Final Practice

TA: Brian Choi

1. Assume the following variable declarations:

```
int foo = 0;
int *ptr = &foo;
```

Which of the following statements will change the value of `foo` to 1?

(a) `ptr++;` (b) `foo++;` (c) `(*foo)++;` (d) `(*ptr)++;`
(e) (a) and (b) only (f) (a) and (d) only **(g) (b) and (d) only** (e) (c) and (d) only

2. What is the output of the following code segment if the input value is the last digit of your student ID?

```
int array[10] = {4, 6, 2, 3, -1, -3, 2, 2, -7, -9};
int index;
cin >> index;  // Enter a digit here.

int *p = array + index;
for (int i = 0; i < 5; i++)
{
  int hops = *p;
  p += hops;
}

cout << *p << endl;
```

input	output
0	**6**
1	**3**
2	**6**
3	**-9**
4	**2**
5	**-7**
6	**4**
7	**2**
8	**-1**
9	**-7**

4. Your high school friend who didn't make it to UCLA and settled for a university called U$C sent you the following message through Facebook:

> Hey, I wrote the following function, `countMatches`, which is supposed to compare two C-strings and count the number of matching characters. Two characters match if they are the same and if they appear in the same position in each string. For example,
>
> `countMatches("UCLA", "U$C", count);` should set `count` to 1,
> `countMatches("Baseball", "ballpark", count);` should set `count` to 2, etc.
>
> I'm not supposed to create any local variable or square brackets, so I ended up submitting the following code:

```
void countMatches(const char *str1, const char *str2, int &count)
{
    count = 0;

    while (*str1 != '\0' && *str2 != '\0')
    {
        if (*str1 == *str2)
            count++;
        str1++;
        str2++;
    }
}
```

> But it doesn't work! Darn pointers! I asked my friends here at U$C and none of them know how to solve this. Can you tell me what I did wrong, like you always did in high school?

You are too busy preparing for your finals, so you decided to copy and paste his message and just mark the corrections. What is your message going to be? (Make the corrections above).

5. Design the class `Goldfish`, which models a creature that is intelligent enough to remember `capacity` characters at a time.

```
class Goldfish
{
      public:
            Goldfish(int capacity);

            void remember(char c);
            void forget();          // Clears m_memory using dots('.')
            void printMemory() const; // Prints the content of m_memory
      private:
            char m_memory[10]; // memory
            int m_amount;       // # of chars remembered.
            int m_capacity;     // # of chars this fish can remember.
};
```

(a) Define the constructor. Initialize `m_memory` with dot(`'.'`)s. If capacity is not positive, then set it to 3 by default. Remember to set `m_capacity`. If capacity is greater than 10, set it to 10.

```
Goldfish::Goldfish(int capacity)
{
    if (capacity < 1)
        m_capacity = 3;
    else if (capacity > 10)
        m_capacity = 10;
    else
        m_capacity = capacity;
    m_amount = 0;                       // Initialize the amount of memory used.
    forget();                           // Fill in dots.
}
```

(b) Implement remember. Store the character `c` into `m_memory`. If you already have `m_capacity` characters memorized, then discard the oldest character in `m_memory` to open up a free slot. (This is an example of LRU (Least Recently Used) replacement.)

```
void Goldfish::remember(char c)
{
    if (m_amount == m_capacity)
    {
        for (int i = 0; i < m_capacity - 1; i++)
            m_memory[i] = m_memory[i + 1];
        m_amount--;
    }
    m_memory[m_amount] = c;
    m_amount++;
}
```


(c) Implement `forget`. Clear memory by filling dot(`'.'`)s into `memory`.

```
void Goldfish::forget()
{
    for (int i = 0; i < m_capacity; i++)
        m_memory[i] = '.';
    m_amount = 0;
}
```

