final

Friday, May 10, 2024 9:10 PM

final

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. You work at New York's hottest night club, and are in charge of developing the club's guestlist software. You make the declaration for a class called `Guestlist`, whose objects represent guestlists for parties. Each `Guestlist` object should have the following **private member variables**:

   - `vector<string> invitees`, the *list of invited guests*, which should start out as empty.
   - `vector<bool> arrived`, which should also start out empty. At all times it should have the same length as `invitees`. Each entry should start out as `false`, but entries will become `true` when the corresponding entry in `invitees` has arrived (see below).

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

   - Even though there's no need for it, write a default constructor which has no parameters and sets nothing.
   - `void invite(string n)`, meant to represent inviting a new guest with name `n`. The function should add `n` to the end of the `invitees` list, and `arrived` should be updated with a new `false` entry.
   - `bool admit(string n)`, which checks if `n` is the name of someone who should be admitted to the party – if *they are in the* `invitees` *list*, and if *the corresponding entry in* `arrived` *indicates that they have not arrived yet* – and returns whether or not they should be. If they should be admitted, the corresponding entry in `arrived` should be updated to `true` to reflect this.

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

   b. Now, write the declaration and definition of a public derived class called `LimitedGuestlist`. Objects of this class should contain two additional attributes: `present`, which represents the number of guests who have arrived, and `max`, which represents the maximum number of occupants who are allowed to be admited.

   The value of `max` should be set by a parameter to the `LimitedGuestlist` constructor, and `present` should start at 0.

   Write an updated definition for `admit()` as well, which checks if `present < max` before proceeding. Any time someone is admitted, the function should also update `present`!

```
class Guestlist {
private:
    vector<string> invitees;
    vector<bool> arrived;
public:
    Guestlist() {}
    void invite(string);
    bool admit(string);
};
void Guestlist::invite(string n) {
    invitees.push_back(n);
    arrived.push_back(false);
}

bool admit(string n) {
    bool should_admit = false;
    for (int i=0; i<invitees.size(); ++i) {
        if (invitees[i] == n) {
            if(!arrived[i]) {
                arrived[i] = true;
```

```
            if(!arrived[i]) {
                arrived[i]=true;
                should_admit=true;
            }
            break;
        }
    }
    return should_admit;
}

class Limited Guestlist: public Guestlist {
Protected:
    int present;
    int max;
Public:
    LimitedGuestlist(int n): Guestlist(), present{0}, max{n} {}
    bool admit(string);
}

bool LimitedGuestlist (string n) {
    if (present < max) {
        bool ad = Guestlist::admit(n);
        if(ad) {
            ++present;
        }
        return ad;
    }
    return false;
}
```

2. a. I have a collection of files which all contain (small) integers separated by spaces, e.g. the contents of the files look like

```
14 19 3 -7 20 51 38 10
```

The names of the files are contained in a `vector<string>` named `fnames`; it might look like

```
int main() {
    vector<string> fnames = {"file1.txt", "file2.txt", "file3.txt"}; // Exact contents may be different.
```

Complete this code so that it prints out *the maximum number contained in each file.* As part of your solution, write a function `filemax()`, which receives the name of a file as an argument, and returns an int which is the maximum value contained in the file with that name. (For example, if the example above was the contents of `file1.txt`, then `filemax("file1.txt")` would return `51`.)

b. The function `filemax()` that you've written should assume that the contents of the file are `int`s and it should return an `int`, but there's no reason that basically the same function couldn't replace the type "`int`" with any other numeric type (e.g. `double`, `float`, `long long`).

Rewrite your function `filemax` using templates so that, when called appropriately, it could be used to return any of the numeric types.

[Doing part b first]

```
template <typename T>
T filemax (string filename){
    ifstream file(filename);
    T entry, max;
    file >> max;
    while (file>>entry){
        if (entry > max) {
            max = entry;
        }
    }
    return max;
}

int main(){
    vector<string> names = ....
    for(auto it=names.begin(); it != names.end(); ++it){
        cout << filemax(*it)<<" ";
    }
}
```

3. Let's create part of a homemade version of the `string` class! We'll call it `MyStr`.

Here is the declaration for the class:

```
class MyStr {
private:
    char *arrptr;
    int size;
public:
    MyStr(char letters[], int k){
        size = k;
        arrptr = new char[k];
        for(int i = 0; i < k; ++i){
            arrptr[i] = letters[i];
        }
    }
    friend ostream & operator<<(ostream &, const MyStr &);
    MyStr(const MyStr&);
    char operator[](int idx);
    ~MyStr();
};
```

a. Provide implementations (outside of the declaration) for the copy constructor (which should make a copy of an existing `MyStr`; the copy should maintain a separate array!), index operator (which allows a user to retrieve a character from a `MyStr` by index), and destructor (which should release all heap-allocated memory).

