CSE 333 Final Exam June 6, 2017

Name _____ ID # _____

There are 8 questions worth a total of 120 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 _____ / 120

1. _____ / 24
2. _____ / 16
3. _____ / 22
4. _____ / 14
5. _____ / 14
6. _____ / 12
7. _____ /16
8. _____ /2

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 `lists` and `sets` 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 `vectors`:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `maps`:
  - `m.insert(x)` – add copy of `x` to `m` (a key-value pair for a `map`)
  - `m.count(x)` – number of elements with key `x` in `m` (0 or 1)
  - `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 `sets`
  - `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.

More reference information. You can also remove this page if you wish.

Some POSIX I/O and TCP/IP functions:

- int **accept**(int sockfd, struct socckaddr *addr, socklen_t *addrlen);
- int **bind**(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
- int **close**(int fd)
- int **connect**(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
- int **freeaddrinfo**(struct addrinfo *res)
- int **getaddrinfo**(const char *hostname, const char *service,
  const struct addrinfo *hints, struct addrinfo **res)
  - Use NULL or listening port number for second argument
- int **listen**(int sockfd, int backlog)
  - Use SOMAXCONN for backlog
- off_t **lseek**(int fd, off_t offset, int whence)
  - whence is one of SEEK_SET, SEEK_CUR, SEEK_END
- ssize_t **read**(int fd, void *buf, size_t count)
  - if result is -1, errno could contain EINTR, EAGAIN, or other codes
- int **socket**(int domain, int type, int protocol)
  - Use SOCK_STREAM for type (TCP), 0 for protocol, get domain from address info struct (address info struct didn't fit on this page – we'll include it later if needed)
- ssize_t **write**(int fd, const void *buf, size_t count)

Some pthread functions:

- 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)

Basic C memory management functions:

- void * **malloc**(size_t size)
- void **free**(void *ptr)
- void * **calloc**(size_t number, size_t size)
- void * **realloc**(void *ptr, size_t size)q

Question 1. (24 points) Some C and POSIX I/O programming. Given an `int` file descriptor returned by `open()`, write a C function `ReadFile` that reads the entire file designated by that file descriptor and returns a pointer to a heap-allocated array of bytes containing the data from that file, and also returns the total number of bytes read. The function result should be 1 (true) if the buffer was allocated and the data was read successfully, and should be 0 (false) if there was an error. If the function detects an error and returns 0, it should free any already-allocated data before returning (i.e., no memory leaks).

The prototype for this function is,

```
int ReadFile(int fd, char **buffer, int *length)
```

where `fd` is the integer file descriptor of an open, ordinary file, `buffer` is the address of a `char*` pointer, and `length` is the address of an integer variable.

`ReadFile` must use the POSIX `read` function to get the file data, and should correctly handle the case where `read` returns -1 and `errno` is `EAGAIN` or `EINTR`, in which case `read` should continue until it is able to successfully read the entire file, or until some different, non-recoverable error is encountered.

Example: here is a small main program that opens a file, then uses `ReadFile` to obtain the file contents.

```
int main(int argc, char **arv) {
  char *buffer;   // address of buffer returned by ReadFile
  int length;     // length of file read by ReadFile

  int fd = open("test.txt", O_RDONLY);
  ...
  int status = ReadFile(fd, &buffer, &length);
  if (status == 1) {
    // read succeeded - process file contents in buffer
    ...
    free(buffer);
  } else {
    // ReadFile encountered an error
  }
}
```

Write your answer on the next page. You can remove this page from the exam if you wish.

Question 1. (cont.) Write your implementation of `ReadFile` below. The function heading is provided for you.

```
int ReadFile(int fd, char **buffer, int *length) {









}
```

Question 2. (16 points) And now a bit of C++ coding. When we were indexing documents in homework 2, one of the extra credit suggestions introduced the idea of "stop words" – words like "a", "the", "of", and others that are not particularly useful in search results because they are so common.

