Queens College, CUNY, Department of Computer Science

Object-oriented programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

Project 2

due date Sunday July 29, 2018, 12:00 noon

- **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_project2_July2018.zip`
     `StudentId_first_last_CS611_project2_July2018.zip`
  2. The archive should contain one "text file" named "Project2.[txt/docx/pdf]" (if required) and one cpp file per question named "Q1.cpp" and "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.

General information

- **Ignore the statements below in this section if they are not relevant for the questions in this document.**
- **You are permitted to copy and use code supplied in the online lecture notes.**
- **64–bit computers**

  1. Hardware architecture will not be taken into account in the questions below.
  2. Solutions involving the writing of programs will not be judged if they work (or were written) on a 64–bit instead of a 32–bit computer.

- **The following refers to questions involving mathematical calculations.**

  1. **Value of π to machine precision on any computer.**
  2. Some compilers support the constant `M_PI` for π, in which case you can write

     ```
     const double pi = M_PI;
     ```

  3. If your compiler does not support `M_PI`, the value of π can be computed via

     ```
     const double pi = 4.0*atan2(1.0,1.0);
     ```

Material to be used in later questions

- **Ignore the statements below in this section if they are not relevant for the questions in this document.**
- **This is material for use in other questions. Nothing to "calculate" here.**
- Form a set of eight digits $(d_1, \ldots, d_8)$ as follows.
- **Take the 8 digits of your student id and define $(d_1, \ldots, d_8)$ as follows:**

$$\begin{aligned} d_1 &= \text{digit 1 of student id}\,, \\ d_2 &= \text{digit 2 of student id}\,, \\ d_3 &= \text{digit 3 of student id}\,, \\ d_4 &= \text{digit 4 of student id}\,, \\ d_5 &= \text{digit 5 of student id}\,, \\ d_6 &= \text{digit 6 of student id}\,, \\ d_7 &= \text{digit 7 of student id}\,, \\ d_8 &= \text{digit 8 of student id}\,. \end{aligned} \tag{0.1}$$

- For example if your student id is 23054611, then

$$\begin{aligned} d_1 &= 2\,, \\ d_2 &= 3\,, \\ d_3 &= 0\,, \\ d_4 &= 5\,, \\ d_5 &= 4\,, \\ d_6 &= 6\,, \\ d_7 &= 1\,, \\ d_8 &= 1\,. \end{aligned} \tag{0.2}$$

- *For some student ids, it is possible that some of the digits may be zero. It is also possible that some of the digits may be equal.* Do not worry.
- For the student id 11111111, all the digits are equal.
- For the student id 33330000, four digits are zero and the other four are all equal to 3.

1 Class `Message1`

1.1 Class declaration

- This is the first step towards implementing an "Email project" for a simple model of an "Email account" with a list of email messages.
- The first step is to create a class to contain an email message.
- Even to do that requires formalism which we have not studied yet.
- Hence here we shall write a simple class "`Message1`" which is the first step towards a full "email message" class.
- We shall build on the `Message1` class in later assignments to write `Message2`, etc., until at the end we write the final "`Message`" class.
- The class has the following private data members:

```
class Message1 {
public:
  // to do

private:
  string _from;
  string _to;
  string _subject;
  string _text;
  int _date;
};
```

- We shall write public and private methods for the class below.
- In real life we can send an email message to multiple recipients (including mail groups), but we shall keep the design simple and have only one recipient.
- For now the "date" is an `int`. We shall replace it by a date object later in later versions of the class.
- We shall follow a common industry practice and attach an underscore "_" for the names of class data members.

1.2 Accessor methods

- The simplest to write are the accessor methods.
- **Write function bodies for the following accessor methods.**

```
string from()    // etc
string to()      // etc
string subject() // etc
string text()    // etc
```

1. They are all public.
2. **They are all `const`.**
3. They all have return type `string`.
4. It should be obvious what data value each accessor returns.

- Also write the following accessor method for the date. We shall improve on it later.

```
string date() const
{
  return string("n/a");
}
```

1.3 `const` method `print()`

- **Next write a method `print()` to print the message.**

  1. The method is public.
  2. The method is `const`.
  3. The return type is `void`.

- Print the output in the following format. *You should copy and paste the code below.*

```
void print() const {
  cout << "From: "    << _from    << endl;
  cout << "To:   "    << _to      << endl;
  cout << "Subject: " << _subject << endl;
  cout << "Date: "    << date()   << endl;
  cout << _text << endl;
  cout << endl;
}
```

- **Important:**
  Explain why the `print()` method is allowed to call the `date()` method.

