MTH 4300, E. Fink

Final Practice

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 `#included`;
- 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.

1. For a positive integer n, let $\pi(n)$ denote the number of prime integers less than or equal to n.

   Write code that first creates a `vector` containing all prime integers less than 1000. The program should then ask the user to input an integer n less than 1000, and print out $\pi(n)$.

2. Consider the following code, which manually constructs a linked list.

```
struct Node {
    string data;
    Node *next;
    Node(string s, Node* n): data{s}, next{n} {}
};

int main(){
    Node e("Elephant", nullptr), c("Cat", &e), b("Bear", &c), a("Alligator",&b);
    Node *head = &a;

    // ADD CODE HERE
}
```

   a. Write code which alters the linked list pointed to by `head`, so that a fifth element containing the data `"Duck"` is inserted between the third and fourth elements of the original linked list.

   b. Write code which alters the linked list pointed to by `head` so that the data in each element has the letter `s` added to the end (i.e. `Alligators`, `Bears`, etc.). Your code should be written so that even if this list had additional nodes added to it, it would still work as intended.

3. a. Write a function `void reverse(Node* head)`. The parameter to this function should be thought of as a pointer to the first element of a linked list. The function should reverse this linked list: that is, it should change each `next` element of each `Node` in the list, in a such a way that the list now starts with the last `Node` and ends with the old first `Node`. **It should also return a pointer to the first node of the reversed list.**

   For example, with the following code, `Evan Carol Bob Alice` should print out:

```
// Prints linked list in order
void print_list(Node *listptr) { ... }

int main() {
    Node n1("Alice", nullptr); Node n2("Bob", nullptr); Node n3("Carol", nullptr); Node n4("Evan", nullptr);
    Node *head = &n1;
    n1.next = &n2;  n2.next = &n3;  n3.next = &n4;

    reverse(head);
    print_list(head);
}
```

   b. If you answered part a *with* recursion, provide a non-recursive implementation. If you answered part a *without* recursion, provide a recursive implementation. (Hints for the recursive implementation: I don't think it's simpler than the iterative implementation; you should probably have two base cases, for empty lists and one-node lists.)

4. Suppose that we wish to modify the binary search tree we made in class, so that each `TreeNode` contains a `count` attribute: if you try to insert an existing word into the search tree, instead of creating a new entry, it increments `count`. Of course, strings that are not currently in the tree should be inserted as before, so that after the insert, each `TreeNode` still satisfies the property that its `data` is greater than that of all the nodes in the `left` sub-tree, and less than that of all the nodes in the `right` sub-tree.

   Here is the new `TreeNode` class:

```
template <typename T>
struct TreeNode {
public:
    T data;
    int count;
    TreeNode *left, *right;
    TreeNode(T s, TreeNode* l = nullptr, TreeNode *r = nullptr): data{s}, left{l}, right{r}, count{1} {}
};
```

   a. Write a function `template <typename T> void insert(TreeNode<T>* &t, const T &value)` which inserts an element into the tree, as described above (either adding a new `TreeNode` at the appropriate place, or, if there is a `TreeNode` whose data is equal to `value`, incrementing the counter of that node).

   b. Write a function `template <typename T> int count(TreeNode<T>* &t, const T &value)`, which returns the number of times that the string has been inserted, which could be 0!

5. Recall our `Stack` class.

```
template<typename T>
struct Node {
   T data;
   Node *next;
   Node(T s, Node *n = nullptr): data{s}, next{n} {}
};

template<typename T>
class Stack {
private:
    Node<T> *top;
public:
    Stack(): top{nullptr} {}
    void push(const T&);
    void pop();
    bool is_empty() const;
    T peek() const;
    ~Stack();

    void operator=(const Stack&) = delete; // Lazy -- prohibiting copying,
    Stack(const Stack&) = delete;       // but maintains Rule-of-3 compliance
};

template<typename T>
void Stack<T>::push(const T &s) { top = new Node<T>(s, top); }

template<typename T>
void Stack<T>::pop() {
    if (!is_empty()) {
        Node<T> *temp = top->next;
        delete top;
        top = temp;
    }
}

template<typename T>
bool Stack<T>::is_empty() const { return top == nullptr; }

template<typename T>
T Stack<T>::peek() const {
    if (is_empty()) {
        throw std::runtime_error("Can't do that");
    }
    return top->data;
}
template<typename T>
Stack<T>::~Stack() {
    while(!is_empty()) {
        pop();
    }
}
```

I have a list of people named `sandwich_line`, some who have peanut butter, and some who have jelly. The people who have peanut butter have a `P` in front of their names, and the people who have jelly have a `J` in front. For example,

```
vector<string> sandwich_line = {"PAlice", "PBob", "JCarol", "JDennis", "JEvan", "PFrank",
"JGeorge"};
```

Write code, **using a single `for` loop and a `Stack`**, which prints out as many pairs of people who can form a peanut butter and jelly sandwich as possible. For example, with the given sample list, it should print out something like

```
PAlice and JCarol
PBob and JDennis
JEvan and PFrank
```

6. Rewrite the `Stack` class above so that the constructor, `push()`, `is_empty()`, `peek()` and `pop()` functions have the same behavior, but the data is stored in a `vector` rather than in a linked list. The destructor and other rule-of-three should not be necessary anymore; there should be different private attribute(s).

