12sp

Problem 1: Multiple Choice Madness (24 points)

Circle exactly one answer for each of the following questions:

i. It is possible for C code to determine the endian-ness of the underlying CPU.

a) true b) false

ii. In C, a pointer is a variable that contains an address. If you add 2 to a pointer, then:

a) the resulting value is the address plus 2

b) the resulting value depends on what value the pointer points to

c) the resulting value depends on the type of the pointer

d) a segmentation fault is thrown

iii. When you pass a struct as an argument to a C function, then:

a) the struct is passed by value (i.e., a copy of the struct is made, including copying each field in the struct))

b) the struct is passed by reference (i.e., a pointer to the struct is passed)

c) a compiler error is thrown, since you cannot pass structs as arguments

d) what happens depends on the type of fields in the struct

iv. When you pass an array as an argument to a C function, then:

a) the array elements are passed by value (i.e., a copy of the array is made, including copying each element of the array)

b) since arrays are really just pointers, a pointer to the first element of the array is passed and no array elements are copied

c) a compiler error is thrown, since you cannot pass arrays as arguments

d) what happens depends on the type of the array

v. The purpose of a header guard is to:

a) prevent more than one .c file from including a particular .h file

b) prevent the header file from being included indirectly, as a side-effect of including some other .h file that includes it

c) document the contents and purpose of the header file

d) prevent the header file from being included twice, directly or indirectly

vi. A C++ reference:

a) serves as an alternative name for an object or variable (i.e., is an alias)

b) serves as a pointer to an object or variable

c) cannot be used as a parameter of a function

d) cannot be passed as an argument to a function

vii. What does "const" in the following code imply?

```
void foo (const int *x) {   …   }
```

a) the value of the pointer "x" cannot be changed inside the function foo

b) the function foo cannot have any side-effects

c) nothing; const in this case has no effect

d) the value that the pointer "x" points to cannot be changed inside the function foo

viii. What does "const" in the following code imply?

```
void Foo::bar (int *x) const {   …   }
```

a) the method bar( ) cannot mutate any of its parameters

b) the method bar( ) cannot have any side-effects at all

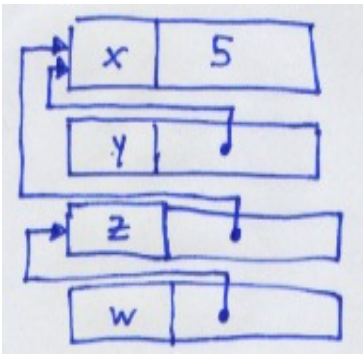
c) the method bar( ) cannot mutate any of Foo's state

d) the method bar( ) can only invoke const-y functions and methods

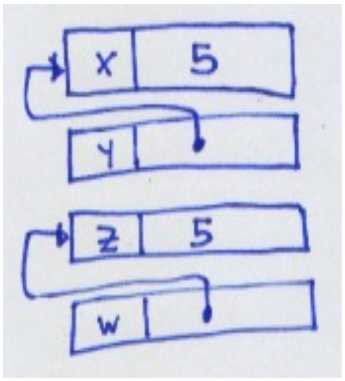
ix. Which of the following box and arrow diagrams correctly represents the following code?

```
int x = 5;
int *y = &x;
int &z = x;
int *w = &z;
```

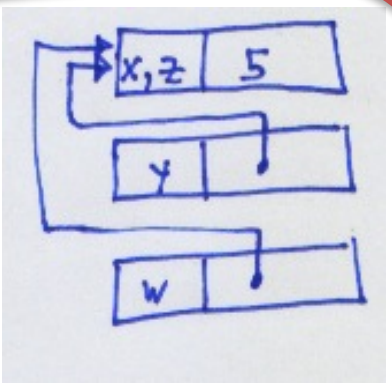
a)



