Final

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/class 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. Evan is creating his own AI called ChatEMF. He has created two functions, `char dumb_char()` and `char smart_char()`, that use artificial intelligence to return random characters; when these functions are called repeatedly, they produce sequences of characters that looks vaguely like intelligent English text. The former tends to give weird output; the latter gives much more impressive output, but will only be available to premium customers. *You can assume these functions are written, and call them in your solution.*

   Now, Evan just needs some help creating classes for Accounts on his website.

   The basic type of Account gives its users a limit of 1000 characters of output each day; each character of output is gotten by calling the `dumb_char()` function.

   On the other hand, the PremiumAccount has a limit of 6000 characters per day: the first 5000 characters are produced by the advanced `smart_char()` function, while the last 1000 characters are produced by `dumb_char()`.

   a. Write a class declaration for `Account`s, and implement all member functions. Each `Account` object should have the following **protected member variables**:

   - `string name`, representing the name of the account holder;
   - `int max_chars`, representing the number of characters of output that are allowed each day – this should be initialized to 1000;
   - `int chars_used`, representing the total number of characters that have been output today – this should be initialized to 0.

   The class should also support the following **public member functions**:

   - A constructor which receives a `string` which sets the name as an argument, and sets the other attributes as above;
   - A member function called `void new_day()`, which should receive no arguments and return nothing, but should simply reset `chars_used` to 0;
   - A member function `void output(int n)`, which should either print out `n` random characters (making calls to the aforementioned `dumb_char()` function), or as many characters as can be printed before hitting the daily character limit given by `max_chars`. Of course, `chars_used` should be updated appropriately when characters are printed.

   b. Now, write the declaration and implementation of a publicly-inherited class called `PremiumAccount`. Each `PremiumAccount` should have, in addition, the following protected member:

   - `int max_premium`, representing the number of characters of PREMIUM output that are allowed each day – this should be initialized to 5000.

   The class should also support the following **public member functions**:

   - A constructor which receives one string parameter for the name, and sets all the attribute variables (keeping in mind that premium customers get 5000 premium "smart" characters and 1000 more non-premium "dumb" characters each day);
   - and another version of `void output(int n)` which hides the one from the base class. This one prints out `n` characters like before, except there are a total of 6000 characters allowed per day: the first 5000 characters will call the `smart_char()` function, and the last 1000 will call the `dumb_char()` function.

2. a. Write a function called `int runtrue(vector<bool> x, int len)`. This function should find the earliest appearance of `len` consecutive `true`s in `x`, and then return the index of the last entry of this run (or -1, if there are not `len` consecutive entries which are all `true`). For example, if `x` contained

   {`false, true, true, false, true, true, true, false`}

   then `runtrue(x, 3)` would return 6, because the first time three `true`s appear in a row is in entries 4 through 6.

   b. I have two vectors of `char`s of the same length, named `responses` and `correct_answers`, which respectively represent the answers that a student has given on a multiple choice test, and the correct answers on the test, which might look something like:

```
vector<char> response =       {'A', 'D', 'E', 'C', 'A'};
vector<char> correct_answers = {'A', 'B', 'E', 'C', 'A'};
```

   In this example the student got 4 out of 5 correct.

   Write code which *locates the first time on the test where the student gets 7 in a row correct.* Your program can report the answer by giving the index (zero-based) of the last question that he gets correct. If the student never gets seven in a row correct, the program should print `No 7-in-a-row`.

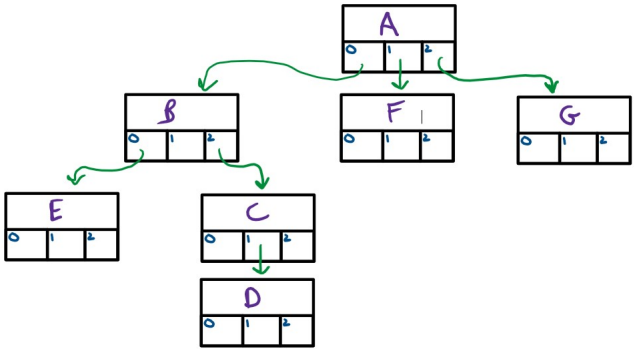
   As part of your answer, you'll probably want to create another vector; and, **for full credit, you MUST use the function from part a.**

3. For this problem, recall that if `x` is a `string` variable, then `x.substr(i)` returns the substring of `x` from index `i` until the end. For example, if `x = "abcdefgh"`, then `x.substr(2)` would return `"cdefgh"`.