7. Ordinarily, arithmetic is written using *infix notation.* This means that operators are written between the operands they work on. For example, in the expression `3 * 4`, the multiplication symbol is placed between the operands 3 and 4.

   Postfix notation is another way of writing arithmetic expressions, where the operator is placed AFTER the two operands – for example, `3 4 *` evaluates to 12 in this setting. If there is more than one operator, the operators are always evaluated from left to right, and each operator operates on the two numbers immediately to the left. So in `3 4 5 6 + * -`, the addition evaluates first, to 11, leaving behind `3 4 11 * -`; then the multiplication evaluates on 4 and 11, giving 44, and leaving behind `3 44 -`; finally, the minus evaluates, giving `-41`.

   Write a program that opens up a file called `postfix.txt`, that contains a single postfix arithmetic expression, including the operands `+`, `-`, `*`, `/`, and well as numbers that may or may not contain decimals. The program should evaluate this postfix expression. As part of your solution, use a stack: specifically, use a `std::stack<double>` which holds the operands. (See homework 8 for a review of `std::stack`.)

   For this problem, it will also be helpful to have the function `std::stod` (from the `<string>` header), which converts a `string` to a `double`, if the string happens to represent a numeric value.

8. Recall our `TreeNode` struct:

```
template <typename T>
struct TreeNode {
public:
    T data;
    int count;
    TreeNode *left, *right;
    TreeNode(T s, TreeNode* l = nullptr, TreeNode *r = nullptr): data{s}, left{l}, right{r}, count{1} {}
};
```

   Write a function `template <typename T> void reverse(TreeNode<T>* &root)` which receives a pointer to the root of a binary tree, and which recursively reverses the tree: that is, it switches the left and right pointers of all nodes in the tree pointed to by `root`.

9. If I add the numbers 10, 5, 18, 2, 14, 6, 20, 24 and 11 to an empty binary search tree, in that order, what does the tree look like? (Assume that lesser elements are in the LEFT subtree.)

10. In a binary tree T, the *level* of a given value x is the number of downward movements from the root needed to arrive at the value x in the tree, or -1 if x does not appear in the tree. For example, if x is the root of T, the level of x is 0; if x is a child of the root, then the level of x is 1; if x is a child of a level-1 node, then the level of x is 2; etc.

Write a function `template <typename T> int level_BST(TreeNode<T> *root, T val)`. This function should return the level at which `val` is stored in the Tree pointed to by `h`. **It should only work for binary SEARCH trees (constructed with lesser keys stored in the left subtree).**

11. a. Write the definition of a class named `Vehicle`. Each `Vehicle` object should have the following **protected member variables**:

    - `double capacity`, the maximum load that the vehicle can support as cargo.
    - `double cargo_load`, the current cargo load that the vehicle contains.

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

    - a constructor which takes one argument, which sets the `capacity` (and `current_load` should be set to 0).
    - `bool load(double w)`, which adds `w` to the `cargo_load` if that would not raise it above `capacity` and returns `true` in this case; otherwise it should return `false`.
    - `double weight() const`, which returns the `cargo_load`.

    If I run the code

    ```
    Vehicle v1(100);
    v1.load(40);
    v1.load(50);
    v1.load(30);
    cout << v1.weight();
    ```

    it should print out `90`.

    b. Now write a class called `PassengerVehicle` which inherits publicly from `Vehicle`, which supports two more protected attributes: an `int` named `seats`, which is the number of seats available, and an `int` named `seats_taken`, which should be initialized to 0.

    This class should have a reasonably defined constructor. It should also support its own version of `weight()`, which should return the `cargo_load` plus 100 times the number of seats that are filled. Finally, it should also have a `bool seat()` function, which adds 1 to the `seats_taken` if that would not raise it above `seats` and returns `true` in this case; otherwise it should return `false`.

    If I run the code

    ```
    PassengerVehicle pv2(100, 2);
    pv2.seat();
    pv2.seat();
    pv2.load(30);
    cout << pv2.seat() << " " << pv2.weight();
    ```

    it should print out `0 230` (the 0 for `false`).

    c. Imagine if I continue the sample code above with

    ```
    Vehicle v3 = pv2;
    cout << v3.weight();
    ```

    what prints out? Why?