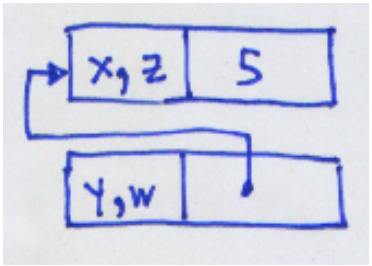
b)



c)



d



x. The destructor of an object that is heap-allocated:

a) is invoked when the function in which it is allocated returns

b) is never invoked

c) must be invoked manually

d) is invoked when somebody uses "delete" to deallocate the object

xi. A vtable:

a) exists for each class, and contains a function pointer for each method in the class

b) exists for each class, and contains a function pointer for each virtual method in the class

c) exists for each object instance, and contains a function pointer for each method in the object's class

d) exists for each object instance, and contains a function pointer for each virtual method in the object's class

xii. Slicing occurs when:

a) the value of a derived class is assigned to an instance of a base class

b) a pointer to a derived class is cast to, and assigned to, a pointer to a base class

c) an N-element array is assigned to an M-element array, where M<N

d) an element is deleted from a std::vector

Question 2. (22 points) Templates, STL, function pointers, and smart pointers all at once! – oh my!!! Don't panic – the answers to this question are actually quite short.

Sometimes a function is called many times to recomputed values. If that turns out to be expensive we may be able to save time by storing previously computed values in a cache and reuse them instead of computing them from scratch each time.

For this problem we want to implement parts of a class named `FunctionCache`. The idea is that an instance of `FunctionCache` stores a pointer to a function *f* and a map of <argument, result> pairs, where the result in each pair is computed by applying function *f* to the corresponding argument. The class also includes a function `apply` that is used by clients to compute values instead of of calling *f* directly. When `apply(x)` is called, it looks in the map to see if x is stored there as a key. If so, it returns the result found in the map without calling function *f*. If x does not appear as a key in the map, then `apply` calls *f* to compute *f*(x), then stores <x, *f*(x)> in the map for future use, and finally returns the result to the caller.

Because we want this to work for all single-argument functions, the class is a template whose parameters are the argument and result types of the function. Here is the main part of this template, except for the `apply` function code:

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

// Wrapper class for a function with argument type T and
// result type U.
template <typename T, typename U>
class FunctionCache {

public:
  // construct a new FunctionCache.  Argument f is a pointer
  // to the function to be used to compute results.
  FunctionCache(U (*f)(T)): cache_(new map<T, U>()), func_(f) { }

  // destructor
  ~FunctionCache() {
    delete cache_;
  }

  // return func_(val), but use cache_ to store and retrieve
  // previously computed values instead of always calling func_
  U apply(T val) { ... }

private:
  map<T, U> *cache_;     // cache of <argument, result> pairs
  U (*func_)(T);         // the function
};
```

(Continued on the next page. You may remove this page for reference.)

Question 2. (cont.) Here is an example of how a client program might use the FunctionCache template:

```
// sample function - return x/2.0
double half(int i) { return i/2.0; }

int main(int argc, char **argv) {
  FunctionCache<int, double> hcache(half);
  double x = hcache.apply(3);  // computes and saves <3,1.5>
  double y = hcache.apply(4);  // computes and saves <4,2.0>
  double z = hcache.apply(3);  // returns 1.5 from the cache
  ...                          //   without recomputing
  return 0;
}
```

(a) (16 points) Provide the full definition of function `apply` for the `FunctionCache` template. Write your solution below. Hint: the solution may be quite short – do not be alarmed.

```
  // return func_(val), but use cache_ to retrieve previously
  // computed values, and store new values there

  U apply(T val) {

    if (cache_->count(val) == 1) {
      return (*cache_)[val];
    } else {
      U res = func_(val);
      (*cache_)[val] = res;
      return res;
    }

