601.220 Intermediate Programming

Summer 2022, Meeting 16 (July 15th)

Today's agenda

- Review exercise 27
- Day 28 recap questions
- Exercise 28

Reminders/Announcements

- HW6 is due **Friday, April 7th** by 11 pm
  - Written homework, note late submissions

Exercise 27 review

Part 2: `mean` and `median` functions

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

Exercise 27 review

Part 3: in `main2.cpp`:

```
GradeList gl;

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 of
`GradeList`) cannot access it directly.

Exercise 27 review

Part 3: one possible solution is

```
// 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);
    }
  }
```

Exercise 27 review

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

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

In main2.cpp, change gl.grades to gl.get_grades().

Arguably, this doesn't violation encapsulation because a const reference can't be used to modify the internal data of the GradeList object. However, it does result in "leaking" the knowledge that the grades in a GradeList are stored in a std::vector<double>.

Exercise 27 review

```cpp
// Part 4 (main3.cpp)
#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 << "75th 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:

```cpp
class Point {
private:
  double x, y;
public:
  Point(double x, double y)
    : x(x), y(y) { }
  // ... other member functions ...
};

// ...elsewhere...

Point p;  // will not compile
```

3. When do we use the this keyword?

The this keyword us useful for explicitly referring to the object a member function is called on, sometimes called the "receiver" object. It is a *pointer* to the receiver object.

Among other uses, this can be useful for disambiguating a member variable that has the same name as a parameter. Example:

```cpp
class Point {
private:
  double x, y;
public:
  // ...
  void set_x(double x) { this->x = x; }
  // ...
};
```

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 automatcially closed by a destructor

Destructor example

```cpp
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...
};
```

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

False.

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

Exercise 28

- `grade_list` again, but this time storing grades in a dynamically allocated array
  - This is very much like how `std::vector` works!
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes