University of Washington – Computer Science & Engineering

Spring 2019 Instructor: Justin Hsia 2019-06-12

CSE333 FINAL

Last Name:	**Perfect**
First Name:	**Perry**
Student ID Number:	1234567
Name of person to your Left | Right	Samantha Student | Larry Learner
All work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CSE333 who haven't taken it yet. Violation of these terms could result in a failing grade. **(please sign)**	

Do not turn the page until 12:30.

Instructions

- This exam contains 14 pages, including this cover page. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, wearable devices, or calculators). You are allowed two pages (US letter, double-sided) of *handwritten* notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, headphones, and watches.
- You have 110 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read *all* questions first and start where you feel the most confident.
- Relax. You are here to learn.

Question	1	2	3	4	5	6	Total
Possible Points	15	18	24	18	20	22	**117**

Question 1: Potpourri – Nice to Smell, Hard to Spell [15 pts]

(A) Will the following lines of C++ code compile? *Briefly* explain. [3 pt]

Code	Answer	Explanation
`int &* a1;`	Y / (N)	A reference is an alias, not a physical location, so you can't point to a reference.
`int *& a2;`	Y / (N)	This is a reference to a pointer, which makes sense and is legal, but a reference must be initialized.

(B) Circle the outcome of the following lines of code with `const int* cip`. [3 pt]

Code	Outcome
`const int** b1 = &cip;`	(Compiles) / Invalid syntax / Violates constness
`int* const * b2 = &cip;`	Compiles / Invalid syntax / (Violates constness)
`int** const b3 = &cip;`	Compiles / Invalid syntax / (Violates constness)

`b1` is a pointer to a pointer to a const `int`. `b2` is a pointer to a const pointer to an `int`. `b3` is a constant pointer to a pointer to an `int`.

(C) The following C casts compile and run without errors. Which type of cast is the *most appropriate* equivalent in C++ code? [3 pt]

```
const int i = (int) 4.2;        ___static_cast____
int* iptr = (int*) &i;          ____const_cast____
float* fptr = (float*) iptr;    _reinterpret_cast_
```

The 1st does the standard conversion from `float` to `int`.
The 2nd strips away the const-ness from `&i`, which is `const int*`.
The 3rd is reinterpreting the address as a different kind of pointer.

(D) *Briefly* describe what **slicing** is and why it is a problem with STL containers. [4 pts]

Slicing: Slicing is when you assign a derived object to a base object and you can't copy the data outside of the subobject corresponding to the base class.

Containers: STL containers copy-by-value, so adding or assigning a derived object into a container that holds the base class causes slicing.

(E) *Briefly* explain whether the Ethernet payload or IP payload takes up a larger percentage of the overall packet sent over the network. [2 pt]

(circle one)	Explain
(Ethernet) / IP	The IP payload (network layer) is contained within the Ethernet payload (data link layer), so it takes up a smaller percentage of the overall packet.

Question 2: A NETWORKING Event For You [18 pts]

(A) What does a **return value of 0** mean for the following network programming functions? [6 pt]

getaddrinfo():	Success
socket():	Valid file descriptor, but `stdin` must have been closed earlier.
bind():	Success
read():	Connection closed for networking (EOF for file I/O).

(B) Layers upon layers of structs! Fill in the blanks using the word bank below: [6 pt]

addrinfo	AF_INET	AF_INET6	in_addr
in6_addr	in_port_t	sa_family_t	ss_family_t
sockaddr	sockaddr_in	sockaddr_in6	sockaddr_storage

An **IPv6** address (*just the address*) is stored in a `struct` ____`in6_addr`____.

The above IPv6 address, plus other info, is stored inside a `struct` __`sockaddr_in6`__.

The above struct fits inside a more general `struct` `sockaddr_storage`.

(C) We are writing *single-threaded* server-side networking code that doesn't use file I/O. Using comments/pseudocode, describe what we should do *in order* if we detect an *unrecoverable* error during `write()` based on the suggested style used in this course. You should write out something for each *line* of code you would expect to write, but it does not need to be properly formatted (*e.g.* just a description is fine). [6 pt]

Two "families" of responses were given full credit: (1) the error is such that we want to exit the process and (2) the error is deemed localized to the client and we want to continue accepting other connections.

```
// Family 1:
// write a useful error message to stderr/cerr
// close the active connection socket (from accept())
// close the listening socket (from socket())
// exit or return from main with EXIT_FAILURE

// Family 2:
// (optional) error message - usually not if not terminating
// close the active connection socket (from accept())
// break out of writing loop
// reuse listening socket to do next accept()
```

Question 3: Let's Give Your PHONE Signal a Boost [24 pts]

Before smart phones, mobile phones used a predictive text system called **T9**, based on the mapping of a single numpad key to any of the corresponding letters shown in the image to the right. Note that the '1', '*', and '#' keys won't be used and that '0' corresponds to [Space].