  }
```

(continued on next page)

Question 2. (cont.) (b) (6 points) The original version of the `FunctionCache` template uses explicit memory management (`delete`) in the destructor to deallocate the `cache_` map data when an instance of `FunctionCache` is destroyed.

Another way to handle the heap data would be to use smart pointers instead of an explicit `delete` to ensure that the `cache_` data is deallocated when a `FunctionCache` object is destroyed.

Below, give a precise description of the changes that need to be made to the original template to make this change. If any code needs to be added or altered, write the new code here and be sure it is clear where the changes should be made and what, if anything, in the original code should be deleted or replaced. If any changes are needed in your implementation of `apply` in part (a), also describe them here.

There are four changes that are needed:

• Add `#include <memory>` at the top of the file.

• Change the instance variable declaration of `cache_` at the bottom of the template from `map<T, U> *cache_;` to `unique_ptr<map<T, U>> cache_;`.

• In the constructor initializer list, change `cache_(new map<T, U>())` to `cache_(unique_ptr<map<T, U>>(new map<T, U>()))`. Or, delete `cache_` from the initializer list and initialize it inside the constructor using the same `unique_ptr` expression in an assignment statement.

• Remove the line `delete cache_;` from the destructor.

Question 3. (16 points) C++ classes. Consider the following program, which compiles and links successfully. (What happens after that is something we'll get to later. ☺)

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

class Base {
public:
  Base() {
    cout << "Base constructor" << endl;
    ia_ = new int[5];
  }
  virtual ~Base() {
    cout << "Base destructor" << endl;
    delete[] ia_;
  }
  Base& operator=(const Base& rhs) {
    if (this != &rhs) {
      delete[] ia_;
      ia_ = rhs.ia_;
      cout << "Base assignment" <<  endl;
    }
    return *this;
  }
private:
  int *ia_;
};

class Derive : public Base {
public:
  Derive() {
    ja_ = new int[5];
    cout << "Derive constructor" << endl;
  }
  virtual ~Derive() {
    cout << "Derive destructor" << endl;
    delete[] ja_;
  }
private:
  int *ja_;
};

int main() {
  Base b1;
  Derive d1;
  b1 = d1;
  return 0;
}
```

(Question continued on next page – you may remove this page if you wish.)

Question 3. (cont.) (a) (8 points) What does this program print when it is executed?

(Reminder/hint: when an object of a derived class is constructed, the base class constructor for that object executes before the derived class constructor. When the object is deleted, the destructors run in the reverse order – derived class destructor first.)

```
Base constructor
Base constructor
Derive constructor
Base assignment
Derive destructor
Base destructor
Base destructor
```

(b) (8 points) Unfortunately, after the program finishes printing the output you described in your answer to part (a), it crashes and does not exit normally. The memory management software detects some sort of problem. What's wrong and what is the error in the code? (Be specific and concise. You do not need to fix the problem – just explain it precisely.)

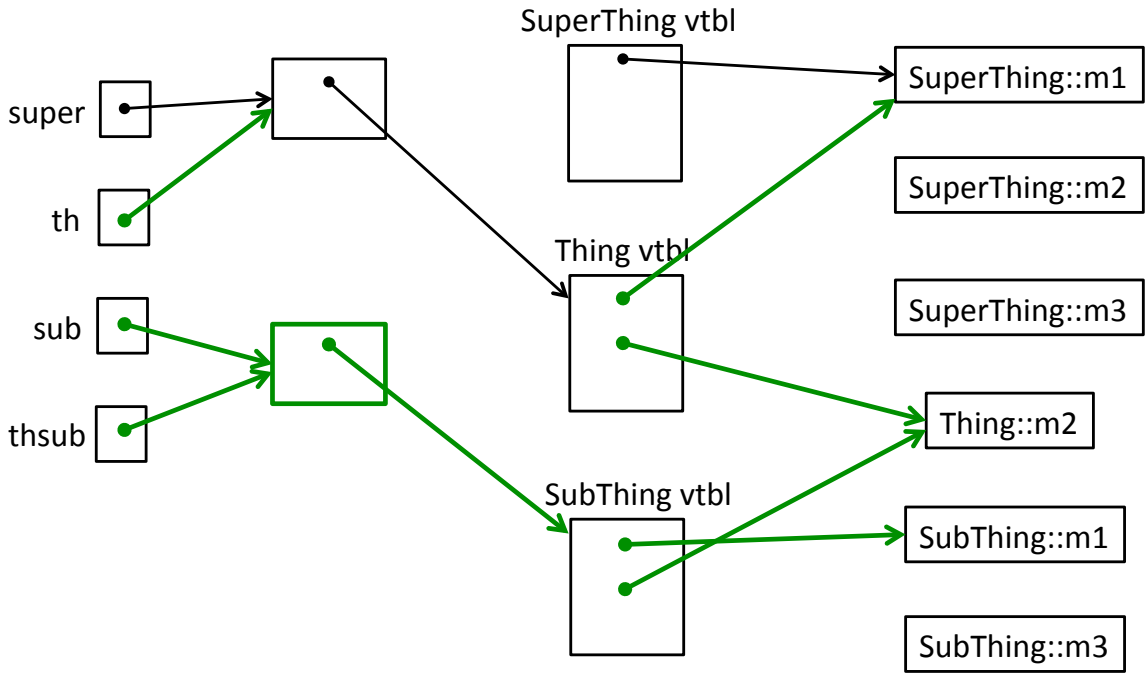
The memory manager reports a "double delete" error when the destructor for the second object is executed. The error is in the `Base::operator=` code. This assignment operator copies a pointer instead of creating a copy of the array. As a result, after the assignment `b1=d1`, both objects point to the single `ia_` array originally allocated to `d1`, and the destructors for `b1` and `d1` both attempt to delete it, causing the double delete error.

