Lecture 21: Intro to Classes

Announcements and reminders

Last time on *Intro Computing…*

We just saw…

- More about how to read some data from a stream!

```
ifstream in_file;
in_file.open("input.txt");
```

  - getline(in_file, line)
  - in_file.get(ch)
  - in_file.unget()

- How to write to a stream!

```
ofstream out_file;
out_file.open("new_output_file_plz.txt");
out_file << var_to_output << " " << value << endl;
```

Object-oriented programming… why?

As programs get large, increasingly difficult to maintain lots of functions and variables

→ different functions need access to different variables

→ becomes soooooooo *tempting* to turn to the Dark Side and use global variables

Recall: global variables are defined outside of any function
→ everyone knows their business

Object-oriented programming… why?

Example: Keep track of characteristics of two players, and a function for them to do battle!

Luke:

```
hit_points_luke = 10;
mana_luke = 100;
attack_strength_luke = 3;
defense_strength_luke = 6;
```

Vader:

```
hit_points_vader = 13;
mana_vader = 80;
attack_strength_vader = 7;
defense_strength_vader = 8;
```

```
int battle(hit_points_luke, mana_luke, attack_strength_luke, defense_strength_luke,
           hit_points_vader, mana_vader, attack_strength_vader, defense_strength_vader)
```

Object-oriented programming… why?

Example: Keep track of characteristics of two players, and a function for them to do battle!

```
int battle(hit_points_luke, mana_luke, attack_strength_luke, defense_strength_luke,
           hit_points_vader, mana_vader, attack_strength_vader, defense_strength_vader)
```

Wouldn't this be simpler?

```
int battle(luke, vader)
```

→ and have all of `luke` and `vader`'s ***attributes*** stored in the `luke` and `vader` variables?

Object-oriented programming… why?

Example: Keep track of characteristics of two players, and a function for them to do battle!

```
int battle(hit_points_luke, mana_luke, attack_strength_luke, defense_strength_luke,
           hit_points_vader, mana_vader, attack_strength_vader, defense_strength_vader)
```

Wouldn't this be simpler?

```
int battle(luke, vader)
```

→ and have all of `luke` and `vader`'s ***attributes*** stored in the `luke` and `vader` variables?

→ ***objects to the rescue!***

Object-oriented programming (OOP)

"A programming style in which tasks are solved by collaborating objects."

… way to use the definition in the name! WTF is an object?

→ **Objects** have their own data associated with them,
and their own functions.

→ luke could be an object

→ **data/attributes:** hit_points, mana, attack_strength, defense_strength…
(data members)

→ **functions:** train(), rest(), push_ups() …
(member functions)

Object-oriented programming (OOP)

"A programming style in which tasks are solved by collaborating objects."

… way to use the definition in the name! WTF is an object?

→ **Objects** have their own data associated with them,
and their own functions.

→ `luke` could be an object

→ **data/attributes:** hit_points, mana, attack_strength, defense_strength…
(data members)

→ **functions:** train(), rest(), push_ups() …
(member functions)

→ and `vader` could be another object (with its own data and functions)

Object-oriented programming (OOP)

→ `luke` could be an object

→ **data/attributes:** hit_points, mana, attack_strength, defense_strength…
(data members)

→ **functions:** train(), rest(), push_ups() …
(member functions)

→ and `vader` could be another object (with its own data and functions)

But here's the thing: both luke and vader have the ***same kinds of data and functions*** associated with them!

→ wouldn't it be nice if there was a **type of variable** with all that info built into it?

Object-oriented programming (OOP)

→ luke could be an object

→ **data/attributes:** hit_points, mana, attack_strength, defense_strength…
(data members)

→ **functions:** train(), rest(), push_ups() …
(member functions)

→ and vader could be another object (with its own data and functions)

But here's the thing: both luke and vader have the ***same kinds of data and functions*** associated with them!

→ wouldn't it be nice if there was a **type of variable** with all that info built into it?

→ there is! We call it a **class**. And we call the variables of that class an **object**.

Object-oriented programming (OOP)

→ wouldn't it be nice if there was a **type of variable** with all that info built into it?

→ there is! We call it a **class**. And we call the variables of that class an **object**.

[define a class of objects called jedi]

```
jedi luke;

luke.train();  // luke trains by running through the
                    swamp with a muppet on his back; increases his
                    attack_strength and defense_strength

luke.rest();   // luke rests and increases his hit_points and mana
```

→ **functions:** train(), rest(), push_ups() … → are part of the class's **public interface**, which are the parts of the object that we can access from the rest of our program

Object-oriented programming (OOP)

Example: What could the data members for a **car** object be?

Object-oriented programming (OOP)

Example: What could the data members for a **car** object be?

- make
- model
- year
- color
- mileage
- fuel level
- electric

Encapsulation and interface

Example: What could the data members for a **car** object be?

- make
- model
- year
- color
- mileage
- fuel level
- electric

The data members are said to be **encapsulated**, because they are hidden from other parts of the program and accessible only through the class's member functions.

→ Hides all the nitty-gritty details so people using the class don't have to worry about it

→ Makes using and modify our classes more manageable

Encapsulation and interface

The data members are said to be **encapsulated**, because they are hidden from other parts of the program and accessible only through the class's member functions.

→ Hides all the nitty-gritty details so people using the class don't have to worry about it

Example: We have used the **string** class, but we didn't have to deal with how `str.substr(6)` works, or what `str[6]` is *actually* doing.

→ We had access to the **public interface** to the string class, and just got to use that

→ Protects the class from us accidentally messing it up

Encapsulation and interface

Example: The interface for a car is similar -- you can successfully (usually…) interact and use a car object without necessarily knowing all the details about how each thingie on the dashboard works.

