CSE 333 Final Exam June 8, 2016

Name _____ ID # _____

There are 8 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed electronics, closed telepathy, open mind.

If you don't remember the exact syntax for something, make the best attempt you can. We will make allowances when grading.

Don't be alarmed if there seems to be more space than is needed for your answers – we tried to include more than enough blank space.

Relax, you are here to learn.

Please wait to turn the page until everyone is told to begin

Score _____ / 100

1. _____ / 18
2. _____ / 18
3. _____ / 16
4. _____ / 16
5. _____ / 6
6. _____ / 14
7. _____ / 11
8. _____ / 1

Reference information. Here is a collection of information that might, or might not, be useful while taking the test. You can remove this page from the exam if you wish.

C++ strings: If `s` is a string, `s.length()` and `s.size()` return the number of characters in it. Subscripts (`s[i]`) can be used to access individual characters.

C++ STL:

- If `lst` is a STL `vector`, then `lst.begin()` and `lst.end()` return iterator values of type `vector<...>::iterator`. STL `list`s and `set`s are similar.
- A STL `map` is a collection of `Pair` objects. If p is a `Pair`, then `p.first` and `p.second` denote its two components. If the `Pair` is stored in a `map`, then `p.first` is the key and `p.second` is the associated value.
- If `m` is a `map`, `m.begin()` and `m.end()` return iterator values. For a `map`, these iterators refer to the `Pair` objects in the `map`.
- If `it` is an iterator, then `*it` can be used to reference the item it currently points to, and `++it` will advance `it` to the next item, if any.
- Some useful operations on STL containers (lists, maps, sets, etc.):
  - `c.clear()` – remove all elements from c
  - `c.size()` – return number of elements in c
  - `c.empty()` – true if number of elements in c is 0, otherwise false
- Additional operations on `vector`:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `map`s:
  - `m.insert(x)` – add copy of `x` to `m` (a key-value pair for a `map`)
  - `m[k]` can be used to access the value associated with key `k`. If `m[k]` is read and has never been accessed before, then a <key,value> `Pair` is added to the map with `k` as the key and with a value created by the default constructor for the value type (0 or `nullptr` for primitive types).
- Some additional operations on `set`s
  - `s.insert(x)` – add `x` to `s` if not already present
  - `s.count(x)` – number of copies of `x` in `s` (0 or 1)
- You may use the C++11 `auto` keyword, C++11-style `for`-loops for iterating through containers, and any other features of standard C++11, but you are not required to do so.

Question 1. (18 points) A bit of C++ coding. In this problem we would like to implement parts of an inverted document index related to, but (much) simpler than, the project code we built this quarter. We also want to take advantage of the standard C++ libraries. After describing the data structure, this problem will ask you to implement two functions.

We want a data structure that stores all of the words found in a collection of documents and, for each word, the name(s) of the documents where that word appears one or more times. So, for instance, if we had the following two documents and the words contained in them:

"Hamlet" "to be or not to be"
"Strangers" "do be do be do"

we would like to create a table that looks like this in part:

"to"	"Hamlet"
"be"	"Hamlet" "Strangers"
"or"	"Hamlet"
…	

In other words, what we want is a map of <key, value> pairs, where each key is a word (a string) contained in one or more documents, and each value is a set of document names (a set of strings).

This data structure can be defined directly in C++ using the following definition to create a type `WordTable` with the above structure:

```
typedef map<string, set<string>> WordTable;
```

On the next pages, implement two functions: one to add information to a `WordTable` and the other to print a formatted listing of a `WordTable`.

Don't be alarmed if your answers are quite short. There are solutions to both problems that require only a few lines of code. We've put each part of the problem on a separate page to leave *way* more than enough room for your answers.

(You may remove this page from the exam if you wish.)

Question 1. (cont.) Implement the following functions.