For this problem, use C++ STL containers to implement a simple stop word filter function. The input to the function is a pair of vectors: one containing the words from a document, and the other containing a list of stop words. The function should return a pointer to a new, heap-allocated `vector<string>` containing a copy of the original words in the original order, but with all stop words removed. For instance, suppose the original text and stop words are

text: { "in", "the", "beginning", "was", "the", "end", "of", "a", "short", "story" }
stop words: { "a", "the", "of", "an" }

The function should return a pointer to a new `vector<string>` containing

{ "in", "beginning", "was", "end", "short", "story" }

There is one catch: for full credit, your solution must run faster than $O(n^2)$ time by storing the stop words in a set or map or some other container that can be accessed quickly, so you don't have to repeatedly re-scan the entire list of stop words to check for a match against each word in the input text. Instead you should store the stop words in a data structure where it is possible to check whether a word is present in either constant or $\log(n)$ time. Also you should not copy the entire set of input words into the output and then delete them one at a time, since each delete operation in a vector is itself an $O(n)$ operation. Just copy the words that do not appear in the stop list.

Hints: `<vector>`, `<set>`, maybe `<map>`

(Write your answer on next page. You may remove this page from the exam if you wish.)

Question 2. (cont). Implement the function `remove_words` below. You should assume all necessary headers are already `#included`, and you can assume a "`using namespace std;`" directive is present to save some writing.

Hint: the sample solution is fairly short, with only about 8-10 lines of actual code plus a few more lines with curly braces.

```
// return a new heap-allocated vector<string> with a copy
// of the input vector text from which every word in
// stop_words has been removed.
vector<string> * remove_words(const vector<string> & text,
                              const vector<string> &stop_words) {




}
```

Question 3. (22 points) The usual, demented virtual function madness. Consider the following program, which does compile and execute with no errors. Feel free to remove this page from the exam while working the rest of the problem.

```
#include <iostream>
using namespace std;
class A {
public:
                A()  { cout << "A::A" << endl; }
  virtual void f1()  { cout << "A::f1" << endl; }
  virtual void f2()  { f3(); cout << "A::f2" << endl; }
          void f3()  { f1(); cout << "A::f3" << endl; }
       virtual ~A()  { cout << "A::~A" << endl; }
};
class B: public A {
public:
                B() { cout << "B::B" << endl; }
          void f1() { cout << "B::f1" << endl; }
          void f3() { f1(); cout << "B::f3" << endl; }
       virtual ~B() { cout << "B::~B" << endl; }
};
class C: public B {
public:
           C() { cout << "C::C" << endl; }
     void f2() { f3(); cout << "C::f2" << endl; }
  virtual ~C() { cout << "C::~C" << endl; }
};
int main() {
  A* a1 = new A();
  a1->f2();
  cout << "----" << endl;
  A* a2 = new B();
  a2->f2();
  cout << "----" << endl;
  B* b1 = new C();
  b1->f2();
  cout << "----" << endl;
  delete a1;
  cout << "----" << endl;
  delete a2;
  cout << "----" << endl;
  delete b1;
  return 0;
}
```

Question 3. (cont.) (a) (12 points) Write the output produced when this program is executed. If the output doesn't fit in one column in the space provided, write multiple vertical columns showing the output going from top to bottom, then successive columns to the right.

(b) (10 points) Now, suppose we edit the original program and remove the keyword `virtual` from every place it appears in the code, then compile and run the program again (it does work if this is done). What output does it produce after this change?

Question 4. (14 points) The following C++ program creates a small binary tree and prints it (the output is `17 5 42`, in case it matters). However, the program leaks memory since it contains no code to delete any of the allocated nodes.

Fix this program so it executes without any memory leaks by replacing ordinary pointers with smart pointers where needed. You may not make any other changes, in particular you may not add any `delete` statements. Write your changes directly in the code.

```
#include <iostream>
#include <memory>  // for solution

// a simple binary tree node
struct BNode {
  int val_;
  BNode *left_;
  BNode *right_;
  // convenience constructor
  BNode(int val, BNode *left, BNode *right) :
    val_(val), left_(left), right_(right) { }
};


// print tree with root r using inorder traversal
void inorder(BNode *r) {
  if (r == nullptr)
    return;
  inorder(r->left_);
  std::cout << " " << r->val_ << " ";
  inorder(r->right_);
}


int main() {
  BNode *a = new BNode(5, new BNode(17,nullptr,nullptr),nullptr);
  BNode *root = new BNode(42, a, nullptr);
  inorder(root); std::cout << std::endl;
  return 0;
}
```

