Queens College, CUNY, Department of Computer Science

Object-oriented programming in C++

CSCI 211 / 611

Summer 2019

Instructor: Dr. Sateesh Mane

Midterm Exam (Makeup) take home for grade boost

due Saturday July 27, 2019 by 9:00 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.**
- This is a **take home** test, for a possible grade boost.
  1. **This take home exam is for a grade boost only. It does NOT substitute for your performance in the in-class exam.**
  2. **Any grade boost applied is entirely at the discretion of the grader.**
  3. **If for any reason it is decided that a grade boost is not warranted for your submitted solution, no grade boost will be awarded.**
  4. **The amount of the grade boost is also entirely at the discretion of the grader.**
- **No grade boost will be awarded if the instructions below are not strictly followed.**
  1. Write your answers in C++ source files named Q2.cpp, Q3.cpp and Q4.cpp. Include all required headers and the statement "`using namespace std;`" at the top of each file. Write the answers to text questions as comments in the file. Put the C++ source files in a zip archive as explained below.
  2. Please submit your solution via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`. The file attachment is a zip archive with one of the following naming formats:

     ```
     StudentId_first_last_CS211_takehome_Summer2019.zip
     StudentId_first_last_CS611_takehome_Summer2019.zip
     ```
  3. **Important: Do not encrypt the zip. Other formats such as RAR archive or OneDrive or Google Drive, etc. will not be accepted.**

- **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, e.g. use of deprecated features).
    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.

1 Question 1 NOT GRADED

NOT CONSIDERED FOR BOOST

2 Question 2

- In this question, we make a simplified version of the `vector<string>` class, call it `svec`.
- The class declaration is as follows.

```
class svec {
private:
  string *ds;              // dynamic array of strings
  int len;                 // length of array
  int num;                 // number of populated array items
public:
  //      *** WRITE CLASS CONSTRUCTORS ***

  ... size();
  ... front();
  ... back();
  ... get(int i);   // ADDRESS of i^th array element (NULL if value of i not valid)

  void push_back(... s);
  cleanup();               // release dynamic memory
};
```

- The data members are `ds` = dynamically allocated array of strings, `len` = length of array and `num` = number of array elements that are actually populated by the calling application. We shall call these the "valid elements" below. Hence we must have `num <= len`.
- Extra credit will be awarded if the keyword "`const`" is used wherever applicable.
- **Write three constructors for the class `svec` as follows.**
  1. Default constructor: array is NULL, `num` and `len` both zero.
  2. Non-default constructor: input is `(int n)`. If `n <= 0` do same as default constructor. Else set `len = n` and allocate array dynamically. Set `num = len`.
  3. Non-default constructor: input is `(int n, string s)`. If `n <= 0` do same as default constructor. Else set `len = n` and allocate array dynamically. Copy the input string s into all the array elements. Set `num = len`.
- **Accessor methods: write the function code, with suitable return types.**
  1. Note: `front()` and `back()` return blank strings if `num` is zero.
  2. Also `get(int i)` returns the ADDRESS of the i^{th} array element, else NULL if the value of i is not valid.
- **Write the code for `cleanup()` to deallocate dynamic memory, set the array to NULL and set `len` and `num` to zero.**

- **Write the code for `push_back(... s)` as follows.**

  1. If `len` is zero, set `len = 10`, dynamically allocate the array to the correct length, copy *s* into the first array element and set `num` to 1.
  2. Else if `num < len`, copy *s* into the appropriate array element and increment `num` by 1.
  3. Else if `num == len`, do the following. Increment `len` by 10 and dynamically allocate a new array of strings of the appropriate length. Execute a loop to copy the data from the old array to the new array. Also copy the string *s* into the new array, at the correct location. Increment `num` by 1. Release the memory in the old array. Redirect `ds` to point to the new array.

