University of Washington – Computer Science & Engineering

Spring 2018 Instructor: Justin Hsia 2018-06-05

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	24	16	16	19	24	16	**115**

Question 1: Potpourri – Nice to Smell, Hard to Spell [24 pts]

(A) Name *two* benefits to utilizing **const correctness** in C++. [4 pt]

It makes sure that you don't change what you intend not to.
It allows us to catch logical errors at *compile* time instead of run time.

(B) Are the following statements about **C++ templates** true or false? Answer T/F. [4 pt]

A template parameter must be a data type.
Can be a non-type like `int`. __F__

Using a template function instead of function overloading *doesn't* decrease the amount of machine code in your program.
Every time you call a template with a different parameter, the compiler expands the code to be included in your program. __T__

Modularizing your code (creating header files and object code for distribution) is the same with and without templates.
We need the template implementation to be visible/present for code expansion. __F__

The `template` keyword does not need to be on the same line as the function or class name. __T__

(C) If `class D` is derived from `class B` and we have object instances `B b_obj` and `D d_obj`, **will the following C++ casts cause errors** (either compile-time or run-time)? Answer Y/N. [4 pt]

```
static_cast <D *> (&b_obj)          __N__
dynamic_cast <D *> (&b_obj)         __N__
dynamic_cast <B *> (&d_obj)         __N__
reinterpret_cast <B *> (&d_obj)     __N__
```

The second line *fails* at runtime, but instead of crashing your program, it just returns `nullptr`.

(D) **The Internet** [4 pts]

How many *times* more IPv6 addresses are there compared to IPv4? Answer as a multiple.

2^{32} IPv4 addresses (4 bytes) and 2^{128} IPv6 addresses (16 bytes). **2^{96}**

Circle one per row:

How many IP addresses can a host be associated with?	**One**	**Multiple**
How many hosts can be associated with an IP address?	**One**	**Multiple**
How many results can a DNS lookup return?	**One**	**Multiple**
How many MAC addresses can a NIC have?	**One**	**Multiple**

(E) For the following **HTTP headers**, circle if they are used in requests or responses and *briefly* explain why that header is important. [4 pt]

`Content-Type` Used in: Request **Response**	
Importance: So the browser knows how it should handle/display the response payload (*e.g.* displaying an HTML file, PDF, image).	
`User-Agent` Used in: **Request** Response	
Importance: So the server can deliver content differently based on the browser being used (*e.g.* mobile vs. desktop versions of a webpage).	

(F) Complete the table below to compare forking processes and dispatching/spawning threads with `pthread`. [4 pt]

	Threads	**Processes**
Function to create	`pthread_create`	`fork`
Function for parent to get child's "return value"	`pthread_join`	`wait` or `waitpid`
Where does the child start code execution?	`*start_routine` or `start_routine(arg)`	the return from `fork`

Question 2: C++ Standard Template Library [16 pts]

We are investigating a social site like Facebook, where connections are **bidirectional** (*e.g.* a friendship between Justin and Hal means that Justin is Hal's friend *and* Hal is Justin's friend).

(A) Given a user and their friend list (a vector of strings), we want to return all associated friendship links as `pair`s, where the pair (p1, p2) represents that p1 is p2's friend. Implement the function `friendPairs()` below. Hint: `auto` will save you writing. [7 pt]

```
#include <string>
#include <vector>
using namespace std;

vector<pair<string,string>> *friendPairs (string user,
                                          vector<string> friends) {
  auto friendships = new vector<pair<string,string>>;
  for (const string &f : friends) {
    friendships->push_back(pair<string,string>(user,f));
    friendships->push_back(pair<string,string>(f,user));
  }
  // for (auto it = friends.cbegin(); it != friends.cend(); it++) {
  //   friendships->push_back(pair<string,string>(user,*it));
  //   friendships->push_back(pair<string,string>(*it,user));
  // }
  return friendships;
}
```

