midterm2

Friday, April 4, 2025 3:54 PM

midterm2

Midterm 2 Sample

All code should be written in C++. Unless otherwise specified, you may (and I generally will):

- assume that the user of any code you write will be cooperative with the input they supply;
- omit `std::` and the return value of `main()`;
- assume that all necessary libraries have been `#include`d;
- omit `main()` entirely for problems that ask ONLY for function definitions;
- not concern yourself with having optimal solutions (within reason);
- not worry yourself about prompt messages for user input (I sometimes give descriptive prompts to clarify problems);
- not recopy code I have provided, or which you have written in other parts of problems.

Partial credit *will* be given, so do your best if you encounter a difficult question. PLEASE BOX YOUR ANSWER if it is not otherwise clear!

1. You run a company that gives private boat tours. Your boat has 6 seats which can be booked, which are conveniently numbered 0, 1, 2, 3, 4, and 5.

   You want to create a class called `Trip`. Each object is meant to represent an individual excursion out in the boat. Each `Trip` object should have the following **private member variables**:

   - `string time`, a string containing data and time information (like `"March 1, 2025, 11:00 PM"`);
   - `string occ[6]`, which contains the *names of the occupants of the six seats* – if a seat is unbooked, the entry should just contain `"---"`.

   The class should also support the following **methods**:

   - a constructor which receives one string, which sets `time`, and initializes `occ` as described above (initially, all seats should be unoccupied).
   - `bool book(int s, string name)`: this function should update `occ` so that entry `s` is equal to `name`, if seat `s` is unoccupied; the function should return `true` in this case. If seat `s` IS occupied, the function should simply return `false`. (You do not have to check the validity of the value `s`.)
   - `int num_seats()`: this function should return the number of seats currently reserved.

   a. Write the declaration for the class `Trip`, and the definitions for all methods. Mark methods as *const* as appropriate.

   b. Write code in `main()` which declares a `Trip` object named `mytour` for `"July 20 1969"` and then books a passenger named `Neil` in seat `0`.

   c. Add two more functions (perhaps member, perhaps friend) in the class declaration, along with their corresponding definitions, as follows:

   - a method so that if I had two `Trip` objects named `x` and `y`, the code `x == y` should evaluate to `true` if the `time` attributes were exactly the same;
   - a method so that if I had two `Trip` objects named `x` and `y`, the code `cout << x << y` would print out the `time` attributes for each of the objects.

```
class Trip{
private:
      string time;
      string occ[6];
public:
      Trip(string);
      bool book(int,string);
      int num_seats() const;
      bool operator==(const Trip&) const;
      friend ostream& operator<<(ostream &, const Trip&);
};
Trip::Trip(string s): time{s} {
    for(int i=0; i<6; ++i){
        occ[i]="---";
    }
}
bool Trip::book (int seat, string n){
     if (occ[i]=="---"){
         return false;
     }
     occ[i]=n;
     return true;
}
```

```
int  Trip::num_seats() const{
     int ct=0;
     for(int i=0; i<6; ++i){
         if (occ[i] != "---"){
            ++ct;
         }
     }
     return ct;
}

bool  Trip::operator==(const Trip &rhs){
      return  time==rhs.time;
}

ostream& operator<<(ostream &os, const Trip &t){
     os << t.time;
     return os;
}

int  main(){
      Trip  t("July 20 1969");
      t.book(0, "Neil");
}
```

2. Suppose that you are really interested in stamps. You have already written a class called `Stamp` to represent individual stamps.

Now, consider a class called `Collection`, which is meant to hold a collection of stamps.

```
class Collection {
private:
   Stamp* collect_arr;
   int entries;
   void reserve_one();
public:
   Collection(): collect_arr{new Stamp[10]}, entries{0} {}
   void append(Stamp);
   ~Collection();
};
```

Each `Collection` is initialized with enough space to hold 10 `Stamps`, but of course that may grow over time. The member variable `entries` is the number of `Stamps` that have been stored in the collection so far – this should start as `0`.

The function `reserve_one()` grows the length of `*collect_arr` by 1: it should allocate new memory, copy over the contents of old memory, and deallocate the old memory, in the usual way.

