CSE 333 25sp Final Exam 6/11/25

Name _____ UW netid _____@uw.edu
Please print legibly

There are 8 questions worth a total of 110 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, no AI, open mind. However, you may have two 5x8 notecards or the equivalent with any hand-written notes you wish written on both sides.

There is a blank page at the end with extra space for your answers if you need more room. It is after all the questions but before the detachable sheet with reference information.

After the extra blank pages for answers, there are two pages of assorted reference information (much of which you probably won't need). You should remove this sheet of paper from the exam for convenience. These pages will not be scanned or graded.

Do not remove any pages from the middle of the exam.

If you do not 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 some 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 _____ / 110

1. _____ / 18
2. _____ / 18
3. _____ / 10
4. _____ / 12
5. _____ / 18
6. _____ / 16
7. _____ / 16
8. _____ / 2

Question 1. (18 points) The traditional C++/STL programming. We have accumulated a list of stores and products that they sell. We'd like to write a small C++ program that reads this information from the standard input file (`cin`) and then prints out a list of all the products sorted by product name and for each shows the stores that sell them.

Example: suppose we have the following input read from `cin`:

```
qfc banana
amazon book
amazon iphone
cvs toothpaste
qfc  toothpaste
apple iphone
apple computer
ubookstore book
amazon banana
```

After reading this input and reaching the end of file on `cin`, the program should produce the following output:

```
banana: qfc amazon
book: amazon ubookstore
computer: apple
iphone: apple amazon
toothpaste: cvs qfc
```

In the output, the products must be listed (sorted) in order, but the stores that sell each product can be listed in any order.

Since this is an exam question, you may assume that input operations like reading strings will succeed, and you should use `cin>>s` to read the strings and skip whitespace. You can assume the input has exactly the right format and does not have extra strings or missing data, and the store and product names consist only of alphabetic characters. You may also assume there are no duplicate "store product" entries in the input data. Strings must match exactly to be equal – you do not need to convert strings to lower-case letters or anything like that – i.e., keep it simple.

Write your code in the space on the following page. You may assume that all necessary `#include` statements are provided for whatever standard libraries you wish to use, but you do need to declare appropriate variables to hold data as you read and process it.

Hints: various STL containers like `map` might be particularly useful. Basic reference information about STL is included on the last page of this exam.

(continued on next page)

Question 1. (cont.) Write your code for this problem below:

```
// assume all necessary libraries are #included
using namespace std;

int main(int argc, char** argv) {
   // add your code below









}
```

Question 2. (18 points) A different twist on the usual inheritance question. Consider the following program and answer questions about it on the next page. This program prints some of the things that someone playing the card game poker might say during the game. But don't worry – you don't need to know anything about poker to answer this question.