(B) We want to print our function results to `stdout` using the **`for_each()`** algorithm, which takes an iterator range and function pointer. Create a function to print out pairs in the format "(p1, p2)" and then fill in the call to **`for_each()`** [7 pt]

```
// define your function here
void PrintPair(const pair<string,string> &p) {
  cout << "(" << p.first << ", " << p.second << ")" << endl;
}

int main() {
  vector<string> friends({"Adam", "Hal", "Ruth"});      // this works
  // yes, this leaks memory, but I couldn't squeeze in the delete
  auto result = friendPairs(string("Justin"), friends);

  _for_each(result->cbegin(), result->cend(), &PrintPair)_;  // print
  return 0;
}
```

(C) Duplicate friendships eventually show up in our data. Name a container we could move our data into that will automatically remove duplicates for us. [2 pt]

`map` or `set`

SID: _____

Question 3: Network Programming [16 pts]

(A) Complete the following inequalities for the relative "heights" (higher is "larger") of the following network layers: [2 pt]

Application __>__ Physical Network __<__ Transport

(B) *Briefly* define the following: [2 pt]

Host byte order: The endianness of the host machine.
Network byte order: The defined endianness for networking (big endian).

(C) Name two *server-side* programming functions that return file descriptors. [2 pt]

`socket()`	**`accept()`**

(D) *Briefly* explain the effects of **`socket()`** and **`bind()`** on the OS descriptor table. [4 pt]

`socket()`: Allocate/create a new file descriptor entry.
`bind()`: Update the descriptor table entry (address info – IP, port, etc.).

(E) *Briefly* explain why the address family (of type `sa_family_t`) is always the first field in the socket-related structs. [2 pt]

So we can always find those associated bytes in the same position and can use them to check for which socket struct to cast to in order to properly read the rest of the data.

(F) Name one advantage and one disadvantage to using a **non-blocking** socket instead of a **blocking** socket for network communications. [4 pt]

Advantage: It doesn't hold up/stall your program because the read/write function calls return immediately.
Disadvantage: The expected work may not have been done/finished. You may need to check for when work is available first.

Question 4: Smart Pointers and Templates [19 pts]

A shared pointer will only increase its reference count when the copy constructor or assignment operator is invoked (*i.e.* a shared pointer's managed pointer is set from *another* shared pointer).

(A) Complete the main function below we've written to test this fact. Fill in the 4 statements involving shared pointers as well as the blanks in the program output. [6 pt]

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

using namespace std;

int main() {
  // create a shared pointer to the int 3.

  shared_ptr<int> p1(new int(3)_____________________________;
  cout << "p1.use_count() = " << p1.use_count() << endl;

  // test copy constructor.  or: shared_ptr<int> p2 = p1

  shared_ptr<int> p2(p1)____________________________________;
  cout << "p2.use_count() = " << p2.use_count() << endl;

  // create a shared pointer to the same int that doesn't
  //   increase the reference count.  = doesn't work here

  shared_ptr<int> p3(p1.get())______________________________;
  cout << "p3.use_count() = " << p3.use_count() << endl;

  // test assignment operator to update p3.  or: p3 = p2

  p3 = p1___________________________________________________;
  cout << "p3.use_count() = " << p3.use_count() << endl;
  return 0;
}
```

Program output:

