This is 183

L15: Week 9 - Wednesday

Reminders

- Assignment 4 due Friday!
- You have until next Wednesday (one week) to fill out request for alternate exam
- Project 4 due a week from Friday

Summary

- Classes have **public** and **private** members
- For private members, we need **constructors** to initialize an instance of the class
- We can then use **getters** and **setters** to change that data.
- We can separate a class into:
  - **declarations** (in the .h file) and
  - **definitions** (in the .cpp file)

Default Constructors Summary

- A constructor with no arguments is called a default constructor

- We can have multiple constructors as long as they have different arguments (through function **overloading**)

- An empty default constructor means variables have their default values

Review: Card Class

```
const char DIAMONDS = 'D';
const char CLUBS = 'C';
const char HEARTS = 'H';
const char SPADES = 'S';

class Card {
    public:
        Card();
        Card(char inSuit, int inRank);
    private:
        char suit;
        int rank;
};
```

Zyante Review: Default Constructors

- An empty default constructor means member variables have default values

```
class Card {
    public:
        Card();
        Card(char inSuit, int inRank);
    private:
        char suit;
        int rank;
};
```

```
Card::Card() {
}
```

```
int main() {
    Card c;
}
```

i>Clicker #1

```
class Card {
    public:
        Card();
        Card(char inSuit, int inRank);
    private:
        char suit;
        int rank;
};
```

```
Card::Card() {
}
```

```
int main() {
    Card c;
}
```

What is the rank of the Card c?

A) 1 (Ace)
B) 13 (King)
C) ??? (garbage)
D) None of the above

i>Clicker #1

```
class Card {
    public:
        Card();
        Card(char inSuit, int inRank);
    private:
        char suit;
        int rank;
};
```

```
Card::Card() {
}
```

```
int main() {
    Card c;
}
```

What is the rank of the Card c?

A) 1 (Ace)
B) 13 (King)
C) ??? (garbage)
D) None of the above

i>Clicker #2

Which of these best describes a *default constructor*?

A) A constructor for the Default class
B) A constructor that has the same name as the class
C) A constructor that initializes member variables to default values
D) A constructor that does not take any arguments

i>Clicker #2

Which of these best describes a *default constructor*?

A) A constructor for the Default class
B) A constructor that has the same name as the class
C) A constructor that initializes member variables to default values
D) A constructor that does not take any arguments

i>Clicker #3

Which of these calls a non-default constructor?

```
A) Person helen;
B) Person helen= {"Helen", "Hagos"};
C) Person helen();
D) Person helen("Helen", "Hagos");
```

i>Clicker #6

Another name for the class interface is the?

```
A) private section
B) public section
C) class definition
D) class
```

i>Clicker #6

Another name for the class interface is the?

```
A) private section
B) public section
C) class definition
D) class
```

i>Clicker #7

In a class, if nothing is labeled private/public, the default is?

```
A) private
B) public
```

i>Clicker #7

In a class, if nothing is labeled private/public, the default is?

A) private
B) public

i>Clicker #8

:: is known as?

A) I have no idea
B) an annoyance
C) a typo
D) scope resolution operator

i>Clicker #8

:: is known as?

```
A) I have no idea
B) an annoyance
C) a typo
D) scope resolution operator
```

Person Class

```cpp
class Person {
  public:
    Person();
    void haveBirthday()
    int getAge();
  private:
    int age;
};
```

```cpp
#include "Person.h"
#include <iostream>
using namespace std;

int main() {
  Person me;
  cout << me.getAge();
  me.haveBirthday();
  cout << me.getAge();
}
```

Person Class

```
#include "Person.h"
Person::Person() {
   age = 0;
}

void Person::haveBirthday() {
   age++;
}

int Person::getAge() {
   return age;
}
```

Person Class

```
int main() {
  Person me;
  cout << me.getAge();
  me.haveBirthday();
  cout << me.getAge();

  Person ryan;
  // want to determine who's older
  ...
```

Person Class

```
// want to determine who's older
// if a regular function would do

bool isOlder(Person first, Person second) {
    if (first.getAge() > second.getAge()) {
        return true;
    } else {
        return false;
    }
}
```

Person Class

```
// want to determine who's older
// if a regular function would do

bool isOlder(Person first, Person second) {
    return first.getAge() > second.getAge();
}
```

Better style

Person Class

```
// want to determine who's older
// if a regular function would do

Person me;   // has birthday
Person ryan; // has birthday
if (isOlder(me, ryan)) {
    cout << "me";
} else {
    cout << "Ryan";
}
```

Really would like a member function

Member Function is Better

```
Person me;
Person ryan;
if (isOlder(me, ryan)
    cout << "me";
} else {
    cout << "Ryan";
}
```

```
Person me;
Person ryan;
if (me.isOlder(ryan)) {
    cout << "me";
} else {
    cout << "Ryan";
}
```

i>Clicker #10

What is the *best* declaration for a member function of the Person class that decides who’s older?

```
A) bool isOlder(const Person& p);
B) bool isOlder(Person& p);
C) bool isOlder(Person p);
D) none of the above
```

i>Clicker #10

What is the *best* declaration for a member function of the Person class that decides who's older?

```
A) bool isOlder(const Person& p);
B) bool isOlder(Person& p);
C) bool isOlder(Person p);
D) none of the above
```

```
class FeetInches {
public:
    FeetInches();
    FeetInches( int f, int i );
    int getFeet();
    int getInches();
    void setData(int f, int i);
    FeetInches add(const FeetInches &f);
    void test_simplify();

    friend ostream& operator<< (ostream& outs,
                                const FeetInches& f);
    friend istream& operator >> (istream& ins,
                                 FeetInches& f);
private:
    int feet, inches;
    void simplify();
    void write(ostream& outs);
    void read(istream& ins);
};
```

```
/**
 * Requires: Nothing
 * Modifies: Nothing
 * Effects : Returns a new instance of
 *           FeetInches where feet, inches
 *           are the simplified sum of feet,
 *           inches of the parameter f
 *           and the calling class object.
*/
FeetInches add(const FeetInches &f);
```

```
/**
 * Requires: Nothing
 * Modifies: feet, inches
 * Effects : Simplifies feet, inches to
 *           equal total length, where
 *           inches >=0 and inches < 12
 *           Note: 12 inches equals 1 foot
 *
 * Example:  feet, inches = 5, 14
 *           becomes feet, inches = 6, 2
*/
void simplify();
```