… because they have a nice ***interface***

Classes

To define a class, we must specify the ***behavior***

… defining the member functions (and what they do)

… and defining the data members (types of variable, size, etc)

Classes

Example: Let's design a class for cash register objects

Need **member functions** to…

- Clear the cash register and start a new sale
- Add the price of an item to a running total
- Get the total amount owed and count the number of items purchased

Classes

Example: Let's design a class for cash register objects

Need **member functions** to…

- Clear the cash register and start a new sale
- Add the price of an item to a running total
- Get the total amount owed and count the number of items purchased

These functions will be our **public interface**

→ specify through a function declaration (prototype) in our class definition

But we also need **data members** too! They will be **private**, only for the member functions

→ running total, number of items, … ?

A tale of two member functions…

There are two types of member functions:

Mutators are member functions that modify the data members

- Increment the item count
- Add price to the total bill
- Clear all data members (reset total bill and item count to 0)

Accessors are member functions that ***query*** a data member(s) of the object, and returns the value(s) to the user

- Get the total bill
- Get the item count

A tale of two member functions…

There are two types of member functions:

Mutators are member functions that modify the data members

- Increment the item count
- Add price to the total bill
- Clear all data members (reset total bill and item count to 0)

SETTERS

Accessors are member functions that ***query*** a data member(s) of the object, and returns the value(s) to the user

- Get the total bill
- Get the item count

GETTERS

Classes -- a generic class interface

```
class NameOfClass
{
public:
    // the public interface
private:
    // the data members /private interface
};
```

Classes -- a generic class interface

```
class NameOfClass
{
public:
    // the public interface
private:
    // the data members /private interface
};
```

Use CamelCase for the names of classes

Any part of our program should be able to call the member functions.
→ they go in the **public interface**

Only member functions (within our class) can access the data members. They're hidden from the rest of the program so we don't screw them up/worry about the guts of our class
→ they go in the **private** section of the class

Classes -- a class interface for our cash register

```
class CashRegister
{
public:
      _____ clear();
      _____ add_item( _____ );
      _____ get_total() const;
      _____ get_count() const;
private:
      // data members will go here
};
```

Classes -- a class interface for our cash register

```
class CashRegister
{
public:
      void clear();
      void add_item( double price );
      double get_total() const;
      int get_count() const;
private:
      // data members will go here
};
```

Question: Which member functions are getters (accessors) and which are setters (mutators)?

Classes -- a class interface for our cash register

```
class CashRegister
{
public:
    void clear();
    void add_item( double price );
    double get_total() const;
    int get_count() const;
private:
    // data members will go here
};
```

setters because they change the value of data members

getters because they simply *report* the values of data members

Question: Which member functions are getters (accessors) and which are setters (mutators)?

Classes -- what's with that 'const' thing?

```
class CashRegister
{
public:
    void clear();
    void add_item( double price );
    double get_total() const;
    int get_count() const;
private:
    // data members will go here
};
```

getters only *report* the values of data members, and never *alter* them

→ we declare these functions to be **const** so they can't mess our stuff up

Classes -- what are our data members?

```
class CashRegister
{
public:
    void clear();
    void add_item( double price );
    double get_total() const;
    int get_count() const;
private:
    _____ ;
    _____ ;
};
```

Classes -- what are our data members?

```
class CashRegister
{
public:
    void clear();
    void add_item( double price );
    double get_total() const;
    int get_count() const;
private:
    int item_count;
    double total_price;
};
```

Always **think carefully** about what the values we might need to access from our class could be!

Common error -- missing semicolon

```
class CashRegister
{
public:
     [public interface goes here...]
private:
     [data members go here...]
}

int main() {
     [events transpire... ]
     return 0;
}
```

Many compilers will report this missing semicolon error as a bug down HERE

Encapsulation

Every `CashRegister` object has its own copy of these data members

```
CashRegister register1;
CashRegister register2;
... [use setter functions] ...
```

register1

```
total_price = 1.95
item_count = 1
```

register2

```
total_price = 17.25
item_count = 5
```

Encapsulation

Every `CashRegister` object has its own copy of these data members

```
CashRegister register1;
CashRegister register2;
... [use setter functions] ...
```

Data members are **private**, so ***this data*** is only accessible via member functions

register1

```
total_price = 1.95
item_count = 1
```

register2

```
total_price = 17.25
item_count = 5
```

Encapsulation

The private data members are only accessible via member functions:

Won't work:

```
CashRegister register1;
... [use setter functions] ...
cout << register1.total_price << endl;
```

Will work!

```
CashRegister register1;
... [use setter functions] ...
cout << register1.get_total() << endl;
```

Encapsulation

The private data members are only accessible via member functions.

Of course, you ***can*** make data members accessible and part of the **public interface**

As a matter of good practice in this class, just don't do it

→ Will keep things tidier and easier to debug. Why is that… ?

→ We can write the **mutator** for `item_count` so it can never be negative

→ On the other hand, if `item_count` were public, we could just straight up set it to be negative.

Encapsulation

The private data members are only accessible via member functions.

Of course, you ***can*** make data members accessible and part of the **public interface**

As a matter of good practice in this class, just don't do it

→ Will keep things tidier and easier to debug. Why is that… ?

Another reason:

We might want to change ***how*** data members are computed and/or manipulated, but the important details (data members) shouldn't necessarily change.

What just happened?!

We just saw…

- What is this **object-oriented programming** you speak of?
- What an ***object*** and a ***class*** is!
- What makes up a class and what is **encapsulation**!
  - Member functions (**public interface**)
  - Data members (private data - encapsulated)
- Different types of member functions
  - Getters (accessors) and setters (mutators)

Next time:

… we get ***class(ier)!***