For the index operator,

```
char x[] = {'a', 'b', 'c', 'd'};
MyStr y(x, 4);
cout << y[2];
```

should print c.

b. Now, add one more line to the declaration class, and one more function, which would allow

```
cout << y;
```

to print abcd.

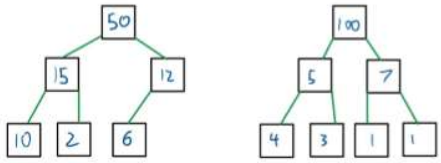
```
MyStr::MyStr(const MyStr &rhs){
    arrptr = new char[rhs.size];
    size = rhs.size;
    for(int i=0; i<size; ++i){
        arrptr[i]=rhs[i];
    }
}
char MyStr::operator[](int idx){
    return arrptr[idx];
}

MyStr::~MyStr(){
    delete[] arrptr;
```

```
MyStr::~MyStr(){
    delete [] arrptr;
}
ostream & operator<<(ostream & os, const MyStr & rhs){
     for(int i=0; i<size; ++i)
        os<< arrptr[i];
     }
     return os;
}
```

4. Consider the familiar class below.

```
struct TreeNode {
public:
    int data;
    TreeNode *left, *right;
    TreeNode(int s): data{s}, left{nullptr}, right{nullptr} {}
};
```

Imagine that we have a binary tree which is NOT necessarily a binary search tree. Such a tree is called *strong* if EVERY node is greater than the sum of all its descendants (children, children's children, etc.); a tree with no nodes is considered strong. For example, in the below, the left tree **is** strong, because 15 is greater than 10 + 2; 12 is greater than 6; and 50 is greater than 15 + 10 + 2 + 12 + 6. But the right tree is **NOT** strong, since 5 is NOT greater than 4 + 3.



Write a function called `bool is_strong(TreeNode*)`, which receives a pointer to the root of a tree as argument, and returns whether or not the pointed-to tree is strong.

You may write and call an additional helper function if you wish; I think there is a clear choice for one that would be very helpful.

```
int sum (TreeNode *root){
    if(root==nullptr){
      return 0;
    }
    return sum(root->left) +root->data +sum(root->right);
}

bool is_strong(TreeNode *root){
    if (root==nullptr){
       return true;
    } else if (root->data >(sum(root)-root->data)){
       return is_strong(root->left) &&
                    is_strong(root->right);
    } else {
       return false;
    }
```

5. Consider the following code.

```
class First {
public:
    string x;
    First(): x{"Default"} {cout << x << " constructed" << endl;}
    First(const First& rhs): x{rhs.x} {cout << "CConstruct" << endl;}
    First& operator=(First& rhs) {
        cout << "CAssign" << endl;
        return rhs;
    }
    void alter(string entry) {x = entry;}
    virtual void print() const {cout << x << endl;}
    void nothing() {}
};

class Second: public First {
private:
   string another;
public:
   Second(string z):First(), another{z} {}
   void alter(string entry) {another = entry;}
   void print() const {cout << x << another << endl;}
};
```

a. What would print from the following? Identify the lines that cause each printed item.

```
int main() {
    First aaa;         // Line 1
    First bbb = aaa;   // Line 2
    aaa.alter("pqrs"); // Line 3
    aaa.print();       // Line 4
}
```

Default constructed
CConstruct
pqrs

b. What would print from the following? Identify the lines that cause each printed item.

```
int main() {
    First *ptr = new Second("hello"); // Line 5
    ptr->print();                     // Line 6
    ptr->alter("goodbye");            // Line 7
    ptr->print();                     // Line 8
}
```

Default constructed
Default hello
goodbye hello

c. Which of lines 9-12 would cause compilation errors, if any? Clearly indicate the line number(s) which would, and for each one, explain briefly what would go wrong.

```
int main() {
    First f;
    Second s("Stringy");
    f = s;             // Line 9
    cout << f.x;       // Line 10
    cout << s.another; // Line 11
    s.nothing();       // Line 12
}
```

Only Line 11, since another is private.