1.4 Non-const methods: mutators

- Next write some mutator methods.
  1. The methods below are all public.
  2. They all have return type `void`.
  3. They are not `const` because they change the values of the class data members.
- **setSubject(...).**
  It should be obvious what this method does.

```
void setSubject(string s) // etc
```

- **setText(...).**
  It should be obvious what this method does.

```
void setText(string txt) // etc
```

- In practice, users rarely replace the entire text.
  1. They save a draft message and *edit* the existing text.
  2. In many cases they append extra text to the end of the existing text.
  3. We shall write two "helper" methods to prepend and append text.
- If users wish to fix typos in the text or insert new text in the middle, etc., they must call `setText`.
- **prependText(...).**
  1. Insert the input string at the beginning of the existing string `_text`.
  2. But also insert a blank space to separate it from `_text`, to avoid a mess.
  3. **Adapt the function `prependText` you wrote in Midterm 1 to make it a class method.**

```
void prependText(string t) // etc
```

- **appendText(...).**
  1. Insert the input string at the end of the existing string `_text`.
  2. But also insert a blank space to separate it from `_text`, to avoid a mess.
  3. **Adapt the function `appendText` you wrote in Midterm 1 to make it a class method.**

```
void appendText(string t) // etc
```

1.5 Mutator: `setRecipient(...)`

- We can update and change the name of the person to whom we wish to send the message.
- This is the "`_to`" data member in the class.
- **Write a public non-const method `setRecipient(...)` as follows.**

```
void setRecipient(string t) {
  _to = addDomain(t);
}
```

- The reason for the strange code is as follows.
  1. An email address has the form `name@domain`.
  2. When users call `setRecipient`, for simplicity let us say they input only the name, e.g. "Bob" or "John.Smith" etc.
  3. We internally append the domain.
  4. **We restrict the domain to `qc.cuny.edu`.**
  5. If a user inputs that a message is to "Bob" then we internally set:

     ```
     _to = Bob@qc.cuny.edu;
     ```

  6. Hence the "`addDomain(...)`" methods appends the domain to the email address.
- Therefore we do not assign `_to = t` instead we set `_to = t + domain`.
- The next step is therefore to write the method `addDomain(...)`.

1.6 Private method: `addDomain(...)`

- **Write a private non-const method `addDomain(...)` as follows.**

```
string addDomain(string s) {
  const string _domain = "@qc.cuny.edu";
  istringstream iss(s);
  iss >> s;                        // remove leading and trailing blank spaces
  // add "domain" to end of "s" and return concatenated string
}
```

- The `istringstream` removes leading and trailing blank spaces from the input string s.

```
"Bob"      ---> Bob@qc.cuny.edu
"Bob "     ---> Bob@qc.cuny.edu        // remove trailing blanks
" Bob "    ---> Bob@qc.cuny.edu        // remove leading and trailing blanks
```

- The simplest way to understand is for you to experiment and try, when the class is written.

1.7 Constructors

- **Write a default constructor and four non-default constructors.**
- **Default constructor.** Set the date to zero (the strings initialize to blank strings).

```
Message1() { _date = 0; }
```

- **Non-default constructor #1.** Input the sender only.
-
```
Message1(string f) {
  _from = addDomain(f);     // set sender ("from") using addDomain
  _date = 0;
}
```

- **Non-default constructor #2.** Input sender and recipient only.

```
Message1(string f, string t) {
  _from = ...      // see non-default constructor #1 how to use addDomain
  _to = ...
  _date = 0;
}
```

- **Non-default constructor #3.** Input sender, recipient and subject.

```
Message1(string f, string t, string s) {
  _subject = s;
  // etc
}
```

- **Non-default constructor #4.** Input sender, recipient, subject and text.

```
Message1(string f, string t, string s, string txt) {
  _text = txt;
  // etc
}
```

1.8 Does not exist: `setSender(...)`

- **Do not write a method `setSender(...)`.**
- **The sender of a message is set in the constructor.**
- **The sender cannot be edited.**
- We can edit the recipient of an email message, but not the sender.
- *What about the default constructor? The sender is blank. How will we change that?*
- **The default constructor is not useless.**
  1. **We require the default constructor so we can instantiate arrays of objects.**
  2. Later we can employ the assignment operator to copy data into the array elements (including the value of the sender).

1.9 Big Three: copy, assign, destroy

- The Big Three copy constructor, assignment operator and destructor **need not be written.**
- None of the class data members are pointers.
- A shallow copy is adequate.

1.10 Class declaration

- The declaration of the class `Message1` looks like the following.
- The `_date` data member will be changed to something else (a class) later.

```
class Message1 {
public:
  Message1();
  Message1(string f);
  Message1(string f, string t);
  Message1(string f, string t, string s);
  Message1(string f, string t, string s, string txt);

  string from()    const;
  string to()      const;
  string subject() const;
  string text()    const;
  string date()    const;

  void setRecipient(string t);

  void setSubject(string s);
  void setText(string txt);
  void prependText(string t);
  void appendText(string t);

  void print() const;

private:
  string addDomain(string s);

  string _from;
  string _to;
  string _subject;
  string _text;
  int _date;
};
```

1.11 Overload `operator≪`

- **Overload `operator<<` with the following signature.**

  ```
  ostream& operator<< (ostream& os, const Message1 &m);
  ```

- It does the same thing as the class method `print()` but sends the output to the ostream `os` instead of `cout`.