Example: a user would type '8', then '4', then '3' to get the word "the", though it could also predict longer words like "the**y**" or "the**re**".

We will use **C++ STL** to generate our T9 predictive dictionary! The top of our file is shown below so that you are aware of what is globally available:

```
#include <iostream>
#include <string>
#include <vector>
#include <map>

using namespace std;
```

First, we will convert everything in a predefined index (*e.g.* imported from a file) of predictable words into the corresponding key presses (*e.g.* "the"→"843", "hello"→"43556").

(A) We will use a global map from *lowercase* letters to keys and a function to initialize this map. Complete just the requested portion below of that function: [4 pt]

```
map<char, char> letters_to_keys;

void InitLettersMap() {
  letters_to_keys['a'] = '2'______;   // key mapping for 'a'
  ...          // assume all other key mappings happen here
  letters_to_keys['z'] = '9'______;   // key mapping for 'z'
}
```

(B) Complete the function below to convert a string to lowercase *in place* (note the parameter type). Hint: use the `char tolower(char)` function. [3 pt]

```
void StrToLower(string * const input) {
  for (char& c : *input) {  // option 1: range for (ref is necessary)
    c = tolower(c);
  }
  // option 2: with iterators
  // for (auto it = (*input).begin(); it < (*input).end(); it++ ) {
  //   *it = tolower(*it);
  // }
}
```

We want the strings to be predictable from any of the *prefixes* (*i.e.* substring starting at the beginning) of their keyed versions. Example: `"the"` is predictable from `"8"`, `"84"`, and `"843"`.

(C) Complete the function to add a mapping from *each* prefix to the *lowercase* string. You may find the string member function `string substr(size_t pos, size_t len);` useful, which returns the substring of length `len` starting from position `pos`. [10 pt]

```
map<string, vector<string>> predictions;   // global prediction map

void AddPrefixesToPredictions(string * const word) {
  // Note: string word also works, just makes an extra copy of the
  // string object.  You would also need to remove one level of
  // indirection below (e.g. word --> &word, *word --> word).

  // lowercase the word
  StrToLower(word);

  // convert string to prefixes
  string prefix;
  for (auto c : *word) {
    prefix += letters_to_keys[c];
    predictions[prefix].push_back(*word);  // soln 1: immediately
                                           // push *word onto prefix
                                           // key
  }

  // soln 2: extra loop to push *word to onto all prefix keys
  // for (size_t i = 1; i <= prefix.length(); i++) {
  //   predictions[prefix.substr(0,i)].push_back(word);
  // }
}
```

(D) Complete the function below to print out the contents of `predictions`. For example, if we've added `"a"` and `"ax"`, it should print out the following (note the formatting): [7 pt]

```
2 : a, ax,
29 : ax,
```

```
void PrintPredictions() {
  // loop over every prediction pair
  for (auto& pred_pair : predictions) {
    cout << pred_pair.first << " : ";
    // loop over every vector entry
    for (auto& w : pred_pair.second) {
      cout << w << ", ";
    }
    cout << endl;
  }
}
```

Question 4: Get SMART With Your POINTERS [18 pts]

We are building a template for a **two-level array**, which is an array of pointers to row arrays. This is how Java implements 2D arrays! We will use smart pointers in order to help with the memory cleanup. Below is an attempt at a definition of our class, where `h` is the "height/rows" and `w` is the "width/columns" of our two-level array:

```
#include <iostream>
#include <memory>

using namespace std;

template <typename T, int h, int w> struct TwoLevelArray {
 public:
  TwoLevelArray();                                     // def ctor
  TwoLevelArray& operator=(const TwoLevelArray& rhs);  // op=
  T* operator[](size_t idx);                           // op[]

 private:
  unique_ptr<T[]> level_one_[h];  // array of unique_ptrs to arrays
};
```

(A) Complete the definition of the default constructor below. Note: you are not allowed to try to add an initialization list. [3 pt]

```
template <typename T, int h, int w>
TwoLevelArray<T, h, w>::TwoLevelArray() {
  for (int i = 0; i < h; i++) {
    // option 1: use reset on default constructed unique_ptr's
    level_one_[i].reset(new T[w]);

    // option 2: use move ctor (not expected to know this)
    //   and also changes your answer to part E
    // level_one_[i] = unique_ptr<T[]>(new T[w]);
  }
}
```

(B) The struct above uses the synthesized copy constructor. Will this work? *Briefly* explain your response. [3 pt]

(Circle one) Works? Yes **No** (circled)

Explain: The synthesized cctor copies all of the elements of the array data member, which would invoke the deleted cctor of `unique_ptr`.

