CSCI-1200 Data Structures — Fall 2021
Lecture 3 — Classes II: More Examples, More Syntax

Review of Lecture 2

- C++ classes, member variables and member functions, class scope, public and private
- Nuances to remember
  - Within class scope (within the code of a member function) member variables and member functions of that class may be accessed without providing the name of the class object.
  - Within a member function, when an object of the same class type has been passed as an argument, direct access to the private member variables of that object is allowed (using the '.' notation).
  - Classes vs. structs, Designing classes
  - Custom sorting of class instances, non-member operators

Today's Lecture

- Review: Custom sorting of class instances, non-member operators
- More Misc C++ / C++ Class Syntax
  - default function parameter values
  - constructor member variable initializer list
  - include guards
  - return w/ const and reference
  - writing our own output stream operator
- Extended example w/ multiple classes: student grading program
  - more complex types as class variables
  - member function implementation in the header file
  - default constructor, copy constructor

Announcements / Reminders

- Office Hours http://www.cs.rpi.edu/academics/courses/fall21/csci1200/schedule.php
  - In addition to DS TAs and mentors, ALAC tutors will be helping students from the office hours queue.
  - Most office hours are in person, some evening office hours will be partially or completely online. Carefully read the information on the Office Hours Queue page then join the appropriate queue: https://submitty.cs.rpi.edu/courses/f21/csci1200/office_hours_queue
- Discussion Forum https://submitty.cs.rpi.edu/courses/f21/csci1200/forum
  - Please search the prior threads before making a new thread. Filter by category, and also use the text search feature. We will merge duplicate questions into the earlier thread when appropriate.
  - Please write a good title for your new thread, so other students can find your question and benefit from our answers.
  - Teaching staff (instructor, ta, mentor) posts on the forum are outlined in yellow – this helps you identify verified answers. Students are encouraged to answer questions too! (Some of our top student forum participants become mentors in future semesters!) Teaching staff monitor all posts and will jump in with clarifications/corrections to answers if necessary.
- Emailing the Professor – please use `ds_instructor@cs.rpi.edu`
  - There are approximately 300 students in the course – my inbox gets overrun and replies are massively delayed.
    Please limit email to personal and confidential communications.
  - Try to ask your questions on the Discussion Forum first – other students may have the same question! You can post anonymously – the instructor and TA can see your user id if that's necessary to help you.

  - Do not attach your homework to the email. The instructor and graduate TAs can access your homework files on Submitty.

- Please update your Submitty Profile https://submitty.cs.rpi.edu/user_profile
  - Preferred first/last names
  - Upload an alternate passport-style photo – the lab TAs & mentors can see these photos and it helps us learn your names.
  - Specify an alternate, non-rpi email in order to receive announcement emails without VPN access.

Frequently Asked Questions

- Q: What things (techniques/data structures/library functions/C++ syntax) can I use on the homework? (And similarly, what things can I use on an exam problem?)

  A: Generally, anything that has been presented in lecture up to that point. But read the whole homework handout (or exam problem) to check if there are specific things that are required or specific things that are forbidden. Using advanced features can sometimes make an assignment "too easy" or can prevent you from demonstrating skill at the things targeted by that homework (or exam problem).

- Q: If I lose points on the Submitty autograding, will a TA look at my output and give me those points back?

  A: No. TAs will not change autograding (typically ∼ 50% of the HW grade). TAs will be grading overall software quality, and making sure you demonstrate mastery of the skills targeted by that assignment (remaining ∼ 50%). Some homeworks will have written answers in the README.txt and TAs will be grading that manually.

- Q: Will I receive a deduction on my homework if I don't follow instructions?

  A: Yes, probably. The TAs are grading lots of homework, so they won't read every line of your program. For example, we may or may not notice if you sneak a global variable into the code, but if you're supposed to use vector and you don't use vector at all we will almost certainly notice.

