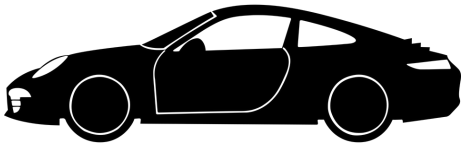
A language agnostic introduction to object-oriented programming (part 1)

Michael R. Nowak
Texas A&M University

1

How would you describe a car?



https://pixabay.com/vectors/automobile-car-drive-porsche-1300239/

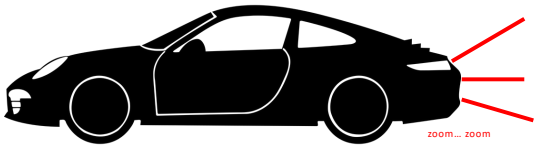
2

What are it's attributes?

https://pixabay.com/vectors/automobile-car-drive-porsche-1300239/

3

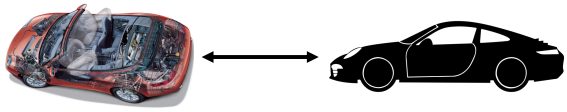
What are it's behaviors?



https://pixabay.com/vectors/automobile-car-drive-porsche-1300239/

4

How would you describe a car?



http://www.seriouswheels.com/2000-2003/2002-Porsche-911-Carrera-Cabriolet-Cutaway-1280x960.htm

https://pixabay.com/vectors/automobile-car-drive-porsche-1300239/

5

Abstraction

- Abstraction | dictates that some information is more important than other information, but does not specify a specific mechanism for handling the unimportant information
  - As a process, denotes the extraction of the essential details about an item, or group of items, while ignoring the inessential details
  - As an entity, denotes a model, view, or some other focused representation for an actual item

6

How would we provide an abstraction of a car in code?

- Types are good for directly representing ideas in code
  - When we want to do
    - Integer arithmetic, `int` is a great help
    - Manipulate text, `std::string` is a great help
  - Types are helpful because they provide
    - **Representation**: A type "knows" how to represent the data needed in an object
    - **Operations**: A type "knows" what operations can be applied to objects

7

How would we provide an abstraction of a car in code?

- The concept of a `car` follows this pattern:
  - A specific `car` is **represented** by attributes
  - We can also perform various **operations** on `car`s, the result of which depends on the state of the object(s) to which it is applied
    - A Ferrari should accelerate faster than a Honda Civic
- We would like to **represent** an **abstraction** of our notion of a `car` as a *user-defined type* along with a *set of functions* that perform `car` **operations**

8

What exactly is a user-defined type?

- A user-defined directly represents a concept in a program
  - If you can think of "it" as a separate entity, it is plausible that it could be a class or an object of a class
  - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
- When we introduce a **user-defined type** to our program, we encapsulate the description of how the **type** is **represented** and the **operations** that can be applied objects of that type
  - As we will see later, a user-defined type provides a **blueprint** from which **objects** are **created**, **used**, and **destroyed**

9

Encapsulation

- Encapsulation |
  - As a process, the act of enclosing one or more items within a (physical or logical) container
  - As an entity, refers to a package or an enclosure that holds (contains, encloses) one or more items
  - In programming languages,
    - Functions, arrays, and structured types (classes, etc.) are common examples of encapsulation mechanisms

10

Representation

- A **user-defined type** provides the description for how **objects** of that **type** are to be **represented**
  - The **representation** of the **user-defined type** is composed of built-in types and other user-defined types that are known together as **data members**
  - An imperfect analogy for this would be an excel spreadsheet:
    - The definition of a table is denoted by the header columns, which provides a description for each field in each row of that column along with its data type; the column headers (metadata) is in a narrow sense like a class
    - Each row has its own storage field for each header column and stores its respectively associated data in that field; each row (data) is in a narrow sense like an object

11

Operations

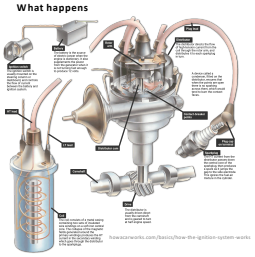
- A **user-defined type** also specifies the **operations** that will be able to be applied its **objects**
  - **Function members** are written to provide the **operations** that we will be able to apply to the **objects** of our **user-defined type**

12

Interact with car through interface

What we do

What happens



13

Information hiding

- Information hiding | accomplished by restricting access to data, functions, types, etc. in order to "hide" implementation details, while providing the user with an interface detailing *what the object does* instead of *how it does it*
- *For example*, we start the car, but the details of how that is accomplished are not of interest to us and can be hidden under the hood
  - *We just turn the key and voilà*

14

User-defined type and information hiding

- What the user interacts with
  - An interface
    - Communicating the set of operations that can be performed
  - The allowable behaviors
    - The way we expect instances of the type to respond to operations
- The implementation can be hidden and consists of
  - An internal representation
  - A set of methods implementing the interface
  - A set of representation invariants, true initially and preserved by all methods

15

E.g., car in simple driving game

- Interface:
  - forward
  - reverse
  - left
  - right
  - accelerate
  - break
- Allowable behaviors:
  - Any position is ok if not crashed, most recent position is used
  - Etc.
- Internal representation:
  - x, y, z (position)
  - s (speed in mph)
  - b (bearing)
  - s_max
- Representation invariant:
  - 0 <= s <= s_speed
  - Etc.
- Methods to implement the interface:
  - …

16

Why user-defined types?

- One of the primary advantages of defining user-defined types is that their instances conduct themselves in nearly the same way as the built-in types
  - The objects instantiated from them follow practically the same rules as the built-in types for naming, scope, lifetime, etc.
  - As a type, they define the operations that can be applied to the objects instantiated from them, as well as context for common operations (such as '+', '–', etc.)

17

Why user-defined types?

- **Encapsulates** data together with the operations that can be performed on that data
- **Data hiding** can be accomplished: restrict interaction with data members across a well-defined public interface; present only the fundamental facilities that the user needs for use, and hide all implementation details
  - Provide the user with the precise interface required to complete the job; keep the public interface to a minimum
  - A change to the implementation should not require a change to the user's code

18

Object-oriented programming

Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance

- Classes can be designed specifically as building blocks for other types, and existing classes can be examined to see if they exhibit similarities that can be exploited in a common class (will get into this later)
- The main focus is on message passing between objects
  - Objects respond to messages by performing some behavior
  - Important to note: each object has its own internal state

19

Using existing classes

- Let's talk about two:
  - std::vector
    - https://en.cppreference.com/w/cpp/container/vector
  - std::string
    - http://www.cplusplus.com/reference/string/string/

20

References

- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.

21