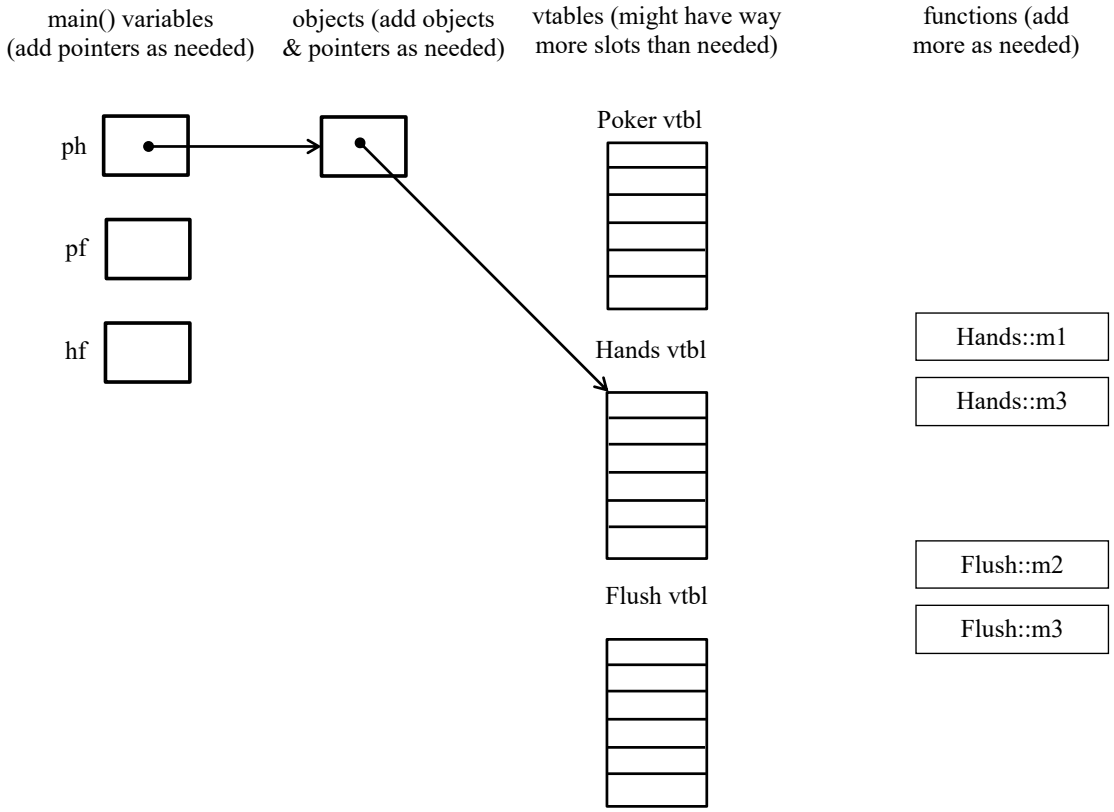
```
#include <iostream>
#include <cstdlib>

using namespace std;

class Poker {
 public:
  /* to be supplied if needed */









};

class Hands : public Poker {
 public:
          void m1() {std::cout << "I'm ";}
  virtual void m3() {std::cout << "not ";}



};

class Flush : public Hands {
 public:
  virtual void m2() {std::cout << "so ";}
          void m3() {std::cout << "in!";}



};

int main(int argc, char** argv) {
  Poker* ph = new Hands();
  Poker* pf = new Flush();
  Hands* hf = new Flush();

  ph->m1();
  pf->m2();
  hf->m3();

  return EXIT_SUCCESS;
}
```

Continue with the problem on the next page.

Question 2. (cont.) (a) (4 points) When we try to run the program on the previous page, it won't compile. What, precisely, are the errors in the code that prevent it from compiling?

(b) (6 points) On the previous page, make changes to the code so that it will compile and so it will print out the following three strings when it is executed: `I'm all in!`. You may not change or delete any of the functions currently in the program – in particular, they must continue to print whatever they print in the original code. You need to decide what new function(s) to add, if any, what they should print, and whether they are `virtual` or not. Hint: you may find it helpful to work on this part of the question while also working on part (c) below.

(c) (8 points) Complete the diagram below to show all of the variables, objects, virtual method tables (vtables) and functions in the program after making your changes in part (b). Parts of the diagram are supplied for you.



Question 3. (10 points) We know that C++ smart pointers can handle some memory management tasks for us automatically. However, they have to be used properly.

(a) (5 points) Suppose we have the following `Node` definition, which uses `shared_ptrs` for links used to create double-linked lists and similar data structures.

```
struct Node {
  int data;
  shared_ptr<Node> next;
  shared_ptr<Node> prev;
};
```

Will this prevent memory leaks if we have a `shared_ptr` variable pointing to the first node in a list? Give a brief justification or example for your answer.

(b) (5 points) One of our colleagues says we should use `weak_ptrs` instead of `shared_ptrs` for the backward (`prev`) links in the nodes:

```
struct Node {
  int data;
  shared_ptr<Node> next;
  weak_ptr<Node> prev;
};
```

Will this prevent memory leaks if we have a `shared_ptr` variable pointing to the first node in a list? Give a brief justification or example for your answer.

Question 4. (12 points) Templates and Boxes. Here is a small program that defines and uses a class `Box` that holds items that have integer weights. Each `Box` has a maximum weight capacity that is set when it is created, and we keep track of the remaining capacity as items are added to the `Box`.