The function `append()` should store one more `Stamp` in the collection, by simply writing the argument as the next unused entry in `*collect_arr`, reserving more space if and only if necessary, and of course updating all attributes appropriately. (Remember that the array gets initialized with space for 10 `Stamps`, so that in the beginning, reserving more space will be unnecessary!)

And the destructor simply deallocates the dynamically allocated array.

a. Write the definitions of `reserve_one()`, `append()` and `~Collection()`.

b. What other functions should almost certainly be added to this class? Write the **DECLARATIONS** of these functions, but **DON'T IMPLEMENT THEM!** (There is more than one acceptable way to declare these functions.)

a.
```
void Collection::reserve_one(){
    Stamp* temp = new stamp [entries+1];
    for (int i=0; i<entries; ++i){
        temp[i] = collect_arr[i];
    }
    delete[] collect_arr;      ← Slightly odd: we won't update num-entries
    collect_arr = temp;              here....
}

void Collection::append (Stamp s){
    if (num_entries >= 10){
        reserve_one();
    }
    collect_arr[num_entries] = s;
    ++ num_entries;            ← ....because we update here
}
Collection::~Collection(){
    delete [] collect_arr;
}
```

b. Rule of 3: should also have
```
Collection (const Collection&);                // copy constructor
Collection& operator= (const Collection&);     // copy assignment
```

3. a. Ten people are playing a game of Scattergories. The category is "Animals." Each player has to list as many examples of animals as they can think of.

Write code which allows the user to store all players' answers in one jagged 2D array named **all_entries**. Specifically, for each of the ten players, there should be a request to input how many animals the player can think of. Then, a "row" in the array should be created with the given length, and the user should be allowed to enter and store the names of that many animals.

You should write your code in such a way that, after it was completed, if I were to hypothetically write

`cout << all_entries[7][1];`

then the *eighth player's second animal* would be printed out. **For full credit: allocate EXACTLY as much memory as you will need, and only use language features we've discussed in class** (i.e. do NOT use `vectors`).

b. Now suppose that the game has a second round. To avoid memory leaks, you should deallocate all the memory you've dynamically allocated in part a. Write code that accomplishes that.

```
int main() {
    String **all_entries = new string *[10];
    for(int player=0; Player <10; ++ player) {
        int num;
        cout << "How many?"
        Cin >> num;
        all_entries [player] = new string [num];
        for(int i=0; i< num; ++i){
            Cout <<"Animal?";
            Cin >> all_entries[i];
        }
    }

        ⋮

    For (int player = 0; player<10; ++player){
        delete[] all_entries[player];
    }
    delete[] all_entries;
```

OR:
String *all_entries[10];

Last line
not necessary
if you used this

4. Consider the following code:

```
class AHHH {
private:
    string attr;
public:
    AHHH(): attr{"Dog"} {cout << "Dog\n";}
    AHHH(string n): attr{"Sheep"} {cout << "Sheep\n";}
    AHHH(AHHH &rhs) {
        cout << "Cat\n";
        attr = rhs.attr;
    }
    void operator=(AHHH &rhs) {
        // What's even going on in this function?
        cout << "Ant\n";
        AHHH why_am_i_even_here = rhs;
    }
    ~AHHH(){
        cout << "Bye " << attr << endl;
    }
};

int main(){
    AHHH x, y("Hi");
    y = x;
}
```

Copy constructor

Default constructor

String constructor

operator=

a. What prints out when this code is run?

b. If the ampersand were removed from `void operator=(AHHH &rhs)`, what additional lines will appear? You don't have to specify where exactly they appear in the output.

a. Dog ← x constructed, x.attr = "Dog"

Sheep ← y constructed, y.attr = "Sheep"

Ant ← Assignment called, doesn't do anything (?!?) to x or y,

Cat ← but why_am_i_even_here is copy constructed
why_am_i_even_here.attr = x.attr = "Dog".

Bye Dog ← Then why_am_i_even_here is destroyed

Bye Sheep
Bye Dog ← Then x and y are destroyed
(Technically, y should be destroyed first b/c it was constructed later, but I wouldn't grade on that)

b. Without &, rhs in operator= is COPY CONSTRUCTED from x, with rhs.attr = "Dog".
Copy constructor prints $\boxed{\text{"Cat"}}$
And, when rhs is destroyed, prints $\boxed{\text{"Bye Dog"}}$