Day 28 (Mon 04/04)

- exercise 27 review
- day 28 recap questions
- exercise 28

Announcement/reminders

- Submit "Midterm Project Contributions" survey on Gradescope by Friday 4/8 (note: do not assume that late submissions will be accepted for credit)

- HW5: due Wednesday 4/6 by 11pm

- HW6: due Wednesday 4/13 by 11pm
  - written assignment, no late submissions

Exercise 27

Part 2: mean and median member functions

grade_list.cpp

grade_list.h

class GradeList {
⋮
double mean(void);
⋮

```
double GradeList::mean(void) {
  assert(!grades.empty());
  double sum = 0.0;
  for (std::vector<double>::const_iterator i = grades.cbegin();
       i != grades.cend();
       i++) {
    sum += *i;
  }
  return sum / grades.size();
}

double GradeList::median(void) {
  return percentile(50.0);
}
```

Part 3

GradeList gl;

in main2.cpp:

```
  double min_so_far = 100.0;
  for (size_t i = 0; i < gl.grades.size(); i++) {
    if (gl.grades[i] < min_so_far) {
      min_so_far = gl.grades[i];
    }
  }
```

This does not work because grades is a private member of GradeList, so a main function (which is not a member function of GradeList) cannot access it directly.

Part 3:

One possible solution:

In grade_list.h (adding new public member functions):

```
size_t get_num_grades() const { return grades.size(); }
double get_grade(size_t i) const { return grades[i]; }
```

In main2.cpp:

```
double min_so_far = 100.0;
for (size_t i = 0; i < gl.get_num_grades(); i++) {
  if (gl.get_grade(i) < min_so_far) {
    min_so_far = gl.get_grade(i);
  }
}
```

Part 3:

Another possible solution: add to grade_list.h (in GradeList class)

```
const std::vector<double> &get_grades() const { return grades; }
```

The code in main2.cpp can call get_grades() on the GradeList object to access a const reference to the internal grades vector.

References to gl.grades could be changed to gl.get_grades().

This doesn't violate encapsulation because a const reference can't be used to modify the internal data of the GradeList object.

Part 4:

```
#include <iostream>
#include "grade_list.h"

int main() {
  GradeList gl;
  for (int i = 0; i <= 100; i += 2) {
    gl.add(double(i));
  }
  std::cout << "Minimum: " << gl.percentile(0.0) << std::endl;
  std::cout << "Maximum: " << gl.percentile(100.0) << std::endl;
  std::cout << "Median: " << gl.median() << std::endl;
  std::cout << "Mean: " << gl.mean() << std::endl;
  std::cout << "7th percentile: " << gl.percentile(75.0) << std::endl;
}
```

Day 28 recap questions:

1. What is a non-default (or “alternative”) constructor?
2. If we define a non-default constructor, will C++ generate an implicitly defined default constructor?
3. When do we use the this keyword?
4. What is a destructor?
5. A destructor will automatically release memories that are allocated in the constructor- true or false?

1. What is a non-default (or “alternative”) constructor?

A non-default constructor has one or more parameters. Usually, these are used to initialize the field(s) of the object being initialized.

Example:

```
class Point {
private:
  double x, y;

public:
  Point() : x(0.0), y(0.0) { }          // default constructor
  Point(double x_, double y_)     // non-default constructor
    : x(x_), y(y_) { }

  // ... other member functions ...
};
```

2. If we define a non-default constructor, will C++ generate an implicitly defined default constructor?

No. For example:

```
class Point {
private:
  double x, y;
public:
  Point(double x_, double y_)
    : x(x_), y(y_) { }

  // ... other member functions ...
};
```

```
// The following code will not
// compile

Point p;
```

The problem is that Point does not have a default constructor, but the declaration of p requires one.

```
Point p(4.0, 5.0);
```

```
Point() : Point(0.0, 0.0) { }
```

(I think?)

3. When do we use the this keyword?

The this keyword is useful for explicitly referring to the object a member function is called on. It is a pointer to the object the member function is called on.

Among other uses, this can be useful for disambiguating a member variable (field) that has the same name as a parameter. E.g.:

```
class Point {
private:
  double x, y;

public:
  // ...
  void set_x(double x) { this->x = x; }
  // ...
};
```

```
Point & Point:: operator=(const Point &rhs){
    if (this != &rhs) {
        x = rhs.x;
        y = rhs.y;
    }
    return *this;
}
```

4. What is a destructor?

A class (or struct) type's destructor member function is called automatically when an object's lifetime ends. It's purpose is to deallocate any dynamic resources associated with the object.

Examples of dynamic resources:

- dynamically allocated memory

- file resources not automatically closed by a destructor

Example:

```
class CBuf {
private:
  char *buf;
  size_t size;

public:
  CBuf(size_t sz)
    : buf(new char[sz]), size(sz)
  { }

  ~CBuf() {
    delete[] buf;
  }

  // ... other member functions ...
};
```

```
void foo() {
  MyType p;
}  // <- p's destructor called
```

```
MyType *p=
    new MyType;
    ⋮
delete p;
```

5. A destructor will automatically release memories that are allocated in the constructor- true or false?

The destructor must *explicitly* de-allocate dynamically-allocated memory using either delete or delete[] (depending on whether or not the memory being deallocated is an array.)