2 Class Vec_Message1

2.1 Introduction

- **Write a class Vec_Message1 with the following declaration.**

```
class Vec_Message1 {
public:
  Vec_Message1();
  Vec_Message1(int n);
  Vec_Message1(int n, const Message1 &a);

  Vec_Message1(const Vec_Message1 &orig);
  Vec_Message1& operator= (const Vec_Message1 &rhs);
  ~Vec_Message1();

  int capacity() const;
  int size() const;

  Message1 front() const;
  Message1 back() const;

  void clear();
  void pop_back();
  void push_back(const Message1 &a);

  Message1& at(int n);

private:
  void allocate();
  void release();

  int _capacity;
  int _size;
  Message1 * _vec;
};
```

- ***Don't panic.***
- **Examine the code of the class Vec_int carefully.**
- The code for the class Vec_Message1 is **almost a copy and paste** of Vec_int.
- **All we need to do is to replace "int" by "Message1" in a few places.**

2.2 `allocate()` and `release()`

- Examine the code of the method `allocate()` in the class `Vec_int` carefully.
- **Edit allocate() to allocate memory of the correct data type.**
- **Explain why no changes are required in the function body of `release()`.**

2.3 Constructors, copy, assign, destroy

- **Explain why no changes are required in the function bodies for any of the constructors, copy, assignment and destructor.**
- We encapsulated all the code changes in `allocate()` and `release()` (actually only `allocate()`).
- **Understand that we are not just learning C++ programming.**
- We are also learning software design.

2.4 `clear()` and `pop_back()`

- **Explain why no changes are required in the function bodies for `clear()` and `pop_back()`.**

2.5 `front()` and `back()`

- The code in the class `Vec_int` for `front()` and `back()` returns an array element (if `_size > 0`) and returns 0 otherwise.

```
int Vec_int :: front() const {
    if (_size > 0) ...
    else return 0;
}
int Vec_int :: back() const {
    if (_size > 0) ...
    else return 0;
}
```

- The items highlighted in **red** must be changed edited.
- The return type must be changed from `int` to `Message1`. This is obvious.
- The statement "`return 0`" is tricky.
  1. We cannot write "`return 0`" for an object of type `Message1`.
  2. What "zero object" shall we return?
  3. Once again, the answer is given by the default constructor.
  4. We instantiate a default object and return that.

```
Message1 Vec_Message1::front() const {
  if (_size > 0) ...
  else {
    Message1 default_obj;
    return default_obj;
  }
}
Message1 Vec_Message1::back() const {
  if (_size > 0) ...
  else {
    Message1 default_obj;
    return default_obj;
  }
}
```

2.6 Default objects (for your information, not a question)

- A **default object** of a user-defined class is an object which is created with **no user inputs.**
- A default object is the analog of an uninitialized variable for a primitive data type such as `int` or `double`, etc.
- The default constructor instantiates a default object.
    1. By definition, the object has no user inputs.
    2. Of course, the default constructor can set values for all the class data members.
    3. It can set all the data values to zero or NULL (or blank strings, etc.)
    4. *But the essential fact is that the calling application does not supply any inputs.*
    5. A default object is essentially an "empty" object.

2.7 `push_back(...)`

- The code in the class `Vec_int` for `push_back(int a)` contains two lines which must be edited.

```
void Vec_int :: push_back(int a) {
        ...
        int *oldvec = vec;
        ...
}
```

- **Edit the code to work correctly for the class `Message1`.**
- Next, when declaring a temporary pointer to save the address of the old array during memory reallocation, the pointer type must be changed from `int*` to `Message1*`.
- Both of the above changes should be obvious.
- *That's it.*

2.8 `at(...)`

- The code in the class `Vec_int` for `at(int n)` returns a reference to an array element if the value of n is valid.
- Else it returns the dereference of a NULL pointer. *(Weird, but it works.)*
- **Edit the code for `at(...)` to return a pointer of the appropriate type.**

2.9 operator []

- **Declare two operators as public methods of `Vec_Message1`.**

```
Message1& operator[] (int n);
const Message1& operator[] (int n) const;
```

1. They are both public.
2. The reason we require two versions of `operator[]` is so that `const` objects can invoke the second version.
3. That is why the return type of the second version is a **const reference.**

- **Edit the code to work correctly for the class `Message1`.**

2.10 Functions `reverse(...)` and `print(...)`

- Write the functions `reverse(...)` and `print(...)` to operate on an object of type `Vec_Message1`.

```
void reverse(Vec\_Message1 &v);                    // input is referemce
void print(ostream &os, const Vec_Message1 &v);    // input is const reference
```

- The function `print(...)` should run a loop and call `operator<<` for $v[i]$, for $i = 0, \ldots,$ and print the output to `os`.

```
// loop i = 0, etc
os <<  // etc
```

- **Explain why the input to "`print(...)`" is `const` but "print(...)" itself is not `const`.**

2.11 Main program

- **Write a main program to test your code.**

```
Message1 a, b;
// etc
a = a = b;
a = b = a;
Message1 c(Message1(Message1(a)));  // oh yes
```