(a) (8 points) Given a document name `doc` (e.g., "Hamlet"), and a list of words `words` (e.g., { "to", "be", "or", "not", "to", "be"}), add entries to `table` as needed to record that those words appear in that document. Do not change any of the strings or attempt to discover if the same word is stored in the table more than once with different capitalization. (i.e., use the strings exactly as provided and don't attempt to convert them to lower-case or something.)

```
void AddDocWordList(const string doc,
                    const vector<string> &words,
                    WordTable &table) {









}
```

(You almost certainly won't need all of this space. ☺)

Question 1. (cont.) (b) (10 points) Given a `WordTable table`, write the contents of the table to `cout` with the following format: there should be one line for each word that appears as a key in the table. Each output line should consist of that word (key) followed by a colon, followed by a list of the documents names that contain that word (i.e., the contents of the associated set of strings), separated by spaces. The output for our example would include the following two lines, among others:

```
be: Hamlet Stranger
not: Hamlet
```

The words and the lists of documents containing each word may be printed in any order. You should print the strings as-is and not attempt to capitalize or modify them in any way.

```
void PrintWordTable(WordTable &table) {




}
```

Question 2. (18 points) Constructor madness. Consider the following program, which as is traditional, compiles and executes without any errors or memory problems.

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

class Thing {
public:
  // constructors
  Thing(): x_(17)     { cout << "Thing::Thing()" << endl; }
  Thing(int n): x_(n) { cout << "Thing::Thing(n)" << endl; }
  Thing(const Thing &t): x_(t.x_) {cout<<"Thing copy ctr"<<endl;}
  // destructor
  virtual ~Thing()    { cout << "Thing::~Thing()" << endl; }
  // other member functions
  virtual int get()   { cout << "Thing::get" << endl;
                        return grab(); }
          int grab()  { cout << "Thing::grab" << endl;
                        return x_; }
private:
  int x_;
};

class Blob: public Thing {
public:
  Blob(): Thing(42), n_(2) { cout << "Blob::Blob()" << endl; }
  Blob(const Blob &b): Thing(), n_(b.n_)
                           { cout << "Blob copy ctr" << endl; }
  virtual ~Blob()          { cout << "Blob::~Blob" << endl; }
  int get()  { cout << "Blob::get" << endl; return grab(); }
  int grab() { cout << "Blob::grab" << endl; return n_; }
private:
  int n_;
};

int main() {
  Thing t;
  cout << t.get() << endl;
  Blob b;
  Thing* ptr = &b;
  cout << ptr->get()  << endl;
  cout << ptr->grab() << endl;
  Blob *q = new Blob(b);
  cout << q->get()  << endl;
  cout << q->grab() << endl;
  delete q;
  return 0;
}
```

(Question continued on next page. You may remove this page for convenience.)

Question 2. (cont). Exactly what does this program print when it is executed? You should work carefully as you trace the code (of course). A few hints:

- Objects are constructed in the order of declaration
- Base class constructors run before derived class constructors
- Destructors run in the reverse order of constructors
- Of course, constructors and destructors are executed as needed when heap objects are explicitly allocated or deleted.

Write the output produced by the program below.

Question 3. (16 points) Templates & things. This C++ code defines a class that implements a linked list of integers and a small main program that uses it. All of the code is included in the class definition to keep the question somewhat shorter – there is no separate implementation file. This code does compile and execute successfully.

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

class List {
public:
  // construct empty list
  List() : head_(nullptr) { }

  // add new node with value n to the front of the list
  virtual void add(int n) {
    Link *p = new Link(n, head_);
    head_ = p;
  }

private:
  struct Link {  // nodes for the linked list
    int val;
    Link * next;
    Link(int n, Link* nxt): val(n), next(nxt) { }
  };
  // List instance variable
  Link * head_;      // head of list or nullptr if list is empty
};  // end of List class

int main() {
  List nums;
  nums.add(1);
  nums.add(2);
  return 0;
}
```

Answer questions about this code on the next page. Leave this page in the exam – you need to mark some changes in the above code.