- Q: What is rainbow grades? When will rainbow grades be posted? When will rainbow grades be updated? What score do I need to pass the course? What score do I need to get an A? What is the cutoff between an A and an A-? Should I use a late day to get 5 more points on this homework or should I save my late day for a future homework? Etc.

  A: I will usually ignore these questions. They are not about computer science or data structures or learning.

- Q: Can I makeup my lab checkpoints?

  A: If you have a good and reasonable excuse (unavoidable personal scheduling conflict, illness) you can contact your lab TA by email to arrange a makeup. Contact them as soon as you know you will be absent. This should be multiple days or a week in advance for a religious holiday or other pre-arranged absence. If you ask for lab makeup on multiple labs or if your requests do not seem reasonable, you will be reported to the instructor – who is not nearly as generous with makeups.

- Q: Can I get an extension on my Homework? My submission was only 2 seconds late, do I have to use my late day? Most of my assignment was on time, but I forgot my README file – can you add that in without charging me a late day?

  A: No. Yes. No. We strictly enforce the homework late day policy:*
  http://www.cs.rpi.edu/academics/courses/fall21/csci1200/homework_policies.php

  ** If you have a significant illness or personal emergency, contact the Office of Student Life and/or the RPI Health Center to request a formal excused absence extension. This is the only exception to the late day policy.*

- Q: Where are the lecture videos? Can I makeup the Submini polls if I miss lecture?

  A: The lecture videos from Fall 2020 are on Mediasite and will be posted on the calendar on the day of lecture and will be very similar to Fall 2021 – EXCEPT for the discussion of homework and exams, which will be different this term. The lectures from Fall 2021 will be recorded and uploaded to Mediasite and posted on the calendar – but the upload process is slow and will be delayed a few days. Submini lecture participation polls will only be available during the lecture period and there will be no opportunity to makeup Submini polls.

3.1 Simple Class Example: Alphabetizing Names, name_main.cpp

```
#include <algorithm>
#include <iostream>
#include <vector>
#include "name.h"

int main() {
  std::vector<Name> names;
  std::string first, last;
  std::cout <<"\nEnter a sequence of names (first and last) for alphabetization\n";
  while (std::cin >> first >> last) {
    names.push_back(Name(first, last));
  }
  std::sort(names.begin(), names.end());
  std::cout << "\nHere are the names, in alphabetical order.\n";
  for (int i = 0; i < names.size(); ++i) {
    std::cout << names[i] << "\n";
  }
  return 0;
}
```

3.2 name.h

```
// These "include guards" prevent errors from "multiple includes"
#ifndef __name_h_
#define __name_h_

#include <iostream>
#include <string>

class Name {
public:
  // CONSTRUCTOR (with default arguments)
  Name(const std::string& fst="", const std::string& lst="");
  // ACCESSORS
  // Providing a const reference to the string allows the string to be
  // examined and treated as an r-value without the cost of copying it.
  const std::string& first() const { return first_; }
  const std::string& last() const { return last_; }
  // MODIFIERS
  void set_first(const std::string & fst) { first_ = fst; }
  void set_last(const std::string& lst) { last_ = lst; }
private:
  // REPRESENTATION
  std::string first_, last_;
};

// operator< to allow sorting
bool operator< (const Name& left, const Name& right);

// operator<< to allow output
std::ostream& operator<< (std::ostream& ostr, const Name& n);

#endif
```

- Create an instance of the Name class:
  `Name person("Tom","Smith");`
- Attempt to permanently change Tom's first name:
  ```
  const std::string &a = person.first();
  a[1] = 'i';  // will fail to compile because a is const
  ```
- Make a copy of the returned `string`:
  ```
  std::string b = person.first();
  b[1] = 'i';  // edit does not affect the person object
  ```