```
p1.use_count() = __1__

p2.use_count() = __2__

p3.use_count() = __1__

p3.use_count() = __3__
```

Interestingly, `*p3` (and `*p1` and `*p2`) at the end is 0. The `int` was `delete`'d when the assignment operator was called on `p3` – its managed pointer dropped to a ref count of 0.

Let's examine a **singly-linked list**. Assume that all necessary headers are included and we are `using namespace std`.

(B) Define a struct template named `Node` that uses shared pointers for its fields `value` and `next`. Include a *declaration* for a two-argument constructor that takes a shared pointer for the next node and a raw pointer for the value. [6 pt]

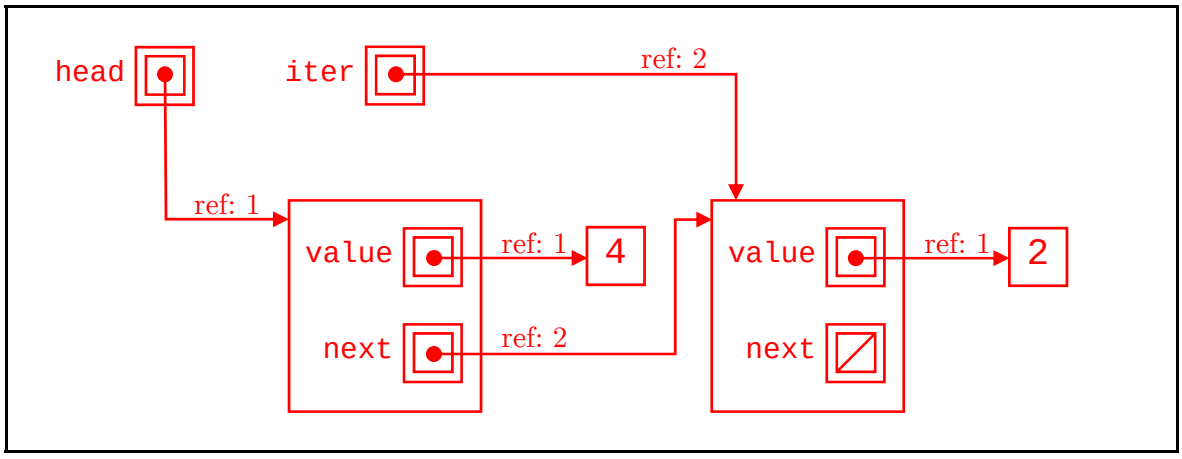
```
// by default, all members of a struct are public
template <typename T> struct Node {

  Node(shared_ptr<Node<T>> node, T *val);

  shared_ptr<T> value;
  shared_ptr<Node<T>> next;

};
```

(C) Assume we have the function defined below to add a new node at the beginning of the list:

```
template <typename T>
shared_ptr<Node<T>> push(shared_ptr<Node<T>> head, T *val) {
  return shared_ptr<Node<T>>( new Node<T>(head, val) );
}
```

Assume we execute the following lines of code. Draw a memory diagram that includes the reference count of each smart pointer as "ref #" on the corresponding arrow. [7 pt]

```
shared_ptr<Node<int>> head;
head = push<int>(head, new int(2));
head = push<int>(head, new int(4));
shared_ptr<Node<int>> iter(head->next);
```



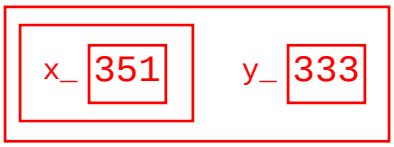
Question 5: C++ Inheritance [24 pts]

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

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

class A {
 public:
  virtual void f1() {       cout << "A::f1" << endl; }
          void f2() { f1(); cout << "A::f2" << endl; }
 protected:
  int x_ = 351;
};

class B : public A {
 public:
          void f1() {       cout << "B::f1" << endl; }
  virtual void f3() {       cout << "B::f3" << endl; }
 protected:
  int y_ = 333;
};

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

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

Subobject of class A
within class B object.

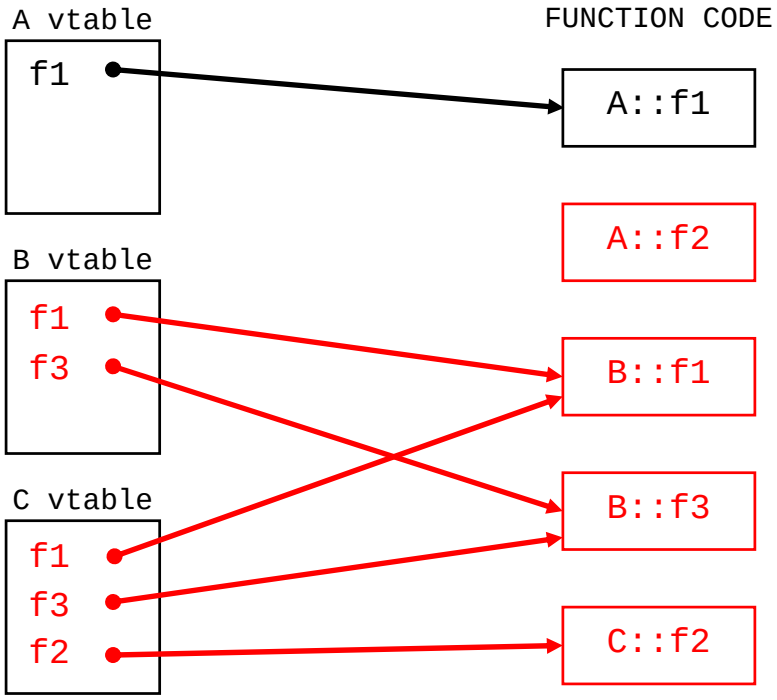


(B) We wish to write constructors for class A and class B to help us initialize our data members. Complete the definitions below: [3 pt]