Question 3. (cont.) (a) (8 points) The `List` class on the previous page would be more useful if it could be used to store lists of any values, not just `int`s. On the previous page show the changes that would be needed to make this class into a template where the type of the payload data in the linked list nodes is a type parameter `T` instead of `int`.

Be sure to mark any corresponding changes needed, if any, in the `main` function at the bottom of the code.

Reminder: you'll need to start by adding `template <class T>` or `template <typename T>` (which means the same thing) at the beginning of the class definition.

(b) (8 points) There is at least one unfortunate problem with the original code. When we run the code using valgrind we get an output report that concludes ominously with:

```
==3276== LEAK SUMMARY:
==3276==    definitely lost: 16 bytes in 1 blocks
==3276==    indirectly lost: 16 bytes in 1 blocks
==3276==      possibly lost: 0 bytes in 0 blocks
==3276==    still reachable: 72,704 bytes in 1 blocks
==3276==         suppressed: 0 bytes in 0 blocks
```

Give a brief description of the what's wrong and then describe how to fix it. If the solution requires additional code, write the new code below and indicate where it should be added in the original code or what code it should replace. If it matters, your changes should assume that the template changes from part (a) of the problem have already been done.

Question 4. (16 points) `virtual` reality. The following program compiles, runs, and produces output with no error messages or other problems. Notice that there are no `virtual` functions in the code. Answer questions about it at the bottom of the page.

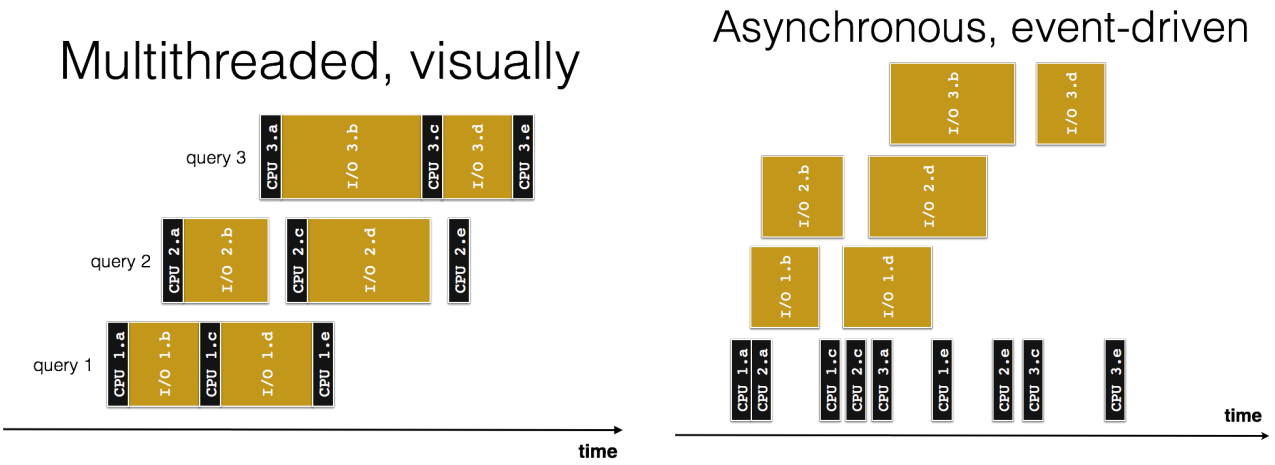
```
#include <iostream>
using namespace std;

class A {
public:
  void m1() { cout << "!"; }
  void m2() { cout << "a"; }
  void m3() { cout << "o"; }
  void m4() { cout << "s"; }
};

class B: public A {
public:
  void m1() { cout << "H"; }
  void m2() { cout << "3"; }
  void m3() { cout << "c"; }
  void m4() { cout << "k"; }
};

class C: public B {
public:
  void m1() { cout << "r"; }
  void m3() { cout << "l"; }
  void m4() { cout << " "; }
};
```

```
int main() {
  B b;
  C c;
  A *aPtr = &b;
  B *bPtr = &c;

  aPtr->m2();
  bPtr->m2();
  bPtr->m2();
  bPtr->m4();
  bPtr->m1();
  aPtr->m3();
  bPtr->m3();

  aPtr = &c;
  bPtr = &b;
  bPtr->m4();
  aPtr->m4();
  aPtr->m1();
  cout << endl;
  return 0;
}
```