To clarify, here is how a `Goldfish` object can be used:

```
int main()
{
     Goldfish nemo(3);
     nemo.remember('a');
     nemo.printMemory();      // prints "a.."
     nemo.remember('b');
     nemo.remember('c');
     nemo.printMemory();      // prints "abc"
     nemo.remember('d');
     nemo.printMemory();      // prints "bcd"
     nemo.forget();
     nemo.printMemory();      // prints "..."
     return 0;
}
```


6. We'll define `Aquarium` class, where `Goldfish` can live.

```
const int MAX_FISH = 20;

class Aquarium

{
    public:
        Aquarium();
        bool addFish(int capacity);
        Goldfish *getFish(int n);
        void oracle();
        ~Aquarium();
     private:
        Goldfish *m_fish[MAX_FISH];  // Pointers to fish.
        int m_nFish;                 // Number of Fish.
};
```

(a) Define the constructor. Initially, there is no fish in the `Aquarium`.

```
Aquarium::Aquarium()
{
    m_nFish = 0;
}
```

OR

```
Aquarium::Aquarium()
: m_nFish(0)
{}
```

(b) Implement `addFish`, which (dynamically) adds a new `Goldfish` into the `Aquarium`, with the specified memory capacity. If the fish cannot be added because the `Aquarium` already has `MAX_FISH`-many fish residing, return false and don't add any. Otherwise, return true.

```
bool Aquarium::addFish(int capacity)
{
    if (m_nFish >= MAX_FISH)
        return false;

    m_fish[m_nFish] = new Goldfish(capacity);
    m_nFish++;
    return true;
}
```


(c) Implement `getFish`, which returns the pointer to the `Goldfish` at position n. Return nullptr if there are fewer than `n` fish in the `Aquarium`, or if `n` is an invalid position.

```
Goldfish* Aquarium::getFish(int n)
{
    if (n < 0 || n >= m_nFish)
        return nullptr;

    return m_fish[n];
}
```

(d) Implement the destructor. Remove all dynamically allocated objects.

```
Aquarium::~Aquarium()
{
    for (int i = 0; i < m_nFish; i++)
        delete m_fish[i];
}
```

Make sure you understand why you can't do "**`delete[] m_fish;`**".

(e) Implement `oracle`. It prints the memory of ALL `Goldfish` in the `Aquarium`. You can use the member function `printMemory` of `Goldfish` class for this purpose. Once everything is printed, reset (i.e. clear) all Goldfish's memories using `forget` function.

```
void Aquarium::oracle()
{
     for (int i = 0; i < m_nFish; i++)
     {
          m_fish[i]->printMemory();
          m_fish[i]->forget();
     }
}
```

Why not **`m_fish[i].printMemory()`**, or **`m_fish[i].forget()`**? That is, why would using dots(**`.`**), instead of arrows (**`->`**), be wrong?

7. Design `BankAccount` class, which lets the account owner to deposit and withdraw money using a password. The class you write should meet the following requirements:

1) There should be no default constructor.

2) There should be only one constructor, which takes in the initial balance in dollars, and the password. The password is an integer, most likely 4 digits, but we won't enforce that limit here (which means it can even be negative). The initial amount, however, must not be negative, if it is negative, set it to $0.00 by default. Your `BankAccount` class should be able to keep track of the balance up to the last cent.

3) It should support two operations, `deposit` and `withdraw`. Both must be Boolean functions. One must specify the amount of money he wants to deposit/withdraw, and the password. If the password is incorrect, the return value must be `false`. Also, for `withdraw`, if the requested amount exceeds the current balance or is negative, return `false` and do not change anything. If the deposit amount is negative, do not change the balance and return `false`.

4) Add a Boolean function `setPassword`, which takes in two passwords – the original password and the new password. If the original password does not match, return `false` and do not change the password. If the original password is correct, update the password with the new password.

5) Provide an accessor function `balance`, which accepts the password. If the password is correct, return the balance. If not, return -1.

6) You can create private member functions and variables as you wish.