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

/* A box with a fixed weight limit */

class Box {
 public:
  // Construct new Box with default max weight 10
  Box() : remaining_weight_(10) {  }

  // Construct new Box with given max weight
  Box(int w): remaining_weight_(w) { }

  // Add item with given weight w to this Box.
  // Return 1 (success) if successful, return 0 if not enough
  //   capacity in the box for the new item.
  int add(int w) {
    if (w <= remaining_weight_) {
      remaining_weight_ -= w;
      return 1;
    } else {
      return 0;
    }
  }

  // return remaining weight capacity in this box
  int remaining() {
    return remaining_weight_;
  }

 private:
  int remaining_weight_;  // remaining weight capacity in this Box
};

/* main program to use a Box */
int main(int argc, char** argv) {
  Box b(5);
  cout << b.add(3) << endl;
  cout << b.remaining() << endl;
  return EXIT_SUCCESS;
}
```

We've decided that this `Box` class would be much more useful if would work with weight values that have any arithmetic type that supports the necessary operations (<=, -=, etc.). Write in changes to the above code to make this into a class template where the weight values could be any suitable type `T`, and adjust the main program to create a `Box` with weights that have values of type `double` by using appropriate type parameter values if needed.

Question 5. (18 points) Sockets & strings – Oh My! We're working on a web program and need to implement a function that reads bytes from a socket and locates and returns a data string extracted from the input stream. We assume the data that we want is immediately preceded by the contiguous (no spaces) characters `#start#` and immediately followed by the characters `#end#`. So, for example, if the input stream is

```
Stuff #start# #stop# this! is not the #end # yet#end# stuff
              ----------------------------------
```

then the function should return all of the characters between the markers, including any whitespace or special characters, but not including the markers. The data returned from this stream is underlined above, and includes an initial space that immediately follows `#start#`. Note that the data string can contain some of the characters that can appear in the marker `#end#`, but only the first complete occurrence of `#end#` marks the data end.

Assumptions: You may assume that the input stream will contain exactly one copy of the markers `#start#` and `#end#` with zero or more characters between them. The characters between the markers in the stream should be returned as a C++ `string`, which might be an empty string if there are no characters between `#start#` and `#end#`.

As with the hw4 server, you will need to use the POSIX `read` function to read bytes from the stream, and your code needs to handle possible error return values from `read`, including `EAGAIN` and `EINTR`. Also, as with hw4, it is possible that a single read operation may return only part of the data. In particular, a single read might end with only part of the `#start#` or `#end#` markers, with the rest of the `#start#` or `#end#` marker appearing in the input the next time we call `read`.

Once your function has read all of the characters surrounded by the `#start#` and `#end#` markers it should return a `string` containing those characters, without the markers, as its result. You do not need to close the stream or do anything else with it. However, if it simplifies things, you can read as much additional data from the stream as you wish – maybe even reading until reaching end of file and then extracting the message – whatever is easiest.

You should assume that all necessary library function headers have been `#included`, and that `using namespace std;` has also been included in the code. You do not need to write these.

Hint: the information about C++ `string`s on the reference page at the end of the exam might be useful.

(continued on next page)

Question 5 (cont.) Write your implementation of function `ReadData` below.

```
/* Read from the socket file descriptor sfd and return
   a string containing the characters that are surrounded
   by the markers #start# and #end#.  The markers are not
   included in the returned string. */
string ReadData(int sfd) {




}
```

Question 6. (16 points, 2 points each part) Concurrency. Consider the following C program that creates a pair of threads and executes them. This program does compile and execute without errors.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int x = 0;

void * thread_worker(void * ignore) {
  x = x + 1;
  printf("x=%d\n", x);
  return NULL;
}

int main() {
  pthread_t t1, t2;
  int ignore;
  ignore = pthread_create(&t1, NULL, &thread_worker, NULL);
  ignore = pthread_create(&t2, NULL, &thread_worker, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("final x=%d\n", x);
  return EXIT_SUCCESS;
}
```

For each of the following output sequences, circle "yes" if it could be produced by some possible execution of the above program and circle "no" if it could never happen under any circumstances. (Hint: at least one of these sequences is possible.)