(a) (8 points) What does this program print when it executes? Answer the question for the program exactly as written with no `virtual` functions.

(b) (8 points) Modify the program above by adding the `virtual` keyword in appropriate places so that the modified program will print `333 rocks!` (including the `!` and the space between `333` and `rocks`, but no spaces before or after). You may not make any other changes to the code other than showing where to add `virtual`. There may be more than one possible solution. Any correct solution is acceptable.

Question 5. (6 points) Concurrency strategies. In lecture we discussed several strategies for organizing a concurrent server. One was to use multiple threads to process transactions as in the left diagram below. The other was an asynchronous, event driven model, as in the right diagram. A third was to use multiple processes (fork) instead of threads, but we will not consider that strategy in this question.



(In case it's hard to read, the small black boxes in the diagrams represent CPU time slices used by individual transactions; the larger, lighter boxes represent I/O operations.)

If implemented properly both strategies provide good overlap of concurrent transactions and good utilization of CPU and I/O resources. But there are significant differences in the programming needed for the two strategies.

(a) (3 points) Give one major advantage of implementing a concurrent server using multiple threads (as in the left diagram) compared to an asynchronous event-driven server.

(b) (3 points) Give one major advantage of implementing a concurrent server using an asynchronous, event-driven architecture (as in the right diagram) compared to using multiple threads.

Question 6. (14 points) More concurrency, in C. This question is about locking and concurrency, but it needs a high-level overview of a new data structure before we get to the main question.

Suppose we had a Queue data structure that was very similar to the linked lists we implemented in HW1. Like the linked lists, our Queue type is a pointer to the actual data structure, which is some sort of linked list. Payload data stored in a Queue is represented by `void*` pointers. The basic operations are these:

```
// return a pointer to a newly allocated Queue data structure
Queue AllocateQueue();

// add payload to the end of the queue
void Enqueue(Queue q, QPayload_t payload);

// remove the front item from the queue and store it in result
void* Dequeue(Queue q, QPayload_t* result);

// return true if q is empty, otherwise return false
bool QueueEmpty(Queue q);
```

These operations are implemented in the obvious way – very much like the list structures from HW1

On the next pages there is a program that reads requests (data) from standard input (file descriptor 0) and writes computed results to standard output (file descriptor 1). However the program uses two threads – one reads requests as they arrive and places them in a queue. The other removes requests from the queue when it is ready to process the next request. If the queue is empty when the second thread is ready to do more work, it waits for more data to become available. The end of data, which signals both threads to shut down, is indicated by a NULL value placed in the queue.

Look at the code on the next pages and then answer the questions at the end. Hint: you won't need to read every line of code in detail at first, so just skim it to get the idea of what's going on, then look at the questions before studying the code in more detail.

The original author of the code had an idea that locks might be needed in the future, but didn't know how to use them. So the main program does create a lock and later destroy it right before exiting, but never does anything with it.

In case it's useful for reference, here are some pthread function prototypes.