12. Consider the code below.

    ```
    class Base {
    protected:
        int x;
    public:
        string y;
        Base(): x{0}, y{"Base"} {}
        void func() { cout << "[Base::f()] " << x << y << endl; }
        virtual void otherfn() = 0;
    };

    class Deriv_one: public Base {
    public:
    ```

```
    double extra;
    Deriv_one(): Base(), extra{3.14} {}
    void otherfn() { cout << "[Deriv_one::otherfn()] " << extra << endl; }
};

class DerDer: public Deriv_one {
public:
    DerDer(double value) { extra = value; }
    void func() { cout << "[DerDer::funct()] " << endl;}
    void otherfn() { Deriv_one::otherfn(); cout << x << endl; }
};

class Deriv_two: public Base {
public:
    Deriv_two() {}
    void func() { cout << "[Deriv_two::func()] " << x << y << endl; }
};

int main() {
    Base bb;       //
    bb.func();     // Block one
    cout << bb.y;  //

    Deriv_one d1d1; //
    d1d1.func();   // Block two
    d1d1.otherfn(); //
    cout << d1d1.y; //

    DerDer dd(2.718); //
    dd.func();      // Block three
    dd.otherfn();   //
    cout << dd.x;   //

    Deriv_two d2d2; //
    d2d2.func();   // Block four
    d2d2.otherfn(); //
    cout << d2d2.y; //
}
```

a. If just the code labeled `Block one` is run (and the other blocks were commented out), will it compile? If so, what will print? If not, what exactly is the problem?

b. Answer the same questions for `Block two`, `Block three`, and `Block four`.

13. a. What does the following code print?

```
class One {
public:
    void f() {
        g();
        cout << "One::f" << endl;
    }
    void g() {
        cout << "One::g" << endl;
    }
    virtual void h() {
        cout << "One::h" << endl;
    }
};

class Two: public One {
public:
    void g() {
        cout << "Two::g" << endl;
    }
    virtual void h() {
        f();
        cout << "Two::h" << endl;
    }
};

class Three: public Two {
public:
    virtual void h() {
        g();
        cout << "Three::h" << endl;
    }
};

int main() {
    One* a = new Two();
    a->h();
    cout << "----" << endl;
    a->g();
    cout << "----" << endl;
    One* b = new Three();
    b->h();
    cout << "----" << endl;
    b->f();
}
```

b. If the two functions named `g()` are also made `virtual`, what changes?

14. a. (*Warning: you will be updating the answer to part a in later parts.*) Write the definition of a class named `Borrowable`, meant to represent items that can be borrowed from the library (either pieces of media, or equipment). Each `Borrowable` object should have the following **protected member variables**:

- `string id`, the *name of the item*;
- `int quantity`, the *quantity of the item currently held.*

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

- a constructor with two parameters, which set the two attributes;
- `void display() const`, which displays info in a reasonable manner;
- `bool borrow()`, which deducts one from `quantity` and returns `true` if the `quantity` is at least one; otherwise it should return `false`.

b. Write a class called `Media` which inherits from `Borrowable`, which should have two additional protected member variables, `string medium` (book, film, music) and `string genre` (comedy, drama, fiction, jazz); the constructor should set these appropriately. This class should also have an appropriately written `display()` method.

c. (*To make this part of the problem work, you will likely need to go back to part a and make at least two changes!*) Now, write a class called `Library`. Each `Library` object should have the following **protected member variable**:

- `vector<Borrowable*> items`, a list of pointers to items that can be checked out.

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

- no constructor necessary; the one provided by the compiler is fine.
- a method `void insert_equip(string i, int q)`, which should create a new dynamically allocated `Borrowable` object whose `id` will equal `i` and `quantity` will equal `q`, and insert that into the `items` list.
- a method `void insert_media(string i, int q, string m, string g)`, which should create a new dynamically allocated `Media` object whose `id`, `quantity`,`medium` and `genre` will be set by `i`, `q`, `m` and `g` respectively; and insert that into the `items` list.
- a method `bool borrow(string i)`, which acts as follows: if there is an entry in `items` whose `id` is given by `i`, then the `borrow()` function should be called on that item, and the value produced should be returned. If there is no such entry in `items`, then `false` should be returned.
- a method `void display_all() const`, which should go through `items`, and call the appropriate `display()` function for each item, each on its own line.

- and a destructor, which deallocates all dynamically allocated objects. (No other rule-of-three functions are necessary.)

15. When the code below is run, the program will have a memory leak. Explain why, and what could be done to prevent it.

```
class Character {
private:
    string name;
public:
    Character(string n): name{n} {}
    void print() const { cout << name << " says \"Hello!\"" << endl; }
};

class CharWithItems: public Character {
private:
    int num_items;
    string *arrptr;
public:
    CharWithItems(string n, int ni): Character(n), num_items{ni}, arrptr{new string[ni]} {}

    void insert(string val, int idx) {
        if (0 <= idx && idx < num_items) { arrptr[idx] = val; }
    }

    void print() const {
        Character::print();
        cout << "Item list: ";
        for (int i = 0; i < num_items; ++i) { cout << arrptr[i] << " "; }
        cout << endl;
    }
    ~CharWithItems() { delete[] arrptr; }
};

int main() {
    Character *c1 = new Character("Alice");
    Character *c2 = new CharWithItems("Bob", 100);
    delete c1;
    delete c2;
}
```