Computer Science II — CSci 1200
Lecture 4
Classes, Sort, Non-member Operators

Review from Lecture 3 and Lab

- C++ classes
- The `Date` class example.
- Member variables and member functions
- Class scope
- Public and private
- Nuances to remember
  - Each member function has its own copy of the member variables
  - 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

Example: Student Grades

Our goal is to write a program that calculates the grades for students in a course and outputs the students and their averages in alphabetical order. An example of running this program will be shown during lecture.

Overall Organization

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.
- The main function controls the overall flow, including input, calculation of grades, and output.

Declaration of class Student

- Names, id numbers, scores and averages are all stored.
  - Importantly, this illustrates using vectors and other containers as member variables inside the class.
- Functionality is relatively straightforward. The member functions allow storing information, compute averages, provide access to names and computed averages, and output some values.
- Two constructors are provided.
- The main "construction" of a student object for this program is provided through a non-member `read_student` function.
  - This function is specific to the form of the input and contains extensive error checking.
  - The `Student` class does not know anything about the form of the input. This makes the class reusable for different input formats.
  - Responsibility for ensuring that the input is consistent between different `Student` objects lies outside the class.

Automatic Creation of Two Constructors By the Compiler

- If we provide no constructors, the compiler would automatically create two of them.
- The first is a default constructor which has no arguments and just calls the default constructor for each of the member variables.
- The default constructor is called when the main function line

  ```
  Student one_student;
  ```

  is executed. In this case, however, we have defined our own default constructor.
  - When we create any constructor, the compiler will not automatically create the default constructor.
- The second automatically created constructor is a "copy constructor", which has a `const` reference to a `Student` object as its only argument and copies each member variable from the passed `Student` object to the corresponding member variable of the `Student` object being created.
  - It 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 is desired.
- Later in the semester we will see circumstances where writing our own default and copy constructors is crucial.

Implementation of class Student

- 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`.
- Output is split across two functions.
  - Stylistically, it is sometimes preferrable to do this outside the class.
- We will discuss the non-member function `less_names` later in this lecture.

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.

Exercise

Add code to the end of the main program to compute and output the average of the semester grades and to output a list of the semester grades sorted into increasing order.

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.
- 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.

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.

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 wrote

```
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.

- In sort, 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.

A word of caution about operators

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

Exercise

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

Mode

If we have time at the end of lecture we will discussion computation of the mode in `stats.cpp`.