- `pthread_create(thread, attr, start_routine, arg)`
- `pthread_exit(status)`
- `pthread_join(thread, value_ptr)`
- `pthread_cancel (thread)`
- `pthread_mutex_init(pthread_mutex_t * mutex, attr)`
  `// attr=NULL usually`
- `pthread_mutex_lock(pthread_mutex_t * mutex)`
- `pthread_mutex_unlock(pthread_mutex_t * mutex)`
- `pthread_mutex_destroy(pthread_mutex_t * mutex)`

Question 6. (cont.) Start of the code. Please do not remove this page from the exam.

```
typedef void *Request;
static Queue queue;  // global Queue pointer variable
                     // Implemented similarly to HW1 LinkedList
static pthread_mutex_t lock;

// Return the next request read from infd. Return NULL if no more
// requests are available.  Implemented in another file.
Request ReadNextRequest(int infd);
// Process given Request and write appropriate response to outfd.
// Implemented in another file.
void WriteResponse(int outfd, Request req);

// Read requests from the int file descriptor encoded in arg and
// store them in the global queue as they become available.
void *producer_fn(void *arg) {
  int infd = (int) arg;
  while (1) {
    Request req = ReadNextRequest(infd);
    Enqueue(queue, (QPayload_t) req);
    if (req == NULL)
      break; // No more requests available - quit after putting
             // NULL in the queue
  }
  return NULL;
}
// Remove requests from the global queue and write the results
// on the int file descriptor encoded in arg.
void *consumer_fn(void *arg) {
  int outfd = (int) arg;
  Request req;
  while (1) {
    if (!QueueEmpty(queue)) {
      Dequeue(queue, (QPayload_t *) &req);
      if (req == NULL)
        break;  // Signals the end of requests to handle
      WriteResponse(outfd, req);
    } else {
      // Nothing to consume, sleep for 1 ms then check again
      sleep(1000);
    }
  }
  return NULL;
}
```

(code continued on next page)

Question 6. (cont.) (remainder of the code) Please do not remove this page either.

```
int main(int argc, char** argv) {
  pthread_t conThd, prdThd;
  pthread_mutex_init(&lock, NULL);
  queue = AllocateQueue();

  // Create threads for producing and consuming
  if (pthread_create(&prdThd, NULL, &producer_fn, (void *)0)!=0||
      pthread_create(&conThd, NULL, &consumer_fn, (void *)1)!=0){
    fprintf(stderr, "pthread_create failed\n");
    return EXIT_FAILURE;
  }
  // Wait for both threads to terminate
  if (pthread_join(prdThd, NULL) != 0 ||
      pthread_join(conThd, NULL) != 0) {
    fprintf(stderr, "pthread_join failed\n");
    return EXIT_FAILURE;
  }

  pthread_mutex_destroy(&lock);
  return 0;
}
```

Now (finally!!) for the questions.

(a) Is locking needed in this code to ensure correct operation? Give a brief explanation of why or why not.

(b) If locking is needed in the code to ensure correct operation, show where to place appropriate `pthread_mutex_lock(&lock)` and `pthread_mutex_unlock(&lock)` statements in the above code. (You cannot modify the `Queue` implementation itself – assume we don't have access to that source code.) For full credit your added code should ensure correct synchronization but still allow for as much concurrency as is reasonably possible in the producing and consuming threads (i.e., the critical sections enforced by the use of locks should be no larger than needed).

Question 7. (11 points) There's no place like 127.0.0.1.

Here is an alphabetical list of some functions that are used in TCP socket programming:

- accept()
- bind()
- close()
- connect()
- getaddrinfo()
- listen()
- read()/write()
- socket()

(a) (5 points) List in the correct order the functions that are executed to perform the 5 steps required for a client program to create a socket, exchange data with a server, and close the connection:

1.

2.

3.

4.

5.

(b) (6 points) List in the correct order the functions that are executed to perform the seven steps needed for a server to create a socket, exchange data with a client machine, and close the connection:

1.

2.

3.

4.

5.

6.

7.

Question 8. (1 free point) Draw something interesting below.

All interesting drawings get the free point.

By definition, all drawings where at least some drawing occurred are interesting.

Have a great summer!!
The CSE 333 staff