3.3 name.cpp

```
#include "name.h"

// Here we use member variable initializer list syntax to call the string
// class copy constructors (this way is technically more efficient).
Name::Name(const std::string& fst, const std::string& lst)
  : first_(fst), last_(lst)
{}
// Alternative implementation first calls default string constructor for the
// two variables, then performs an assignment in the body of the constructor
// function (this way is technically less efficient).  For this example,
// both versions will yield the same overall resulting program behavior.
/*
Name::Name(const std::string& fst, const std::string& lst) {
  first_ = fst;
  last_ = lst;
}
*/

// operator<
bool operator< (const Name& left, const Name& right) {
  return left.last()<right.last() ||
    (left.last()==right.last() && left.first()<right.first());
}
// The output stream operator takes two arguments: the stream (e.g., cout)
// and the object to print.  It returns a reference to the output stream
// to allow a chain of output.
std::ostream& operator<< (std::ostream& ostr, const Name& n) {
  ostr << n.first() << " " << n.last();
  return ostr;
}
```

3.4 Larger Example Using Multiple Classes: Student Grades

Our goal is to write a program that calculates the grades for students in a class and outputs the students and their averages in alphabetical order. The program source code is broken into three parts:

- Re-use of statistics code from Lecture 2.
- Class `Student` to record information about each student, including name and grades, and to calculate averages.
- Note: The `Student` class uses STL `string`, STL `vector`, and the `Name` class defined above!
- The main function controls the overall flow, including input, calculation of grades, and output.

3.5 Sample Input

```
6 2
0.4
11111111 George Washington 72 78 34 78 80 82  69 80
10232145 Benjamin Franklin 87 62 26 94 98 72   68 88
73413414 Lighthorse Lee  89 97 45 78 77 80   85 82
15442765 Robert Lee 99 98 78 93 93 95  92 98
38475452 Samuel Adams 64 55 28 64 69 70   60 66
23415634 Abigail Adams 94 97 69 90 95 96  96 95
42520984 John Adams  92 94 62 88 97 94   92 94
24589724 Patrick Henry  66 65 44 79 75 69   66 69
```

3.6 Desired Output

```
Here are the student semester averages
Name                HW  Test Final
-----------------------------------
Adams, Abigail      90.2 95.5 93.4
Adams, John         87.8 93.0 90.9
Adams, Samuel       58.3 63.0 61.1
Franklin, Benjamin  73.2 78.0 76.1
Henry, Patrick      66.3 67.5 67.0
Lee, Lighthorse     77.7 83.5 81.2
Lee, Robert         92.7 95.0 94.1
Washington, George  70.7 74.5 73.0
```

3.7 student_main.cpp

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <vector>
#include "student.h"

int main(int argc, char* argv[]) {
  if (argc != 3) {
    std::cerr << "Usage:\n  " << argv[0] << " infile-students outfile-grades\n";
    return 1;
  }
  std::ifstream in_str(argv[1]);
  if (!in_str) {
    std::cerr << "Could not open " << argv[1] << " to read\n";
    return 1;
  }
  std::ofstream out_str(argv[2]);
  if (!out_str) {
    std::cerr << "Could not open " << argv[2] << " to write\n";
    return 1;
  }

  int num_homeworks, num_tests;
  double hw_weight;
  in_str >> num_homeworks >> num_tests >> hw_weight;
  std::vector<Student> students;
  Student one_student;

  // Read the students, one at a time.
  while(one_student.read(in_str, num_homeworks, num_tests)) {
    students.push_back(one_student);
  }

  // Compute the averages.  At the same time, determine the maximum name length.
  unsigned int i;
  unsigned int max_length = 0;
  for (i=0; i<students.size(); ++i) {
    students[i].compute_averages(hw_weight);
    unsigned int tmp_length = students[i].first_name().size() + students[i].last_name().size();
    max_length = std::max(max_length, tmp_length);
  }
  max_length += 2;   // account for the output padding with ", "

  // Sort the students alphabetically by name.
  std::sort(students.begin(), students.end(), less_names);

  // Output a header...
  out_str << "\nHere are the student semester averages\n";
  const std::string header = "Name" + std::string(max_length-4, ' ') + "  HW  Test Final";
  const std::string underline(header.size(), '-');
  out_str << header << '\n' << underline << std::endl;