SID: __1234567__

(C) Complete the definition of the assignment operator below *using good style.* [6 pt]

```
template <typename T, int h, int w>
TwoLevelArray<T, h, w>&
  TwoLevelArray<T, h, w>::operator=(const TwoLevelArray &rhs) {

  if (this != &rhs) {
    for (int i = 0; i < h; i++) {
      for (int j = 0; j < w; j++) {
        level_one_[i][j] = rhs.level_one_[i][j];
      }
    }
  }

  return *this;

}
```

(D) Complete the definition of the subscript operator so that it will return the address of the requested row. Hint: pay attention to the return type. [2 pt]

```
template <typename T, int h, int w>
T* TwoLevelArray<T, h, w>::operator[](size_t idx) {

  return level_one_[idx].get();

}
```

(E) What happens when we try to execute the following `main` function (circle one)? If there is an issue, give a *brief* explanation. If it works fine, how many unique pointers are constructed? [4 pt]

```
int main() {
  TwoLevelArray<int, 2, 3> ar;
  cout << "ar[0] = " << ar[0] << endl;
  ar[0][1] = 333;
  TwoLevelArray<int, 2, 3> ar_copy;
  ar_copy = ar;
  return EXIT_SUCCESS;
}
```

Compiler Error	Double Delete	Memory Leak	Segmentation Fault	Works Fine (circled)

Explanation/#: If you used `reset()` in part A, then there are 4 unique pointers constructed, 2 in `ar` and 2 in `ar_copy`. If you used a move ctor in part A, then there are 8 unique pointers constructed, 4 per TwoLevelArray ctor.

[Other answers accepted based on answers to part A and C]

Question 5: You Are Entitled To Your INHERITANCE [20 pts]

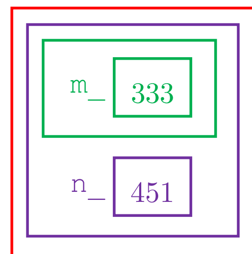
Consider the following **C++ classes**. The code below causes no compiler errors.

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

class A {
 public:
          void f1() { f2(); cout << "A::f1, "; }
  virtual void f2() {       cout << "A::f2, "; }

 private:
  int m_ = 333;
};

class B : public A {
 public:
          void f1() {       cout << "B::f1, "; }
  virtual void f3() {       cout << "B::f3, "; }

 private:
  int n_ = 451;
};

class C : public B {
 public:
  virtual void f1() {       cout << "C::f1, "; }
          void f2() { f3(); cout << "C::f2, "; }
};
```

(A) Draw a conceptual diagram of a default-constructed object of `class C` below. Don't show `vptr`'s. [2 pt]

Orientation and sizing doesn't matter

- **Class C object**
- **Class B subobject**
- **Class A subobject**



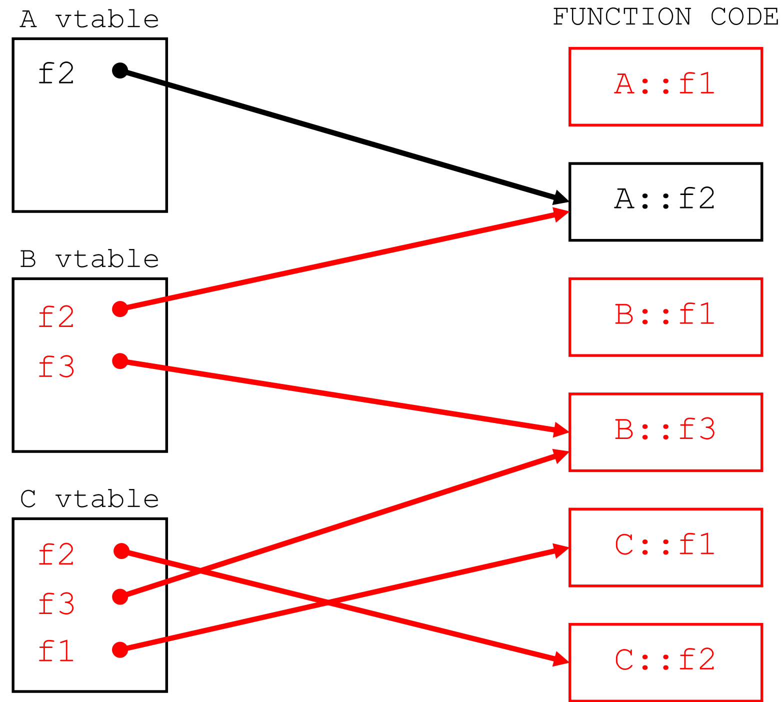
(B) List out *in order* all functions (synthesized or defined) that are called during the execution of the code below. Make sure to use the full `Class::Function` names. [3 pt]