A possible usage of this class looks like this:

```
BankAccount ba(500, 1234);       // initial amount: $500, password: 1234
cout << "$" << ba.balance(1234) << endl;    // prints $500.00
cout << "$" << ba.balance(2345) << endl;    // prints $-1
ba.deposit(25, 1234);                       // deposit $25
cout << "$" << ba.balance(1234) << endl;    // prints $525.00
ba.withdraw(500, 2345);                     // should return false
ba.withdraw(1000, 1234);                    // should return false
ba.withdraw(100, 1234);                     // make the withdraw
cout << "$" << ba.balance(1234) << endl;    // prints $425.00
ba.setPassword(1234, 2345);                 // change the password
ba.withdraw(300, 2345);                     // make the withdraw
```


Solution to (a)~(f). Your solution did not have to include the helper function `checkPwd`.

```
class BankAccount
{
    public:
        BankAccount(double initAmt, int pwd);
        double balance(int pwd) const;
        bool deposit(double amt, int pwd);
        bool withdraw(double amt, int pwd);
        bool setPassword(int oldPwd, int newPwd);
    private:
        double m_balance;
        int m_password;
        bool checkPwd(int pwd) const;  // Helper function.
};


BankAccount::BankAccount(double initAmt, int pwd)
: m_password(pwd)
{
    if (initAmt > 0)
        m_balance = initAmt;
    else
        m_balance = 0;
}

double BankAccount::balance(int pwd) const
{
    if (checkPwd(pwd))
        return m_balance;

    return -1;
}

bool BankAccount::deposit(double amt, int pwd)
{
    if (checkPwd(pwd) && amt >= 0)
    {
        m_balance += amt;
        return true;
    }

    return false;
}
```

```
bool BankAccount::withdraw(double amt, int pwd)
{
    if (checkPwd(pwd) && amt >= 0 && amt <= m_balance)
    {
        m_balance -= amt;
        return true;
    }

    return false;
}

bool BankAccount::setPassword(int oldPwd, int newPwd)
{
    if (checkPwd(oldPwd))
    {
        m_password = newPwd;
        return true;
    }

    return false;
}

bool BankAccount::checkPwd(int pwd) const
{
    return m_password == pwd;
}
```

(g) Write a (global) function called `getNewAccount` which takes in a password as the only parameter and returns a pointer to a dynamically created `BankAccount` instance with $10 in it.

```
BankAccount* getNewAccount(int pwd)
{
    return new BankAccount(10, pwd);     // $10 bonus award!
}
```

8. I've got a brilliant start-up idea here -- we never use our coins, and they just keep piling up on our desks. How about we build a machine that accepts all these coins, and gives you bills that amount to the same dollar value? Let me call this machine "`CoinMoon`". Don't you think it'll be a successful business?

```
class CoinMoon
{
    public:
       CoinMoon();
       void addDollar();       // Add a dollar coin to the machine.
       void addQuarter();      // Add a quarter to the machine.
       void addDime();         // Add a dime to the machine.
       void addNickel();       // Add a nickel to the machine.
       void addPenny();        // Add a penny to the machine.
       void giveMeBills();     // Print what the user will get.
                               // See the example below.

    private:
        // TODO: Add private variables here.
        int totalInCents;


};
```

Here's one way of using it.

```
CoinMoon cm;
cm.addDollar();    // Currently $1.00.
cm.addQuarter();   // Currently $1.25.
for (int i = 0; i < 10; i++)
  cm.addDime();      // Add 10 dimes, such that the total becomes $2.25.
for (int i = 0; i < 32; i++)
  cm.addPenny();     // Add 32 pennies, such that the total becomes $2.57.
cm.giveMeBills();  // Produces the following output.
```

Output:

```
2 dollar bill(s)
2 quarter(s)
0 dime(s)
1 nickel(s)
2 penny(ies)
```

Note that the total number of coins that are returned by the machine must be minimal. Also, once `giveMeBills()` is called, the machine returns to the original state and starts anew.