Question 5. (14 points) Templates. Now take another look at the original code from the previous problem. Among other limitations, the values stored in the tree nodes are restricted to `int`s. Your job this time is to make changes to the code so the node value type is given by a template parameter. To get started, we've added the initial template declaration and changed the type of `val_` in the `BNode` declaration. You need to make all of the other changes needed in the rest of the code, including any changes in the rest of the `BNode` declaration, and any changes needed elsewhere so the program will use nodes with `int` as the actual type and work just as it did before. (Do *not* make any smart pointer changes here – just change the types and don't worry about memory leaks.)

```
#include <iostream>
#include <memory>  // for soution

// a simple binary tree node
template <class T>
struct BNode {
  T val_;
  BNode *left_;
  BNode *right_;
  // convenience constructor
  BNode(int val, BNode *left, BNode *right) :
    val_(val), left_(left), right_(right) { }
};
// print tree with root r using inorder traversal


void inorder(BNode *r) {
  if (r == nullptr)
    return;
  inorder(r->left_);
  std::cout << " " << r->val_ << " ";
  inorder(r->right_);
}
int main() {
  BNode *a = new BNode(5, new BNode(17,nullptr,nullptr),nullptr);
  BNode *root = new BNode(42, a, nullptr);
  inorder(root); std::cout << std::endl;
  return 0;
}
```

Question 6. (12 points) A bit of networking. When we were describing how a network server works, we listed 7 steps that need to be done to establish communication with a client, exchange data, and shut down. In the list below, fill in the name of the function that is used at each step (the reference information at the beginning of the exam may be useful for this), then give a 1-sentence description of the purpose of that step. Step 6 (read/write) is done for you as an example, and the function name for step 2 is also provided. You should fill in the rest of the table.

1. Function: Purpose:

2. Function: **socket** Purpose:

3. Function: Purpose:

4. Function: Purpose:

5. Function: Purpose:

6. Function: **read/write** Purpose: exchange data with the client using the socket

7. Function: Purpose:

Question 7. (16 points) A bit of concurrency. One of the new interns is experimenting with threads and came up with the following demo program which creates 5 threads, each of which increments a global variable 1000 times. This program does compile and execute with no compile- or runtime errors.

```
#include <pthread.h>
#include <stdio.h>
#include <iostream>
#include <time.h>

static int total = 0;

void *thread_start(void *arg) {
  for (int i = 0; i < 1000; i++) {
    total = total + 1;
  }
  std::cout << "The total is now " << total << std::endl;
  return NULL;
}
int main(int argc, char** argv) {
  int threads = 5;                  // make 5 threads
  pthread_t thr_array[threads];
  for (int i = 0; i < threads; i++) {
    if (pthread_create(&thr_array[i],NULL,&thread_start,NULL)
                                                          != 0) {
      std::cerr << "pthread_create() failed." << std::endl;
    }
  }
  for (int i = 0; i < threads; i++) {
    if (pthread_join(thr_array[i], NULL) != 0) {
      std::cerr << "pthread_join() failed." << std::endl;
    }
  }
  std::cout << "The sum is " << total << std::endl;
  return 0;
}
```

(continued on the next page, but leave this page in the exam. You will need to write some changes to the code above.)

Question 7. (cont) When we ran the program on the previous page several times, we got strange results. Here is the output from three different executions:

```
$ ./sum
The total is now 1000
The total is now 1001
The total is now 2228
The total is now 2001
The total is now 3001
The sum is 3001
$ ./sum
The total is now The total is now 1030
1000
The total is now 2000
The total is now 2926
The total is now 2551
The sum is 2926
$ ./sum
The total is now The total is now The total is now 400010002000

The total is now 5000
The total is now 3000

The sum is 5000
```

(a) (6 points) Something odd is going on here. Not only is the output scrambled, but the final sum isn't always 5000, which it should be. Give a brief explanation of what is causing this strange behavior.

(b) (10 points) On the previous page, make the minimal changes needed to the program to guarantee that the global variable `total` is updated the correct number of times (e.g., 5000 in this case) and also to guarantee that the output is printed with each message occupying a single line and without having different parts of the output messages from the different threads mixed together. Your solution should, however, still allow for as much concurrent execution of the 5 threads as possible. For example, it is not reasonable to rewrite the code so the 5 threads run sequentially, one after the other.

Question 8. (2 free points – all answers get the free points)

Draw a picture of something you plan to do this summer, hopefully something interesting, exciting, fun, relaxing, or some combination of those and other adjectives.

Congratulations on a great quarter!!
Have a great summer and best wishes for the future.
The CSE 333 staff