  // Output the students...
  for (i=0; i<students.size(); ++i) {
    unsigned int length = students[i].last_name().size() +
      students[i].first_name().size() + 2;
    students[i].output_name(out_str);
    out_str << std::string(max_length - length, ' ') << "  ";
    students[i].output_averages(out_str);
  }

  return 0;  // everything fine
}
```

3.8 Declaration of Class `Student`, `student.h`

- Stores names, id numbers, scores and averages. Member variables of a class can be STL classes (e.g., `string`) or custom classes (e.g. `Name`)! The raw scores are stored using an STL `vector`!
- Functionality is relatively simple: input, compute average, provide access to names and averages, and output.
- No constructor is explicitly provided: `Student` objects are built through the `read` function.
  (Other code organization/designs are possible!)
- Overall, the `Student` class design differs substantially in style from the `Date` class design. We will continue to see different styles of class designs throughout the semester.
- Note the helpful convention used in this example: all member variable names end with the "`_`" character.
- The special pre-processor directives `#ifndef __student_h_`, `#define __student_h_`, and `#endif` ensure that this files is included at most once per `.cpp` file.

  For larger programs with multiple class files and interdependencies, these lines are essential for successful compilation. We suggest you get in the habit of adding these *include guards* to all your header files.

```
#ifndef __student_h_
#define __student_h_
#include <iostream>
#include <string>
#include <vector>
#include "name.h"

class Student {
public:
  // ACCESSORS
  const std::string& first_name() const { return name_.first(); }
  const std::string& last_name() const { return name_.last(); }
  const std::string& id_number() const { return id_number_; }
  double hw_avg() const { return hw_avg_; }
  double test_avg() const { return test_avg_; }
  double final_avg() const { return final_avg_; }
  // MODIFIERS
  bool read(std::istream& in_str, unsigned int num_homeworks, unsigned int num_tests);
  void compute_averages(double hw_weight);
  // PRINT HELPER FUNCTIONS
  std::ostream& output_name(std::ostream& out_str) const;
  std::ostream& output_averages(std::ostream& out_str) const;
private:
  // REPRESENTATION
  Name name_;
  std::string id_number_;
  std::vector<int> hw_scores_;
  double hw_avg_;
  std::vector<int> test_scores_;
  double test_avg_;
  double final_avg_;
};

