Object Oriented Programming

Object-oriented programming… why?

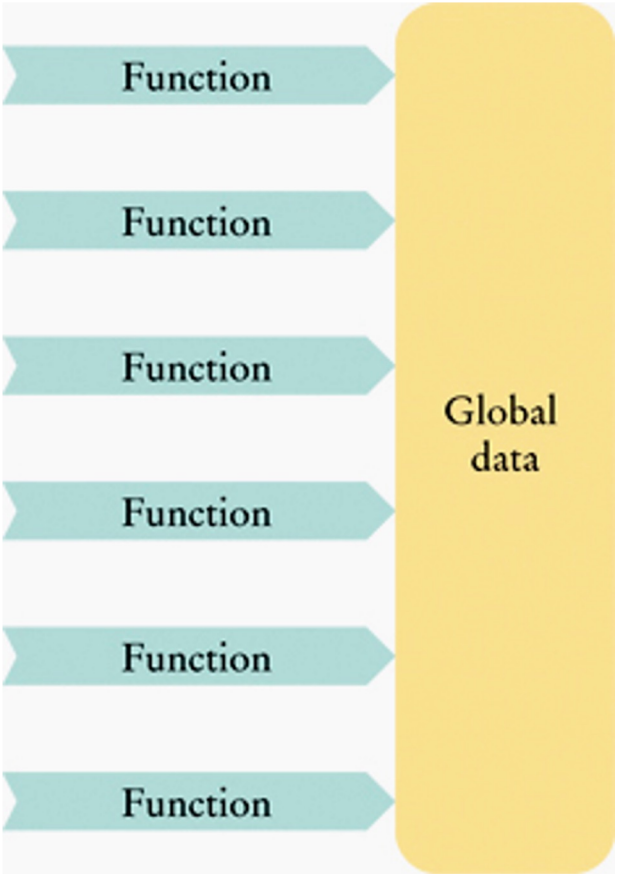
- As programs get large, increasingly difficult to maintain lots of functions and variables
  - different functions need access to different variables
  - becomes soooooooo tempting to turn to the Dark Side and use

GLOBAL VARIABLES

Object-oriented programming… why?

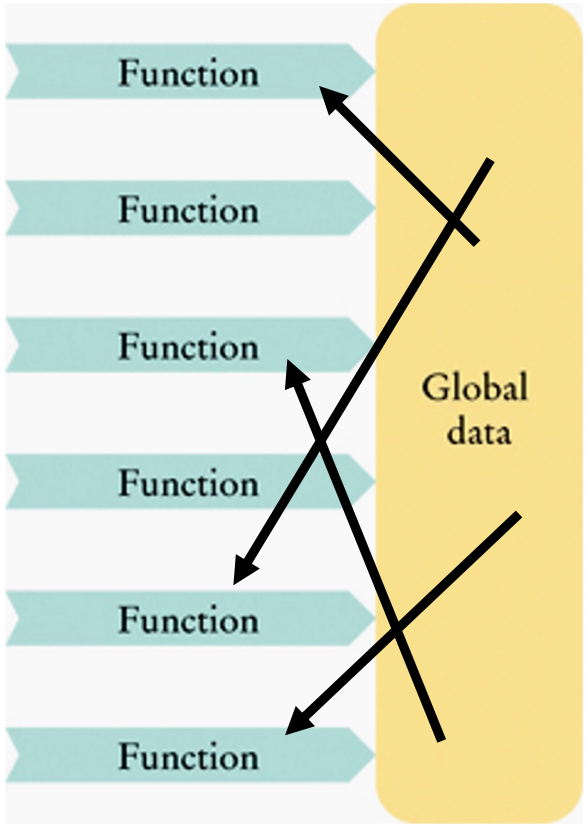
- global variables are defined outside of any function
  - everyone knows their business

```
#include..
int var1 = 12;
double var2 = 20.5;
function1..
function2..
function3..
main()..
```



Object-oriented programming… why?

- When some part of the global data needs to be changed:
  - to improve performance or to add new capabilities
  - a large number of functions may be affected
  - – you will have to rewrite them –
  - and hope everything still works!



Object-oriented programming… why?

- Example: a battle function for two pokemons!
- Pikachu:
  - `hit_points_pikachu = 8;`
  - `speed_pikachu = 82;`
  - `attack_strength_pikachu = 3;`
  - `defense_strength_pikachu = 6;`
- Charmander:
  - `hit_points_charmander = 10;`
  - `speed_charmander = 78;`
  - `attack_charmander = 4;`
  - `defense_charmander = 3;`

```
battle(hit_points_pikachu, speed_pikachu, attack_pikachu,
defense_pikachu, hit_points_charmander, speed_charmander,
attack_charmander, defense_charmander);
```

Object-oriented programming… why?

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

```
battle(hit_points_pikachu, speed_pikachu, attack_pikachu,
defense_pikachu, hit_points_charmander, speed_charmander,
attack_charmander, defense_charmander);
```

- Wouldn't this be simpler?

```
battle(pikachu, charmander)
```

  - and have all of pikachu and charmander's attributes stored in the pikachu and charmander 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! What is an object?

