Introduction

This assignment is a warm-up exercise, so that you get comfortable with programming in C++, and using the Unix command line. It will help prepare you for future assignments in this class.

Description

You will write a program that reads employee data from an unsorted binary file, sorts the employee data by id number, and then writes the sorted data back to a binary file. Your program should also print the sorted records to the screen.

Your program will read the employee data into a `vector` of `Employee` objects. To sort the vector, you may use the `std::sort` function defined in the `algorithm` header file. Note that you will need to define a *comparator* for sort to work on your data.

Getting Started

Download the file `dm-proj00.tar.gz`, which is a compressed archive containing all of the files you will need to complete the assignment. To get started, you can run the following commands:

```
$ tar xvfz dm-proj00.tar.gz
$ cd dm-proj00
```

Files

The `dm-proj00` directory contains the following files, which you will need to use:

./src/Main.cpp This is where the main function of your program is, and where you will need to add your code. The source file is heavily commented, explaining step-by-step what you need to write.

./src/Employee.cpp This file defines a class to hold your Employee record data. You should not need to make any edits to this file.

./include/Employee.h This is the header file for the Employee class.

./input/test.bin This is a test input file for your program. It contains a number of Employee records, written in binary format, as described above.

Makefile This is the input to the `make` utility. You will not need to edit this file.

./bin/employee-sort After you compile your program with `make`, the executable will be in a `bin` directory.

Binary File

A *binary* file is one in which data is stored in *binary* format, as opposed to a text file, which stores data in ascii format. A binary file is more compact than a text file, and can therefore store data more efficiently.

For example, the integer value 12345 is 11000000111001 in binary. In a binary file, an int is stored in 4 bytes. However, to store the same integer value 12345 in an ascii file, we would need 5 bytes (one byte of each digit). In other words, we would need 1 byte for the ascii char '1', (i.e., 00110001), another byte for the ascii char '2', (00110010), and so on.

Employee File Format

Employee data is stored as a sequence of records in an employee file. Each employee record has three fields: an id (4 byte integer), a name (variable length), and a salary (8 byte double). Because the name field can be variable length, we must encode the size of the name when we read and write it to the file. The format of each record is as follows:

4 byte integer (Employee id)	4 byte integer (Number of characters in employee name)	N character string (Employee name)	8 byte double (Employee salary)

Reading and Writing Binary Files

To open a binary file, you can use the following code. It creates an instance of an `fstream` named `infile`. The constructor to `fstream` takes two parameters, the filename, and a mode, represented by bits. The `|` is the binary or operator. The `ios::in` means the file will be read, and the `os::binary` means that it is a binary file.

```
fstream infile(filename, ios::in | ios::binary);
```

You can read a variable length string with the code below. You will first read an int, which will give you the number of characters in the name. Then you will read that number of characters. Be careful. Before you read into the variable `name`, you will need to allocate memory. (How much memory will you need?)

```
int nameLen;
char *name;
...
infile.read((char*)&nameLen, sizeof(int));
...
infile.read(name, nameLen);
```

Writing a file is similar to reading. To open a file for writing, use `ios::out`.

```
fstream outfile(filename, ios::out | ios::binary);
int nameLen;
char *name;
...
outfile.write((char*)&nameLen, sizeof(int));
...
outfile.write(name, nameLen);
outfile.close();
```

Standard Template Library

The C++ Standard Template Library (STL) includes a number of common algorithms and data structures. The important ones for this assignment are `vector` and `sort`. But, there are other containers, such as `map` and `queue`, and algorithms that you may find useful in the future. These data structures are parameterized with a type, which is usually written `T`, such as in `vector<T>`. To use the container, you will need to pass the correct type parameter. For example, a vector of ints is written `vector<int>`.

Containers can be traversed using an *iterator*. Iterators are also parameterized with a type. To traverse a vector of ints, you need a `vector<int>::iterator`.

The code below shows a declaration of a vector of ints, two push operations to insert data, and a traversal to print the elements using an iterator:

```
#include <vector>
#include <iostream>

int main(int argc, char* argv[]) {
  std::vector <int> v;    // declare a vector
  v.push_back(1);         // insert 1
  v.push_back(2);         // insert 2

  std::vector<int>::iterator it; // an 'iterator' to traverse the vector
  for (it = v.begin(); it != v.end(); it++) {
    int x = *it;
    std::cout << x << std::endl;  // prints '1', then '2'...
  }
  return 0;
}
```

Sorting Employee Data

You may sort the records any way you like. One easy way is to use the `std::sort` function, which will sort a `vector`. In order to sort a vector of `Employee` objects, you will need to define a comparator function. Below is an example for calling sort.

```
  std::vector<Employee> employees;
  std::sort( employees.begin(), employees.end(), byEmployeeId());
```

The comparator is a functor for comparing Employee references. A functor is simply a class or struct that defines an operator(), which means that they can be called in a similar style to how you would call a function. Here is an example, with the logic for doing the actual comparison missing:

```
struct byEmployeeId {
  bool operator()( const Employee& lhs, const Employee& rhs ) const {
      // your code here
  }
};
```

Printing Employee Data

The `Employee` class defines a `toString` function, which will convert the employee record into a string. You can then print it with `std::cout`.

```
Employee e;
...
std::cout << e.toString() << std::endl;
```

Example Run

Below is a sample run of what your program will look like, when it is complete:

```
$ ./bin/employee-sort input/test.bin ./out.bin
1 Jaimieinabethn 108336.000000
2 Semirlaehnmihn 164250.000000
3 Elenoraynesesn 177905.000000
4 Jazilynnn 154931.000000
5 Savioureighsan 137820.000000
...
```