Question 4. (20 points) The always entertaining virtual function question. The following program compiles, runs, and produces output with no error messages or other problems. Answer questions about it on the next page.

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

class SuperThing {
public:
  virtual void m1() { m2(); cout << "super::m1" << endl; }
          void m2() {       cout << "super::m2" << endl; }
          void m3() {       cout << "super::m3" << endl; }
};
class Thing: public SuperThing {
public:
  virtual void m2() { m1(); cout << "thing::m2" << endl; }
};
class SubThing: public Thing {
public:
  virtual void m1() {       cout << "sub::m1" << endl; }
          void m3() { m2(); cout << "sub::m3" << endl; }
};

int main() {
  SuperThing *super = new Thing();
  Thing      *th    = (Thing*)super;
  SubThing   *sub   = new SubThing();
  Thing      *thsub = sub;
  ///// HERE /////
  cout << "---" << endl;
  th->m1();
  th->m3();
  cout << "---" << endl;
  sub->m1();
  sub->m3();
  cout << "---" << endl;
  thsub->m1();
  thsub->m3();

  return 0;
}
```

(Question continued on next page – you may remove this page if you wish.)

Question 4. (cont.) (a) (8 points) Complete the following diagram to show the runtime state of the program when execution reaches the comment `///// HERE /////` in function `main`. The diagram should include the variables in `main` (already supplied), the objects they point to, pointers from objects to their vtables, and pointers from vtables to the correct functions. To save time, boxes for the variables in main, the vtables, the functions, and the first object created by the program, have been provided for you. A couple of the arrows representing some of the pointers are also included to get you started. You need to supply all additional objects and pointers needed (if any). Be sure that the order of pointers in the various vtables is clear.



(b) (12 points) What does this program print when it is executed?

```
---
super::m2
super::m1
super::m3
---
sub::m1
sub::m1
thing::m2
sub::m3
---
sub::m1
super::m3
```

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: **getaddrinfo** Purpose: **Get ip address and port on which to listen**

2. Function: **socket** Purpose: **Create a socket**

3. Function: **bind** Purpose: **Bind socket created in step 2 to address/port from step 1**

4. Function: **listen** Purpose: **Identify socket as listening socket to which clients can connect**

5. Function: **accept** Purpose: **Accept client connection and get new socket fd that can be used to communicate with client**

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

7. Function: **close** Purpose: **Shut down client socket and free resources**

Question 7. (20 points) Threads. Consider the following simple C++ program, which prints a sequence of even numbers followed by a sequence of squares. It also contains an extra `#include` and a lock variable that might (☺) be useful later.

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

using namespace std;

static pthread_mutex_t lock;

void print(string what, int num) {
  cout << what << " " << num << endl;
}

// print first n even numbers: 2, 4, 6, ..., 2*n
// you may not modify this funciton
void print_evens(int n) {
  for (int i = 1; i <= n; i++) {
    print("evens", 2*i);
  }
}

// print first n squares: 1, 4, 9, 16, ... n*n
// you may not modify this function
void print_squares(int n) {
  for (int i = 1; i <= n; i++) {
    print("squares", i*i);
  }
}

int main(int argc, char** argv) {
  int nsquares = 4;
  int nevens = 5;

  print_evens(nevens);
  print_squares(nsquares);
  return 0;
}
```

Remove this page from the exam, then continue with the question on the next page. **Do not write anything on this page.** It will not be scanned for grading.

Question 7. (cont.) For this question we would like to modify this program so it executes the two functions `print_evens` and `print_squares` concurrently in separate threads. You may not modify the existing `print_evens` and `print_squares` code. You will need to add appropriate thread starter functions that accept parameters from `main` and call the existing functions with the appropriate arguments. You will also need to make whatever modifications are needed in function `print` so that each output line appears on a separate line by itself without output from the other thread interfering. The existing `print` and `main` functions are copied below and on the next page for you to modify (don't modify the code on the previous page). Make whatever changes and additions are needed to implement correct, concurrent C++ threaded code using pthreads.

```
// modify this function so it is thread safe
void print(string what, int num) {
  pthread_mutex_lock(&lock);
  cout << what << " " << num << endl;
  pthread_mutex_unlock(&lock);
}
// add additional thread starter function definitions here
void *thread_worker_evens(void *arg) {
  int n = *(int*)arg;
  print_evens(n);
  return NULL;
}
void *thread_worker_squares(void *arg) {
  int n = *(int*)arg;
  print_squares(n);
  return NULL;
}
```

(continue with `main` function on the next page)

Question 7. (cont.) Modify the main function below to replace the sequential calls to `print_evens` and `print_squares` with code that executes these two functions concurrently in independent threads and performs whatever other initialization, synchronization, and termination is needed for the concurrent program to work correctly.

```
int main(int argc, char** argv) {
  int nsquares = 4;
  int nevens = 5;


//  print_evens(nevens);
//  print_squares(nsquares);

  pthread_mutex_init(&lock, NULL);
  pthread_t th1, th2;


  pthread_create(&th1, NULL, &thread_worker_evens,
                                        (void*)&nevens);
  pthread_create(&th2, NULL, &thread_worker_squares,
                                        (void*)&nsquares);


  pthread_join(th1, NULL);
  pthread_join(th2, NULL);


  pthread_mutex_destroy(&lock);

  return 0;

}
```

Question 4. (20 points, 4 each) Smart pointers. Below we have several small programs, each of which calls a function `foo`, and each of which uses smart pointers. Some of them are buggy, and some of them work correctly. Your job is to determine, for each program, the following:

(1) Does it compile?
(2) Is its behavior correct (i.e. no memory leaks, run-time errors, or undefined behavior)? Choose n/a if it did not compile.
(3) If you answered "no" to either of the above, explain concisely what the problem is and how to fix it.

You should assume that none of the `foo` functions will attempt to free, delete, or otherwise modify their argument. You can assume that the actual value returned by `main` is not relevant.

(a)
```
int foo(int *n);  // defined elsewhere

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course);
  return ans;
}
```

Does it compile? (circle) yes **(no)** not sure

Does it execute correctly (circle) yes no **(n/a (didn't compile)**

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

`foo`'s parameter type is `int*` but the argument type is `unique_ptr<int>`. A fix is to use `best_course.get()` as the parameter.

(b)
```
int foo(std::shared_ptr<int> p);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(std::shared_ptr<int>(best_course.release()));
  return ans;
}
```

Does it compile? (circle) **(yes)** no not sure

Does it execute correctly (circle) **(yes)** no n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

(continued next page)

Question 4. (cont.)

(c)
```
int foo(int, std::shared_ptr<int> p);

int main() {
  std::shared_ptr<int> best_course(new int[10]);
  int ans = foo(10, best_course);
  return ans;
}
```

Does it compile? (circle) (yes) no not sure

Does it execute correctly (circle) yes (no) n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

The parameter type for the `shared_ptrs`, `<int>`, is not correct for an array. The `shared_ptrs` will do an ordinary `delete` on the array, but it must be a `delete[]`. The fix is to use `shared_ptr<int[]>` for the `shared_ptr` types.

(d)
```
int foo(std::unique_ptr<int> n);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course);
  return ans;
}
```

Does it compile? (circle) yes (no) not sure

Does it execute correctly (circle) yes no (n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

Function `foo` has a call-by-value parameter, and `unique_ptrs` cannot be copied. The solution is to use `shared_ptrs` or to use a reference parameter (&) for `foo`.

(e)
```
int foo(const int *p);

int main() {
  std::unique_ptr<int> best_course(new int(333));
  int ans = foo(best_course.release());
  return ans;
}
```

Does it compile? (circle) (yes) no not sure

Does it execute correctly (circle) yes (no) n/a (didn't compile)

What is the problem (if any) and how do we fix it? (leave blank if not applicable)

There is a memory leak. `best_course.release()` returns the raw pointer and the smart pointer is then no longer responsible for deleting the data. Either we need to explicitly delete the data, or use `get()` instead of `release()` to make a copy of the pointer while still letting the shared pointer delete the heap data later.