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

No more global variables – *Hurray!*

Objects to the rescue

- The data stored in an object are called:

  data members (attributes/fields)

- The functions that work on data members are:

  member functions

(Instead of “variables” and “functions” – separately)

Example: Objects

- `battle(pikachu, charmander)`

- `pikachu` could be an object
  - **data/attributes:** hit_points, speed, attack_strength, defense_strength…. (data members)
  - **functions:** train(), rest…

    (member functions)

Example: Objects

- `battle(pikachu, charmander)`

- `pikachu` could be an object
  - **data/attributes:** hit_points, speed, attack_strength, defense_strength…. (data members)
  - **functions:** train(), rest…

    (member functions)
- And `charmander` could be another object (with its own data and functions)

Object-oriented programming (OOP)

- both pikachu and charmander 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)

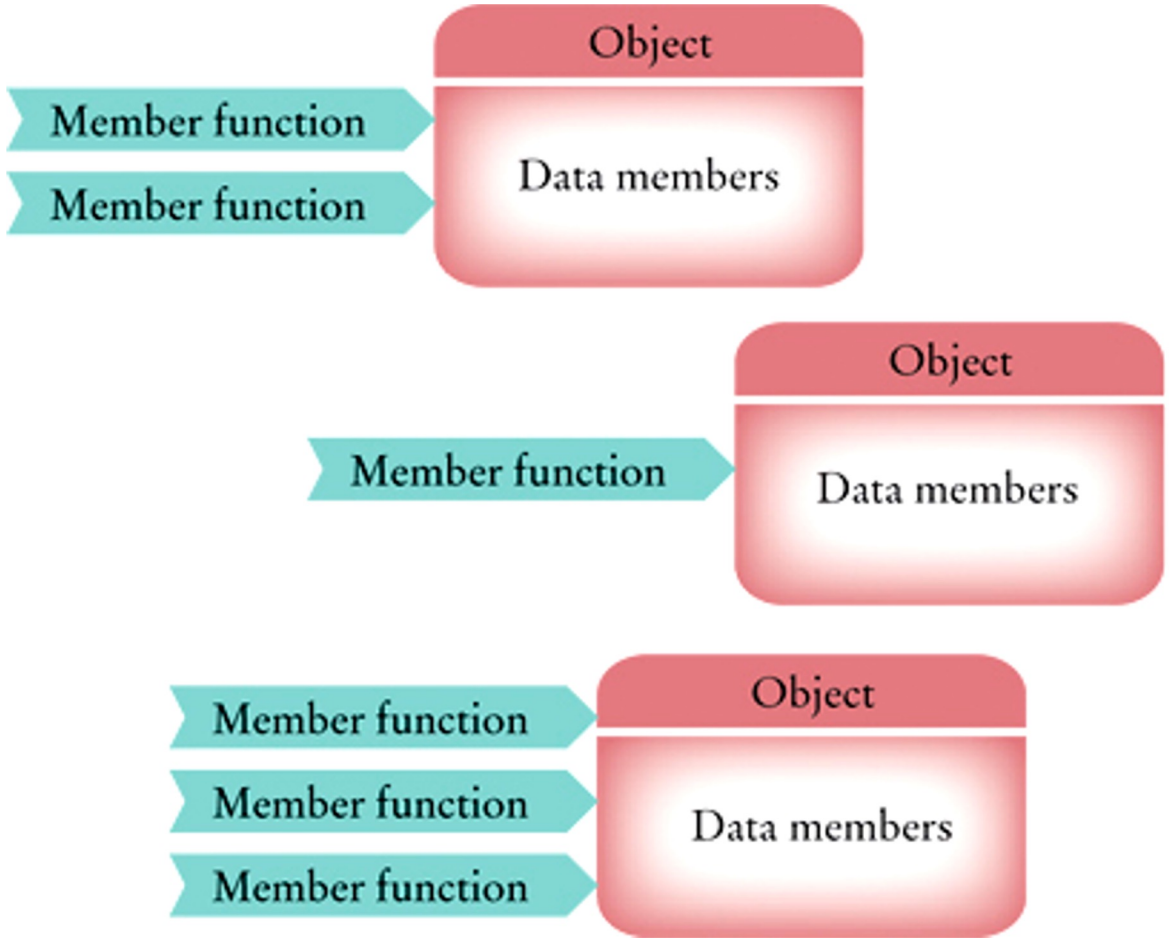
- both pikachu and charmander have the same kinds of data and functions associated with them!
- wouldn’t it be nice if there was a type of variable with all that info built into it?
- there is! We call it a class. And we call the variables of that class an object.

Class

- A class describes a set of objects with the same behavior
- Variables of a class are called objects

- Every class has:
  - Data members
  - Member functions

Objects to the rescue



Classes

- A class describes a set of objects with the same behavior.

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)

Designing a class: `car`

- By observation we would need functionality like:

Member Functions

- Member 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
  - make, model, color … ?