- **Write the code for the lines indicated in the main program below.**
  (Do not write the whole of `main()`.)

```
int main()
{
  int n;
  string s;
  cout << "Please enter a positive number and a string:" << endl;
  cin >> n >> s;

  //  INSTANTIATE AN SVEC OBJECT "v" using n and s in constructor   // WRITE CODE
  //  PRINT THE SIZE OF v                                           // WRITE CODE

  while (true) {
    cout << "Please enter a string (type ''end'' to break):" << endl;
    cin >> s;
    if (s == "end") break;
    v.push_back(s);
  }

  //  PRINT THE DATA IN THE FIRST AND LAST VALID ELEMENTS OF v      // WRITE CODE
  //  WRITE A LOOP TO PRINT THE DATA IN ALL VALID ELEMENTS OF v     // WRITE CODE
  //  RELEASE ALL DYNAMICALLY ALLOCATED MEMORY                      // WRITE CODE
  return 0;
}
```

- **MANDATORY FOR ALL STUDENTS**

  1. Explain what happens if `v` is instantiated using the default constructor.
  2. Explain what happens if `v` is instantiated using the non-default constructor with only *n* but not *s*.

3 Question 3

- You are given a C++ class "Student" as follows:

```
class Student {
public:
  Student(... n);                  // non-default constructor
  ... getName();                   // return name
  ... getID();                     // return id
  void setID(...);                 // set value of id

private:
  const string name;
  int id;
};
```

- **Write the code for the class `Student` as follows:**

  1. Non-default constructor: initialize `name` to n and `id` to zero.
  2. Methods `getName()` and `getID()`, with the appropriate return type for each method.
  3. Method `setID(...)`, with the appropriate function argument(s).
  4. Extra credit will be awarded if the keyword "`const`" is used wherever applicable.

- The main program below works as follows.

  1. First `main()` prompts the user to input the number of students (assume $n > 0$ below).
  2. **Write code to dynamically allocate `pps` to an array of length n of pointers to `Student`.**
  3. Then `main()` executes a loop to prompt the user to input a name and id on each pass, and calls `add(...)` to add elements into `pps`.
  4. Then `main()` prompts the user to input an integer j, and calls `remove(...)` to remove the element at the j^{th} position in `pps`.
  5. Then `main()` calls two functions `display(...)` and `release(...)`.

- **Write the code for the functions `add(...)`, `remove(...)`, `display(...)` and `release(...)`.**

  1. The function `add(... i)` receives inputs and dynamically creates an object and assigns it to the i^{th} location in `pps`. *Do nothing if the value of i is out of bounds.*
  2. The function `remove(... j)` deallocates the element at the j^{th} location in `pps` and sets that array element to `NULL`. *Do nothing if the value of j is out of bounds.*
  3. The function `display(...)` executes a loop to print the name and id for each element in `pps`, as appropriate.
  4. The function `release(...)` releases all the dynamically allocated memory in a correct manner.

- **MANDATORY FOR ALL STUDENTS**
  1. Use references (instead of call by value) wherever applicable in each method or function.
  2. Write the keyword "`const`" wherever applicable in each method or function.
  3. Explain what will happen and why if the `Student` object in `add(...)` is instantiated using static (not dynamic) memory.

```
#include <iostream>
#include <string>
using namespace std;

class Student { ... };                                  // write the code

void add(...  i)                                        // write the code
{
  // dynamically create Student object using input data
  // assign object to i^th element of pps
  // do nothing if value of i out of bounds
}

void remove(... j)                                      // write the code
{
  // remove element at location j in pps
  // do nothing if value of j out of bounds
}

void display(...)                                       // write the code
{
  // loop over elements of pps, print name and id as appropriate
}

void release(...)                                       // write the code
{
  // release dynamic memory correctly
}
```

main program on next page

```
int main()
{
  int n = 0;
  cout << "Please enter the number of students (positive integer): " << endl;
  cin >> n;

  Student **pps = ...           // WRITE CODE, DYNAMIC ALLOCATION ARRAY OF POINTERS

  for (int i = 0; i < n; i++) {
    string s;
    int id = 0;
    cout << "Please enter a name and id (positive integer): " << endl;
    cin >> s >> id;
    add(pps, s, id, n, i);
  }

  int j = 0;
  cout << "Please enter an integer >= 0: " << endl;
  cin >> j;
  remove(pps, n, j);

  display(pps,n);
  release(pps,n);
  return 0;
}
```