Define all member functions of `CoinMoon`, such that the machine works as intended. Do not add or change the public members. You are free to add private members as needed.

```
CoinMoon::CoinMoon()
: totalInCents(0)
{}

void CoinMoon::addDollar()
{
  totalInCents += 100;
}

void CoinMoon::addQuarter()
{
  totalInCents += 25;
}

void CoinMoon::addDime()
{
  totalInCents += 10;
}

void CoinMoon::addNickel()
{
  totalInCents += 5;
}
```

```
void CoinMoon::giveMeBills()
{
  int dollars = totalInCents / 100;
  totalInCents %= 100;
  int quarters = totalInCents / 25;
  totalInCents %= 25;
  int dimes = totalInCents / 10;
  totalInCents %= 10;
  int nickels = totalInCents / 5;
  totalInCents %= 5;
  int pennies = totalInCents;

  cout << dollars << " dollar bill(s)"
       << endl;
  cout << quarters << " quarter(s)"
       << endl;
  cout << dimes << " dime(s)" << endl;
  cout << nickels << " nickel(s)"
       << endl;
  cout << pennies << " pennie(s)"
       << endl;

  totalInCents = 0;
}
```

... what? Someone has capitalized on this idea already?

9. Implement the class `Hotel`, as declared below, which keeps track of the reservation status of each room in a hotel. Each room can be `RESERVED`, `OCCUPIED`, or `EMPTY`, and this information is stored in a 2-dimensional array, where each row represents a floor, and each column represents a room. Each room is represented by an integer (e.g., `425`). For example, the status of room 425 is stored in `m_rooms[4][25]`.

```
const char RESERVED = 'R';
const char OCCUPIED = 'O';
const char EMPTY = 'E';

const int FLOORS = 20;
const int ROOMSPERFLOOR = 50;

class Hotel
{
    public:
        Hotel();
        bool reserve(int roomNum);
        bool cancel(int roomNum);
        bool checkIn(int roomNum);
        bool checkOut(int roomNum);
        int numEmpty(int floor) const;
    private:
        char m_rooms[FLOORS][ROOMSPERFLOOR];
        // More private members here, if necessary.
        bool validNum(int roomNum);
        bool changeState(int roomNum, char from, char to);
};

Hotel::Hotel()
{
   // EMPTY the rooms.
   for (int i = 0; i < FLOORS; i++)
        for (int j = 0; j < ROOMSPERFLOOR; j++)
            m_rooms[i][j] = EMPTY;
}
```

Implement other functions below.

```
bool Hotel::reserve(int roomNum)
{
    return changeState(roomNum, EMPTY, RESERVED);
}

bool Hotel::cancel(int roomNum)
{
    return changeState(roomNum, RESERVED, EMPTY);
}
```



```
bool Hotel::checkIn(int roomNum)
{
    return changeState(roomNum, RESERVED, OCCUPIED);
}

bool Hotel::checkOut(int roomNum)
{
    return changeState(roomNum, OCCUPIED, EMPTY);
}

int Hotel::numEmpty(int floor)
{
    if (floor < 0 || floor >= FLOORS)
        return -1;

    int count = 0;
    for (int i = 0; i < ROOMSPERFLOOR; i++)
    {
        if (m_rooms[floor][i] == EMPTY)
            count++;
    }
    return count;
}

// Write helper functions down here if necessary.

// Check if the room number is valid.
bool Hotel::validNum(int roomNum)
{
    int floor = roomNum / 100;
    int room = roomNum % 100;

    return (floor >= 0 && floor < FLOORS) && (room >= 0 && room < ROOMSPERFLOOR);
}

// Change the reservation state from "from" to "to", only if the room is in the
// specified "from" state.
bool Hotel::changeState(int roomNum, char from, char to)
{
    // Change the room specified by roomNum from one state to another.
    if (validNum(roomNum) && m_rooms[roomNum / 100][roomNum % 100] == from)
    {
        m_rooms[roomNum / 100][roomNum % 100] = to;
        return true;
    }
    return false;
}
```