Encapsulation and Interface

- public:
  - accessible outside the class definition
  - member functions

- private:
  - not accessible outside the class definition
  - data members

Encapsulation and Interface

The data members are said to be encapsulated because:

- they are hidden from other parts of the program
- 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

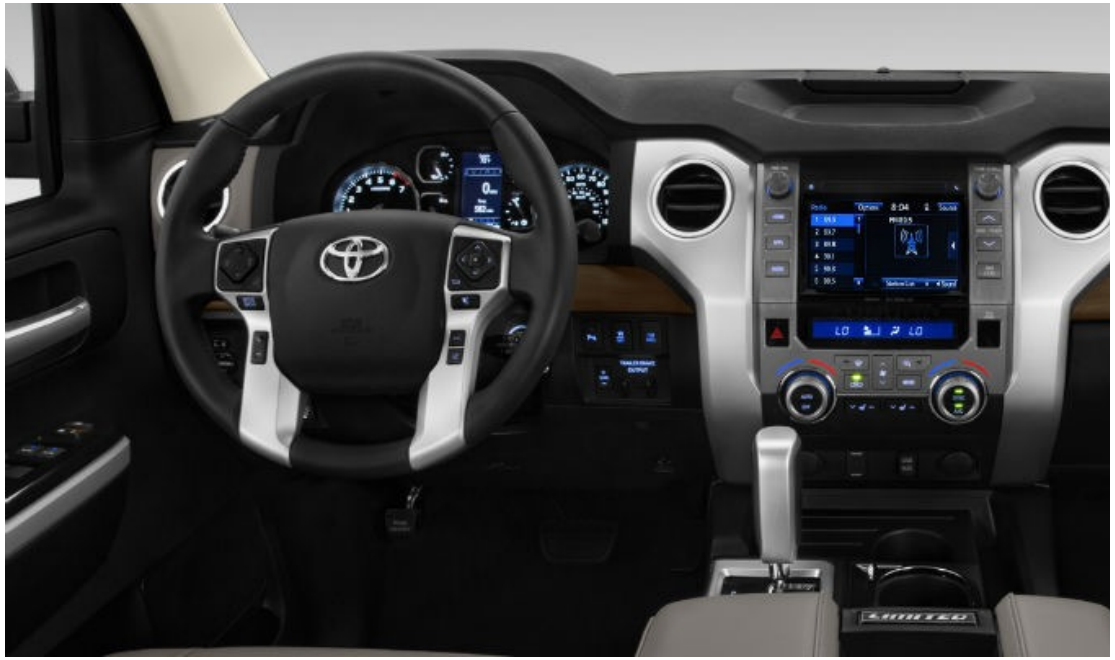
- Now when we want to change the way that an object is implemented, only a small number of functions need to be changed, and they are the ones in the object.

- Because most real-world programs need to be updated often during their lifetime, this is an important advantage of object-oriented programming.

- Program evolution becomes much more manageable.

Encapsulation and Interface

- 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 thing on the dashboard works.
- … because they have a nice interface



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

A generic class interface

```
class NameOfClass
{
  public:
     // the public interface

  private:
     // the data members
};
```

A generic class interface

```
class NameOfClass
{
  public:
     // the public interface

  private:
     // the data members
};
```

Use CamelCase for the names of classes

A generic class interface

```
class NameOfClass
{
  public:
    // the public interface

  private:
    // the data members
};
```

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

A generic class interface

```
class NameOfClass
{
  public:
    // the public interface

  private:
    // the data members
};
```

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

Data members are defined in the *private section* of the class. Only member functions (within our class) can access the data members. They're hidden from the rest of the program
→ they go in the private section of the class

Common Error: Missing Semicolon

```
class CashRegister
{
public:
    [public interface goes here]
private:
    [data members go here]
}
int main()
{
    // Many compilers report that error here in main!
    ...
}
```

Don’t forget the semicolon!

Designing a class: pokemon

```
class Pokemon
{
public:
    // function prototypes
private:
    // data members or class attributes

};
```

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

Designing a class: pokemon

```
class Pokemon
{
public:
   // function prototypes
private:
   string _name;
   int _hit_points;
};
```

Member Functions

Two types:

1. Mutators / setters
2. Accessors / getters

Mutators / Setters

Mutators are member functions that modify the data members

- Increment/decrement health
- Level up or level down

Accessors / Getters

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

- Get the name
- Get current health
- Get current level

Designing a class: pokemon

```
class Pokemon
{
public:
   void setName(string n);
   void setHP(int h);
   string getName() const;
   int getHP() const;
private:
   string _name;
   int _hit_points;
};
```

These are function prototypes.
We'll define them later.

Designing a class: pokemon

```
class Pokemon
{
public:
   void setName(string n);
   void setHP(int h);
   string getName() const;
   int getHP() const;
private:
   string _name;
   int _hit_points;
};
```

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

Designing a class: pokemon

```
class Pokemon
{
public:
   void setName(string n);
   void setHP(int h);
   string getName() const;
   int getHP() const;
private:
   string _name;
   int _hit_points;
};
```

setters because they change the value of data members

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

Designing a class: pokemon

```
class Pokemon
{
public:
   void setName(string n);
   void setHP(int h);
   string getName() const;
   int getHP() const;
private:
   string _name;
   int _hit_points;
};
```

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)?