(a) YES NO `x=0 x=1 final x=1`

(b) YES NO `x=1 x=1 final x=1`

(c) YES NO `x=1 x=2 final x=1`

(d) YES NO `x=1 x=1 final x=2`

(e) YES NO `x=1 x=2 final x=2`

(f) YES NO `x=2 x=2 final x=1`

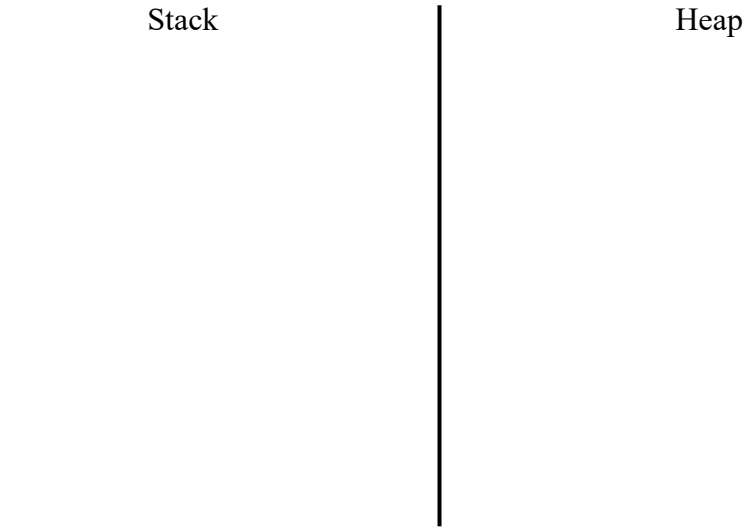
(g) YES NO `x=2 x=1 final x=2`

(h) YES NO `x=2 x=2 final x=2`

Question 7. (16 points) The question we dropped from the midterm to save time! One of your colleagues has some questions about C++ classes. Their example is the following code, which doesn't do very much, but does include the required constructors, destructor, and assignment for a typical class:

```
#include <cstdlib>
class Ogre {
 public:
  Ogre(){ name_ = new char[10]; }
  Ogre(const Ogre &other) = default;
  Ogre& operator=(const Ogre &other) = default;
  ~Ogre(){ delete [] name_; }
 private:
  char* name_;
};

int main(int argc, char **argv) {
  Ogre shrek;                       // Line 1
  Ogre* fiona = new Ogre();         // Line 2
  shrek = *fiona;                   // Line 3
  delete fiona;                     // Line 4
  return EXIT_SUCCESS;              // Line 5
}
```

(a) (12 points) Below, draw a diagram showing the contents of memory immediately after `Line 3` is executed. Be sure to carefully distinguish between local (stack) variables and heap-allocated memory. Variables that are local to `main` should be drawn in a box labeled main on the stack.



(b) (4 points) Does this program execute successfully when it runs? If not, what goes wrong and why? (A couple of sentences should be more than enough.)

Question 8. (2 free points – all answers get the free points) Draw a picture of something you're planning to do this summer!

Congratulations on lots of great work this quarter!
Have a great summer and best wishes for the future!!
The CSE 333 staff

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

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

C++ strings: If `s` is a string, `s.length()` and `s.size()` return the number of characters in it. `s.find(`*search_string, start_pos* = 0`)` returns the location of the first occurrence of *search_string* starting no earlier than *start_pos* or returns `string::npos` (string length) if not found. `s.substr`(*pos* = 0, *len* = *npos*) returns a new string that starts at location *pos* of `s` with *len* characters, or until the end of the string if that comes first. `s.append`(*t*) appends string *t* to `s`; `s.append`(*t*, *pos*, *len*) appends the substring of *t* starting at *pos* with *len* characters (or length of string if less then *len*) to `s`. `s+t` returns a new `string` that is the concatenation of `strings` `s` and `t`. Subscripts: `s[i]` can be used to access individual characters. C-strings: `s.c_str()` returns a `char*` pointer to a heap-allocated, `\0`-terminated C-string copy of `s`.

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.find`(*item*) – iterator pointing to element with key that matches *item* if found, or `m.end()` if not found.
  - `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).
  - The elements of a `Pair p` can be accessed as `p.first` and `p.second`.
- 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)

More reference information. You should remove this page from the exam.

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(pthread_t *thread, attr, start_routine, arg)
- pthread_exit(void *status_ptr)
- pthread_join(pthread_t thread, void **value_ptr)
- pthread_cancel (pthread_t thread)
- pthread_detach(pthread_t 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)