```
A* ap = new B;
delete ap;
```

```
B::B()    // def ctor called by new
A::A()    // def ctor called on A subobject
A::~A()   // dtor is statically dispatched (ap)
```

SID: __1234567__

(C) Complete the **virtual function table diagram** below by adding the remaining class methods on the right and then drawing the appropriate function pointers from the vtables. *Ordering of the function pointers matters!* One is already included for you. [7 pt]



(D) Assume we have objects and pointers as defined in the five lines of code below. Then, for each row of the table below, **fill in the result** on the right, which should either be the corresponding `stdout` output, "compiler error," or "runtime error." [8 pt]

```
B b;           // object instances
C c;
A *ap1 = &c;   // pointers
B *bp1 = &b;
B *bp2 = &c;
```

`bp1->f1();`	B::f1, (static)
`bp1->f2();`	A::f2, (dynamic inherited)
`bp2->f2();`	B::f3, C::f2, (dynamic f2, dynamic inherited f3)
`bp2->f3();`	B::f3, (dynamic inherited)
`ap1->f1();`	B::f3, C::f2, A::f1, (static f1, dynamic f2, dynamic f3)
`ap1->f3();`	compiler error (A has no f3)

Question 6: Staying A-THREAD Of The Game [22 pts]

We are going to write a *multithreaded* **C program** that will count all of the instances of a specified integer in an integer array. The top of the file is given below:

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

#define NUM_THREADS 4
#define AR_SIZE 100
#define COUNT_ME 2
static int global_count = 0, ignore;
static pthread_mutex_t count_lock;

typedef struct {
  int *ar_ptr;    // array pointer
  int elements;   // number of consecutive elements to check
} thd_arg;
```

(A) Complete the function below that the children threads will run. Assume that the passed values are *found on the Heap*. [8 pt]

```
// Adds the # of occurrences of COUNT_ME in a portion
// of the array to global_count and cleans up.
// Assumes count_lock is initialized already.
void *ThreadMain(void *arg) {

1    thd_arg *a = (thd_arg *) arg;
2    int local_count = 0;

3    for (int i = 0; i < a->elements; i++) {

4      if ( a->ar_ptr[i] == COUNT_ME ) {
5        local_count++;
6      }
7    }

8    pthread_mutex_lock(&count_lock);

9    global_count += local_count;

10   pthread_mutex_unlock(&count_lock);

11   free(a);

12   return NULL;
}
```

(B) How would you expect this multithreaded code to perform compared to a sequential version of the code in the following scenarios? *Briefly* explain your choices. [4 pt]

Single CPU:	Faster	About the Same	(Slower)
Multiple CPUs:	(Faster)	About the Same	Slower

Explanation: Single CPU is executing the counting sequentially (might even have worse spatial locality), but with additional thread overhead (creation, switching).

Multiple CPUs can run local sums in parallel and won't fight too much over the critical section, which is accessed just once per thread.

It turns out that we can avoid using a mutex altogether if we make use of the return value from `ThreadMain`!

(C) What changes need to be made to the following lines in `ThreadMain`? (other minor changes are needed but are not part of this question) [4 pt]

Line 2:	`_int *_ local_count = (int *) calloc(1, sizeof(int));`
Line 12:	`return (void *) local_count;`

We need to return the address of an `int`, so that `int` can't be on the Stack. So we convert `local_sum` to a pointer to a heap-allocated `int` and return its address.

(D) Complete the portion of `main` below that will sum up the local counts of each thread. Here, we allow you to skip error checking by storing into the `ignore` variable. [6 pt]

```
pthread_t thds[NUM_THREADS];  // array of thread ids

_int *_ retval;              // declare a needed variable

...                           // threads created here

// Wait for all child threads to finish and add their counts
// to global_count.
for (int i = 0; i < NUM_THREADS; i++) {
  ignore = pthread_join(thds[i], (void **)&retval);
  global_count += *retval;
  free(retval);
}
```

`retval` is an output parameter to get the value returned from `ThreadMain`, so it needs to be an `int *` to match and we pass the address of it (casted to `void **`). Because it was dynamically-allocated in `ThreadMain`, we must free it afterwards.