

CSCI-1200 Computer Science II — Spring 2006

Lecture 6 — Classes II Sort, Non-member Operators

Announcements: Test 1 Information

- Test 1 will be held **Tuesday, February 7th, 2006 from 10-11:50am in West Hall Auditorium**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 1-6, Labs 1-3, HW 1-2.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, etc. are not permitted and must be turned off.
- All students must bring their Rensselaer photo ID card.
- Practice problems are available on the course website. There will be an optional review session on Monday, February 6th from 6-8pm in DCC 324. Bring questions!

Review from Lecture 5 and Lab

- C++ classes, member variables and member functions, class scope, public and private
- Nuances to remember (many of these are syntactic)
 - 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

Today's Lecture

- Extended example of student grading program
- Passing comparison functions to `sort`
- Non-member operators

6.1 Example: 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 that we wrote in Lecture 4.
- Class `Student` to record information about each student, including name and grades, and to calculate averages.
- The main function controls the overall flow, including input, calculation of grades, and output.

```
// File:      main_student.cpp
// Author:    Chuck Stewart
// Purpose:   Compute student averages and output them alphabetically.

#include <algorithm>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>
```

```

#include "student.h"

unsigned int max( unsigned int x, unsigned int y ) {
    return x>y ? x : y;
}

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 );
        max_length = max(max_length,
            students[i].first_name().size()
            + students[i].last_name().size());
    }
    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
}

```

6.2 Declaration of Class Student

- Stores names, id numbers, scores and averages. The scores are stored using a vector! Member variables of a class can be other classes!
- 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.
- Overall, the `Student` class design differs substantially in style from the `Date` class design. We will 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.

```
// File:      student.h
// Author:    Chuck Stewart
// Purpose:   Header for declaration of student record class and
//           associated functions.

#ifndef student_h_
#define student_h_
#include <iostream>
#include <string>
#include <vector>

class Student {
public:
    bool read( std::istream& in_str,
              unsigned int num_homeworks,
              unsigned int num_tests );

    void compute_averages( double hw_weight );

    const std::string& first_name() const { return first_name_; }
    const std::string& last_name()  const { return last_name_; }
    const std::string& id_number()  const { return id_number_; }
    double hw_avg() const { return hw_avg_; }
    double test_avg() const { return test_avg_; }

    std::ostream& output_name( std::ostream& out_str ) const;
    std::ostream& output_averages( std::ostream& out_str ) const;

private:
    std::string first_name_;
    std::string last_name_;
    std::string id_number_;
    std::vector<int> hw_scores_;
    double hw_avg_;
    std::vector<int> test_scores_;
    double test_avg_;
    double final_avg_;
};

bool less_names( const Student& stu1, const Student& stu2 );

#endif
```

6.3 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 and 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.

6.4 Implementation of Class Student

- 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.
 - In many class designs, this type of input would be done by functions outside the class, with the results passed into a constructor. We generally prefer this style because it separates elegant class design from clunky I/O details.
- The accessor functions for the names are defined within the class declaration in the header file. **Unless explicitly stated otherwise, in this course, you are allowed to do this for one-line functions only!**
- The computation of the averages uses some but not all of the functionality from `stats.h` and `stats.cpp` (not included in your handout).
- Output is split across two functions. Again, stylistically, it is sometimes preferable to do this outside the class.
- We will discuss the non-member function `less_names` later in this lecture.

```
// File:      student.cpp
// Author:    Chuck Stewart
// Purpose:   Implementation of the class Student

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include "student.h"
#include "stats.h"

// Read information about a student, returning true if the information was read
// correctly and false otherwise.  The input format is id_number first_name last_name
// homeworks tests.  num_homeworks and num_tests ints are expected for the latter.

bool Student::read(std::istream& in_str, unsigned int num_homeworks, unsigned int num_tests) {
    // Start with the id number.  If there’s no id, then end-of-file
    // will likely have been reached, so 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.  Start by reading the name.
```

```

if ( ! (in_str >> first_name_ >> last_name_)) {
    std::cout << "Failed reading name for student "
        << id_number_ << std::endl;
    return false;
}

unsigned int i;
int score;

// Read the homework scores
hw_scores_.clear();
for ( i=0; i<num_homeworks && (in_str >> score); ++i)
    hw_scores_.push_back( score);

// If we didn't get all scores, it is an error.
if ( hw_scores_.size() != num_homeworks) {
    std::cout << "Pre-mature 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 we didn't get all scores, it is an error.
if (test_scores_ . size() != num_tests) {
    std::cout << "Pre-mature 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.
// Here we use the stats package from lecture 4
void Student::compute_averages(double hw_weight) {
    double dummy_stddev;
    avg_and_stddev(hw_scores_, hw_avg_, dummy_stddev);
    avg_and_stddev(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 determine if one student should come earlier
// than another based on an alphabetical order of names. It is very,
// VERY important that the student objects be passed by CONSTANT
// REFERENCE here. That is what the sort function is expecting.
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());
}

