you extend them, you should try to use as much of the original capability in `Memory` as possible.

3. You should consider boundary cases such as what happens if you never actually accessed memory? Do we want to disable value semantics? Thus there may be additional methods you want to provide in addition to the requested `get_lowest_used` and `get_highest_used`.

Example:

```
#include <stdio.h>
#include "memory.h"
#include "trackedmem.h"
int main(int argc, char *argv[]) {
    TrackedMemory m(0, 20);

    m.set_cell(2, 22);
    printf("[%d,%d]\n", m.get_lowest_used(), m.get_highest_used() );
    m.get_cell(16);
    printf("[%d,%d]\n", m.get_lowest_used(), m.get_highest_used() );
    m.get_cell(1);
    printf("[%d,%d]\n", m.get_lowest_used(), m.get_highest_used() );
}
```

will output

```
[2,2]
[2,16]
[1,16]
```