Queens College, CUNY, Department of Computer Science

Object-oriented programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

Project 3, part 5

due date Tuesday August 14, 2018, 11:59 pm

- **NOTE: It is the policy of the Computer Science Department to issue a failing grade to any student who either gives or receives help on any test. A student caught cheating on any question in an exam, project or quiz will fail the entire course.**
- **Students who form teams to collaborate on projects *must inform the lecturer of the names of all team members* ahead of time, else the submissions will be classified as cheating and will receive a failing grade.**
- **Any problem to which you give two or more (different) answers receives the grade of zero automatically.**
- Please submit your solution via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.
- Please submit one zip archive with all your files in it.
  1. The zip archive should have either of the names (CS211 or CS611):

     `StudentId_first_last_CS211_project3_Aug2018.zip`
     `StudentId_first_last_CS611_project3_Aug2018.zip`
  2. The archive should contain one "text file" named "Part5.[txt/docx/pdf]" (if required) and one cpp file per question named "Pt5_Q1.cpp" and "Pt5_Q2.cpp" etc.
  3. Note that not all questions may require a cpp file.
  4. A text file is not always required for every project.
- **In all questions where you are asked to submit programming code, programs which display any of the following behaviors will receive an automatic F:**
  1. Programs which do not compile successfully (non-fatal compiler warnings are excluded).
  2. Array out of bounds, reading of uninitialized variables (including null pointers).
  3. Operations which yield NAN or infinity, e.g. divide by zero, square root of negative number, etc. *Infinite loops.*
  4. Programs which do NOT implement the public interface stated in the question.
- **In addition, note the following:**
  1. All debugging statements (for your personal testing) should be commented out.
  2. Program performance will be graded solely on the public interface stated in the questions.

1 Class `EmailAccount`

- The class `EmailAccount` is the "high level" class which manages the message folders.
- It is a relatively small class.
- Most of the work is done in the folder classes.
- The class declaration is shown below.
- The class contains private data as follows, all of which are obvious.
  1. The name of the owner `Name _owner`.
  2. A drafts folder (pointer) `Drafts * _drafts`.
  3. A polymorphic pointer to the inbox `BaseFolder * _in`.
  4. A polymorphic pointer to the outbox `BaseFolder * _out`.

```
class EmailAccount {
public:
  EmailAccount(string s);
  ~EmailAccount();

  const Name& owner() const { return _owner; }  // accessor
  Drafts& drafts()  { return *_drafts; }        // accessor/mutator
  BaseFolder& in()  { return *_in; }            // accessor/mutator
  BaseFolder& out() { return *_out; }           // accessor/mutator

  void send(Message *m);
  void receive(Message *m);
  void insert(Message *m);

private:
  Name _owner;
  Drafts * _drafts;
  BaseFolder  * _in;
  BaseFolder * _out;
};
```

- **Write the constructor (non-default) as follows.**

  1. Set the owner using the input string `_owner.set(s)`.
  2. Dynamically allocate the draft folder `_drafts = new ...`.
  3. Dynamically allocate the inbox folder `_in = new ...`.
  4. Dynamically allocate the outbox folder `_out = new ...`.
  5. **Register the account with the ISP** (see below).

```
EmailAccount(string s) {
  _owner.set(s);
  _drafts = new            // invoke constructor correctly
  _in  = new               // invoke constructor correctly
  _out = new               // invoke constructor correctly
  ISP::addAccount(this);   // register the account with the ISP
}
```

- **Write the destructor.** Release all the dynamic memory.

- **Write functions to (i) make a deep copy, else (ii) disable copies by making the relevant functions private.** It is your choice.

- Next we write the class methods. They are straightforward.

  1. **The `receive` method puts the incoming message in the inbox.**
     The method has only one line.

```
void receive(Message *m) { // invoke "receive" method of inbox, with input "m" }
```

  2. **The `insert` method puts the message into the drafts.**
     The method has only one line.

```
void insert(Message *m) { // invoke "addDraft" method of drafts, with input "m" }
```

  3. **The `send` method does two things:**
     (i) put the message into the outbox,
     (ii) send the message to the ISP.

```
void send(Message *m) {
  _out->receive(m);
  ISP::send(m);
}
```

- *That is the complete class definition!*

2 Class ISP

- The declaration of the class `ISP` (internet service provider) is given below.
- All the class data and methods are static.
- The data is a map, whose key is a `Name` object and value is a `EmailAccount` pointer.
- The constructor is private. Objects of the `ISP` class cannot be instantiated.

```
class ISP {
public:
  static void addAccount(EmailAccount *e) {
    _accounts[e->owner()] = e;
  }

  static void send(Message *m);

private:
  ISP() {}
  static map<Name, EmailAccount*> _accounts;
};
```

- **Write the `send` method as follows.**

  1. Get a pointer to the relevant account in the map `ac = _accounts[m->to()]`.
     This is because the message is sent "to" someone, i.e. `m->to()`.
  2. If `ac` is NULL, print an error message and exit the function.

     ```
     cout << "Delivery failed, to recipient: " << m->to().name() << endl;
     return;
     ```

  3. Else if `ac` is not NULL, do the following.
     (a) **Instantiate a dynamically allocated copy of the message m.**
        i. *This requires the Message class to have a public copy constructor.*
        ii. It is your responsibility to make sure the copy is made correctly.
     (b) Invoke the `receive` method of the target email account `ac`.

     ```
     Message *clone = (dynamically allocated copy of m)
     ac->receive(clone);
     return;
     ```