```

6.5 New C++ in the main function — manipulating strings

- The statement:

```
const string header = "Name" + string(max_length-4, ' ') + " HW Test Final";
```

creates a constant string by “adding” (concatenating) existing strings.

- The expression `string(max_length-4, ' ')` within this statement creates a temporary string but does not associate it with a variable.
- This is done again during the output of each individual student to create an evenly spaced table.

6.6 Exercise

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.

6.7 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.

6.8 Exercise

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.

6.9 Operators As Non-Member Functions

- A second option for sorting is to define a function that creates a `<` operator for `Student` objects! At first, this seems a bit weird, but it is extremely useful.
- Let’s start with syntax. The expressions `a < b` and `x + y` are really function calls! Syntactically, they are equivalent to `operator<(a, b)` and `operator+(x, y)` respectively.
- When we want to write our own operators, we write them as functions with these weird names.

- For example, if we write:

```
bool operator< (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());
}
```

then the statement `sort(students.begin(), students.end());` will sort `Student` objects into alphabetical order.

- Really, the only weird thing about operators is their syntax.
- We will have many opportunities to write operators throughout this course. Sometimes these will be made class member functions, but that comes later.

6.10 A Word of Caution about Operators

- Operators should only be defined if their meaning is intuitively clear.
- `operator<` on `Student` objects fails the test because the natural ordering on these objects is not clear.
- By contrast, `operator<` on `Date` objects is much more natural and clear.

6.11 Exercise

Write an `operator<` for comparing two `Date` objects.

6.12 Another Class Example: Alphabetizing Names

```
// main.cpp
//
// Demonstrates another example with the use of classes, including an output
// stream operator

#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) and this program will alphabetize them\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;
}
```

```

// name.h
//
// Demonstrates another example with the use of classes, including an
// output stream operator

#include <iostream>
#include <string>

class Name {
public:

    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:
    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 );

```

```

// name.cpp

#include "name.h"

// The constructor is the only member function that is not
// defined in the .h file.
//
// Here we use special syntax to call the string class copy
// constructors instead of calling the default constructor and then
// performing an assignment in the body of the constructor function.
Name::Name(const std::string& fst, const std::string& lst)
    : first_(fst), last_(lst)
{}

// operator< There is little surprise here.
bool operator< ( const Name& left, const Name& right ) {
    return left.last() < right.last() ||
        ( left.last() == right.last() && left.first() < right.first() );
}

// operator<< is the output stream operator. It takes two arguments:
// the stream (such as cout) and the name to be output. The actual
// output uses the ability to output strings. Then, the operator
// returns a reference to the output stream. This allows a sequence
// of outputs in a single statement.
std::ostream& operator<< ( std::ostream& ostr, const Name& n ) {
    ostr << n.first() << " " << n.last();
    return ostr;
}

```