   Consider the following struct:

```
struct Node3 {
   string data;
   Node3 *zero, *one, *two;
   Node3(string s): data{s}, zero{nullptr}, one{nullptr}, two{nullptr} {}
};
```

   We use this struct to create *ternary trees*: these are trees where each Node has three children, labeled "zero", "one" and "two". For example:



   Write the body of a function

   `void insert3(string entry, string pos, Node3* &root)`

   This function should take a ternary tree which is pointed to by `root`, and add a new entry to the tree, by using the string `pos`. This string will be composed of `0`'s, `1`'s and `2`'s, which describe the path from the `root` node to where the new node should be added. For example, the following code would produce the tree shown above:

```
Node3 *root = nullptr;
insert3("A", "", root);
insert3("B", "0", root); // B should be the 0-child of A
insert3("C", "02", root); // C should be the 2-child of B
insert3("D", "021", root); // D should be the 1-child of C
insert3("E", "00", root); // E should be the 0-child of B
insert3("F", "1", root); // F should be the 1-child of A
insert3("G", "2", root); // G should be the 2-child of A
```

   Your function only needs to work when `pos` corresponds to the child of an existing node, where there is no currently existing node.

4. You manage a bus company. The bus company has 20 buses Each bus holds 35 passengers.

   Each `BusFleet` object should have the following **private member variables**:

   - `string* buses[20]`. This will hold pointers to arrays which hold the passengers on 20 bus routes.
   - `int caps[20]`. This will hold the capacity of each bus. Each entry should be initialized to 35.

   The class should also support the following **public member functions**:

   - A default constructor which creates 20 heap-allocated arrays of 35 `string`s, with pointers to them stored in `buses`; and which sets each entry of `caps` to be 35.
   - A destructor which releases all the heap-allocated memory.
   - `void add_passenger(int n, int seat, string name)`, which should go the array for the bus whose index is `n`, and set the entry with index `seat` to be set to `name`. (You don't need to do any index checking.)
   - `void add10(int n)`, which adds 10 seats to the bus with index `n`. More precisely, it should allocate an array with 10 more entries than what currently exists for the `n`th array, copying the contents of the current array into the front of the new array. Don't forget to deallocate the old array and update `caps`.

   Write the declaration for this class, and implement all the methods.

5. Consider the following code.

```
class Top {
protected:
    string label;
public:
    Top(): label{"Blank"} {
        cout << "A: " << label << endl;
    }
    Top(string x): label{x} {
        cout << "B: " << label << endl;
    }
    Top(const Top& rhs): label{rhs.label} {
        cout << "C: " << label << endl;
    }
    Top& operator=(Top& rhs) {
        cout << "D: " << label << endl;
        return rhs;
    }
    virtual void f() {cout << "[Top::f()]" << label << endl;}
};

class Inher: public Top {
private:
    string label2;
public:
    Inher(string x, string y):Top(x), label2{y} {cout << "E: " << label << "|" << label2 << endl;}
    void f() {cout << "[Inher::f()]" << label << label2 << endl;}

};
```

a. What would print out from the following code?

```
int main() {
   Top pppp("Potato");
   cout << "---------------" << endl;
   Top rrrr = pppp;
   cout << "---------------" << endl;
   rrrr.f();
}
```

b. What would print out from the following code?

```
int main() {
   Inher abab("Apple", "Banana");
   abab.f();
   cout << "---------------" << endl;
   Top x = abab;
   x.f();
}
```

c. What would print out from the following code?

```
int main() {
   Top* cdcd = new Inher("Cat", "Dog");
   cdcd->f();
}
```

d. Which answers from the above questions would change if the word `virtual` was removed from the declaration of `f()` in `Top`? (Your answer can just be "part a", "part b" and/or "part c".)