```
A::A(int x) : x_(x) { }
```

```
B::B(int x, int y) : A(x), y_(y) { }
// must call base class constructor in initializer list
```

(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. [9 pt]



(D) Assume we have objects and pointers as defined in the two 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, "compile error," or "runtime error." [10 pt]

```
A a; B b; C c;                          // object instances
A *ap1 = &a; A *ap2 = &b; B *bp1 = &c;  // pointers
```

`ap1->f1();`	`A::f1`
`ap1->f2();`	`A::f1` `A::f2`
`ap2->f1();`	`B::f1`
`ap2->f3();`	`compile error` (A has no `f3`)
`bp1->f2();`	`B::f1` (static dispatch of `f1`) `A::f2`
`bp1->f3();`	`B::f3`

Question 6: Pthreads [16 pts]

Consider the **C program** below that uses pthreads and compiles and executes without error.

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

1  int x = 3, ignore;

2  void *task1(void *p) {
3    x -= 1;
4    return NULL;
5  }

6  void *task2(void *p) {
7    x *= 2;
8    return NULL;
9  }

10 int main() {
11   pthread_t t0, t1;
12   ignore = pthread_create(&t0, NULL, &task1, NULL);
13   ignore = pthread_create(&t1, NULL, &task2, NULL);
14   pthread_join(t0, NULL);
15   pthread_join(t1, NULL);
16   printf("%d\n", x);
17   return 0;
18 }
```

(A) List ALL possible printed values of this program if it is run as is. Separate the possible values with commas in the box below. [4 pt]

2, 4, 5, 6

read 3, read 3, write 6, write 2 → **2**

read 3, write 2, read 2, write 4 → **4**

read 3, write 6, read 6, write 5 → **5**

read 3, read 3, write 2, write 6 → **6**

(B) We will **add lock synchronization** to prevent the threads from interfering with each other. We will add the commands shown in the table below. In the right column, fill in the *half line position(s)* where we will insert the command (*e.g.* "16.5" would mean just before `return 0;` in `main`). [6 pt]

pthread Command	**Insert At Line(s)**
`static pthread_mutex_t lock;`	0.5 (or 1.5)
`pthread_mutex_init(&lock, NULL);`	11.5 (or 10.5)
`pthread_mutex_lock(&lock);`	2.5, 6.5
`pthread_mutex_unlock(&lock);`	3.5, 7.5

(C) After adding lock synchronization, how *many* printed values are still possible? [2 pt]

4 and 5 still possible (swap order of thread execution).

2

(D) Even without lock synchronization, we can guarantee a single possible output by moving a single line from our original code. Indicate which line to move and which half line position to move it to: [2 pt]

Move Line __**14**__ to _**12.5**_

Equivalently, Move Line 13 to 14.5.

Yes, also possible to put both "work" statements in the same thread:

- Move Line 3 to 6.5 or 7.5
- Move Line 7 to 2.5 or 3.5

Yes, also possible to move the `printf()` statement:

- Move Line 16 to 10.5 or 11.5

(E) *Briefly* describe what is problematic about the solution to part D. [2 pt]

Worse than sequential. We create a thread, but then wait until it finishes before we create the next one.

Creating one thread that does no work involves unnecessary overhead.

Moving the `printf()` means the output doesn't reflect the work done.