4 Question 4

- **This question is about forward declarations and non-inline class methods.**
- You are given three clases `Owner`, `Rider` and `Horse`. An owner owns one or more horses and employs one or more riders. A rider works for an owner. A horse belongs to an owner and a rider is assigned to it. Here are the class declarations for `Owner`, `Rider` and `Horse`.

```
class Owner {
public:
  Owner(string n);
  ... getName();
  ... addHorse(...);
  ... addRider(...);
  ... getHorse(int i);    // return NULL if value of i not valid
  ... getRider(int i);    // return NULL if value of i not valid
  ... numHorses();
  ... numRiders();
private:
  const string name;
  vector<const Rider*> vec_rider;  // vector of const pointers
  vector<const Horse*> vec_horse;  // vector of const pointers
};

class Rider {
public:
  Rider(string n, const Owner &o);
  ... getName();
  ... getOwner();
private:
  const string name;
  const Owner *owner;
};

class Horse {
public:
  Horse(string n, const Owner &o, const Rider &r);
  ... getName();
  ... getOwner();
  ... getRider();
private:
  const string name;
  const Owner *owner;
  const Rider *rider;
};
```

- The functions in the `Owner` class perform the following tasks:
  1. Constructor: set the name and initialize vectors to empty vectors.
  2. `getName()`: return the name of the owner.
  3. `addHorse(...)`: add a pointer to `Horse` to the end of the relevant vector.
  4. `addRider(...)`: add a pointer to `Rider` to the end of the relevant vector.
  5. `getHorse(...)`: return the i^{th} element in the vector, else NULL if value of i is not valid
  6. `getRider(...)`: return the i^{th} element in the vector, else NULL if value of i is not valid
  7. `numHorses()`: return the number of horses owned by the owner
  8. `numRiders()`: return the number of riders employed by the owner
- The functions in the `Rider` class perform the following tasks:
  1. Constructor: set the name and the owner (employer).
  2. `getName()`: return the name of the rider.
  3. `getOwner(...)`: return a const reference to the owner.
- The functions in the `Horse` class perform the following tasks:
  1. Constructor: set the name and the owner and rider.
  2. `getName()`: return the name of the rider.
  3. `getOwner(...)`: return a const reference to the owner.
  4. `getRider(...)`: return a const reference to the rider.
- **Write suitable constructors and methods for all of the above classes, with appropriate declarations and function bodies and return types.**
- **Class methods which MUST be non-inline should be declared and written appropriately.**
- **Insert the keyword `const` (or use const references) wherever applicable.**

- **MANDATORY FOR ALL STUDENTS**
- **Write a function "`int main()`" and place the code below in it.**

1. You are given the following information. There are two owners with names "Alice" and "Bob" respectively. There are three riders with names "Alpha" and "Beta" (both employed by Alice) and "Gamma" (employed by Bob).
2. **Write suitable code to instantiate the above objects and invoke `addRider(...)`.**

```
Owner ...                       // pseudocode, write in full
Rider.,,,                       // pseudocode, write in full

"Alice" addRider( "Alpha" );    // pseudocode, write in full
"Alice" addRider( "Beta" );
"Bob"   addRider( "Gamma" );
```

3. **Create four `Horse` objects by dynamic allocating memory as follows.**

```
Horse **h = new Horse*[4];
h[0] = ("Blue",   Alice, Alpha);     // pseudocode, write in full
h[1] = ("Flash",  Alice, Beta);
h[2] = ("Gold",   Bob,   Gamma);
h[3] = ("Silver", Bob,   Gamma);

"Alice" add horse h[0]               // pseudocode, write in full
"Alice" add horse h[1]
"Bob"   add horse h[2]
"Bob"   add horse h[3]
```

4. **For each owner, write code to print the name, number of riders and numbers of horses.**

```
"Alice"  name,  number of riders,  number of horses
"Bob"    name,  number of riders,  number of horses
```

5. You are given the following array of strings (4 strings).
   Write a loop to print the following on each pass.

```
string races[] = { "Derby", "Preakness", "Belmont", "Ascot" };

for (int i = 0; i < 4; ++i) {
  cout << "race: " << races[i] << endl;
  cout << "winning horse: " <<           // name of horse h[i]
  cout << "rider: " <<                   // name of rider of horse h[i]
  cout << "owner: " <<                   // name of owner of horse h[i]
}
```

6. **Write code to release the dynamic memory correctly.**

- **End of function `main()`.**