Queens College, CUNY, Department of Computer Science

Object-oriented programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

Project 3, part 4

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 "Part4.[txt/docx/pdf]" (if required) and one cpp file per question named "Pt4_Q1.cpp" and "Pt4_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 Drafts folder

- The Drafts folder is different from the others and is not part of the polymorphic set.
- The class declaration is shown below.
- The class contains **private** data as follows.
    1. Integer `int _newKey`. This will be used to generate keys to index draft messages.
    2. A **map** of draft messages `map<int, Message*> _drafts`.
    3. **You must include the header file <map>.**
    4. A pointer `EmailAccount *_ac` to the email account which owns the folder.
    5. All folder objects have a parent `EmailAccount` pointer.

```
class Drafts {
public:
  Drafts(EmailAccount *ac);
  ~Drafts();

  void display() const;
  void send(int n);
  void erase(int n);

  Message* addDraft();
  Message* addDraft(Message *m);
  Message* getDraft(int n) { return _drafts[n]; }
  Message* operator[](int n) { return _drafts[n]; }

private:
  int newKey();

// data
  int _newKey;
  map<int, Message*> _drafts;
  EmailAccount *_ac;
};
```

- The key of the map is an integer.
    1. Since a new message is blank (by definition), there is no way to index it using "from" or "to" or any other data field in a message.
    2. Hence we index the messages in the map using integers 1,2,...
    3. To get a pointer to a message we invoke `getDraft(int n)` or `operator[](int n)`.
    4. Both `getDraft` and `operator[]` return NULL if the message is not found in the map.

- **Write the constructor (non-default) as follows.**
  1. Set `_ac = ac`. Use memberwise initialization if you wish (optinal).
  2. Set `_newKey = 0`. Use memberwise initialization if you wish (optinal).
  3. Set `_drafts[0] = NULL`.
  4. Similar to the Inbox and Outbox folders, we insert a NULL message, with a key value of zero.
  5. This is so that the real draft messages are numbered 1,2,... which looks prettier than 0,1,2,...
- **Write the destructor.**
  1. Recall that the map contains pointers to messages.
  2. **Traverse the map using an iterator** and deallocate the memory for each message (operator `delete`) to realease 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.
- **Write the private method `newKey()` as follows.**
  1. Return `++_newKey`. *That's it.* The logic is as follows.
  2. Every time we add a message into the map, we increment the value of `_newKey` and use it to index the message.
  3. This guarantees the added message is assigned a key which does not interfere with the existing drafts.
  4. As stated above, the constructor initializes `_newKey` to zero.
  5. Of course, this policy means that when messages are sent (or erased), then "holes" will develop in the list of keys in the drafts folder.
  6. This is a weak feature of the software design, but let us not waste time being too clever.
- **Explain why `_newKey` cannot be declared as static class data.**
- **Write the method `erase(int n)` as follows.**

```
void Drafts::erase(int n) {
  delete _drafts[n];
  _drafts.erase(n);
}
```

  1. If the message is not found in the map, then `_drafts[n]` is NULL so calling `delete` is safe.
  2. The `map` class has an `erase` method, which erases the entry from the map.

- **Write the method `addDraft()` as follows.**

  1. Get a key `int key = newKey()` for the message. This should be obvious.
  2. Dynamically allocate `ptr = new Message`. Use the constructor `Message(_ac->owner().name())`.
  3. Add it to the map `_drafts[key] = ptr`.
  4. Return `ptr`. After all, the user wants to edit the new message.

- **Write the method `addDraft(Message *m)` as follows.**

  1. This is when the user invokes "reply" or "forward" and a message is put into the drafts folder.
  2. Again, get a key `int key = newKey()` for the message.
  3. The message already exists. Nothing to allocate.
  4. Add m to the map `_drafts[key] = m`.
  5. Return `m`. After all, the user wants to edit the message.

- **Write the method `display()` as follows.**

1. Print the account owner's name and test if there are any messages to display.

```
cout << _ac->owner().name() << " drafts: " << endl;
if (_drafts.size() <= 1) {
  cout << "no messages to display" << endl;
}
```

2. If there are messages to display, run a loop over the messages.
3. **Use a iterator of suitable type to traverse the map.**

```
(iterator of suitable type) mit;
```

4. Because the message with `key == 0` is a dummy NULL message, we skip it.

```
if (mit->first == 0) continue;
```

5. For each valid message (i.e. `key > 0`), print the following:
   (a) Print the value of the key `mit->first`.
   (b) Declare a temporary variable `const Message *ptr =` ("value" of map element).
   (c) Print the "to" name `ptr->to().name()`. *This is a draft, so it may be a blank string.*
   (d) Print the subject `ptr->subject()`. *This is a draft, so it may be a blank string.*
6. Hence the overall display is as follows.

```
for (iterator ...) {
  if (mit->first == 0) continue;
  // ptr = ...
  // print  i,   ptr->to().name(),   ptr->subject()
}
```

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

  1. Get a pointer `ptr = _drafts[n]`.
  2. If `ptr == NULL` print “Message not found” and exit the function
  3. Else if the address of “to” is blank, print a different error message and exit the function. A message cannot be sent if there is no recipient.

     ```
     if (ptr->to().address() == "") {
       cout << "You must specify a recipient" << endl;
       return;
     }
     ```

  4. Else do the following.

     ```
       ptr->send();
       _drafts.erase(n);
       _ac->send( ptr );
     ```

  5. The explanation is as follows.
     (a) The “send” method of the message sets the date of the message object. (See the `Message` class.)
     (b) We erase the message from the drafts folder.
     (c) The “send” method of the account pointer `_ac` is invoked.
     (d) The account object will transfer the message to the outbox.
     (e) The account object will contact the ISP, to send the message to the recipient’s account.