// COMPARISON FUNCTION FOR SORTING
bool less_names(const Student& stu1, const Student& stu2);
#endif
```

3.9 Automatic Creation of Two Constructors By the Compiler

- Two constructors are created automatically by the compiler because they are needed and used.
- The first is a default constructor which has no arguments and *just calls the default constructor for each of the member variables.* The prototype is `Student();`

  The default constructor is called when the `main()` function line `Student one_student;` is executed.

  If you wish a different behavior for the default constructor, you must declare it in the `.h` file and provide the alternate implementation.

- The second automatically-created constructor is a "copy constructor", whose only argument is a `const` reference to a `Student` object. The prototype is `Student(const Student &s);`

  This constructor *calls the copy constructor for each member variable* to copy the member variables from the passed `Student` object to the corresponding member variables of the `Student` object being created. If you wish a different behavior for the copy constructor, you must declare it an provide the alternate implementation.

  The copy constructor is called during the vector `push_back` function in copying the contents of `one_student` to a new `Student` object on the back of the vector `students`.

- The behavior of automatically-created default and copy constructors is often, but not always, what's desired. When they do what's desired, the convention is to not write them explicitly.

- Later in the semester we will see circumstances where writing our own default and copy constructors is crucial.

3.10 Implementation of Class `Student`, `student.cpp`

- The `read` function is fairly sophisticated and depends heavily on the expected structure of the input data. It also has a lot of error checking.
  - NOTE: Alternatively, we could do this input parsing *outside of the class*, and pass the results into the class constructor. Separating the clunky I/O details from the class implementation may be a *better design*. Why? The class could potentially be re-used for a different project with a different input file format!

- The accessor functions for the names are defined within the class declaration in the header file. **In this course, you are allowed to do this for one-line functions only!** For complex classes, including long definitions within the header file has dependency and performance implications.

- The computation of the averages uses some but not all of the functionality from `stats.h` and `stats.cpp` (not included in today's handout).

- Output is split across two functions. Again, stylistically, it is sometimes preferable to do this outside the class.

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include "stats.h"
#include "student.h"
// note: this next #include is unnecessary because student.h already includes name.h,
// but it would not cause an error, because name.h has "include guards".
// #include "name.h"

// Read information about a student, returning true if the information was read correctly.
bool Student::read(std::istream& in_str, unsigned int num_homeworks, unsigned int num_tests) {

  // If we don't find an id, we've reached the end of the file & silently return false.
  if (!(in_str >> id_number_)) return false;
  // Once we have an id number, any other failure in reading is treated as an error.

  // read and construct the name
  std::string first_name,last_name;
  if (! (in_str >> first_name >> last_name)) {
    std::cerr << "Failed reading name for student " << id_number_ << std::endl;
    return false;
  }
  name_ = Name(first_name,last_name);

  // Read the homework scores
  unsigned int i;
  int score;
  hw_scores_.clear();
  for (i=0; i<num_homeworks && (in_str >> score); ++i) { hw_scores_.push_back(score); }
  if (hw_scores_.size() != num_homeworks) {
    std::cerr << "ERROR: end of file or invalid input reading hw scores for " << id_number_ << std::endl;
    return false;
```

```
  }

  // Read the test scores
  test_scores_.clear();
  for (i=0; i<num_tests && (in_str >> score); ++i) { test_scores_.push_back(score); }
  if (test_scores_.size() != num_tests) {
    std::cerr << "ERROR: end of file or invalid input reading test scores for" << id_number_ << std::endl;
    return false;
  }
  return true;  // everything was fine
}

// Compute and store the hw, test and final average for the student.
void Student::compute_averages(double hw_weight) {
  double dummy_stddev;
  avg_and_std_dev(hw_scores_, hw_avg_, dummy_stddev);
  avg_and_std_dev(test_scores_, test_avg_, dummy_stddev);
  final_avg_ = hw_weight * hw_avg_ + (1 - hw_weight) * test_avg_;
}

std::ostream& Student::output_name(std::ostream& out_str) const {
  out_str << last_name() << ", " << first_name();
  return out_str;
}

std::ostream& Student::output_averages(std::ostream& out_str) const {
  out_str << std::fixed << std::setprecision(1);
  out_str << hw_avg_ << " " << test_avg_ << " " << final_avg_ << std::endl;
  return out_str;
}

// Boolean function to define alphabetical ordering of names.  The vector sort
// function requires that the objects be passed by CONSTANT REFERENCE.
bool less_names(const Student& stu1, const Student& stu2) {
  return stu1.last_name() < stu2.last_name() ||
         (stu1.last_name() == stu2.last_name() && stu1.first_name() < stu2.first_name());
}
```

3.11 Providing Comparison Functions to Sort

Consider sorting the students vector:

- If we used `sort(students.begin(), students.end());` the sort function would try to use the `<` operator on `student` objects to sort the students, just as it earlier used the `<` operator on doubles to sort the grades. However, this doesn't work because there is no such operator on `Student` objects.
- Fortunately, the sort function can be called with a third argument, a comparison function:
  `sort(students.begin(), students.end(), less_names);`

  `less_names`, defined in `student.cpp`, is a function that takes two const references to `Student` objects and returns true if and only if the first argument should be considered "less" than the second in the sorted order. `less_names` uses the `<` operator defined on `string` objects to determine its ordering.

3.12 Exercises

- Add code to the end of the `main()` function to compute and output the average of the semester grades and to output a list of the semester grades sorted into increasing order.
- Write a function `greater_averages` that could be used in place of `less_names` to sort the `students` vector so that the student with the highest semester average is first.