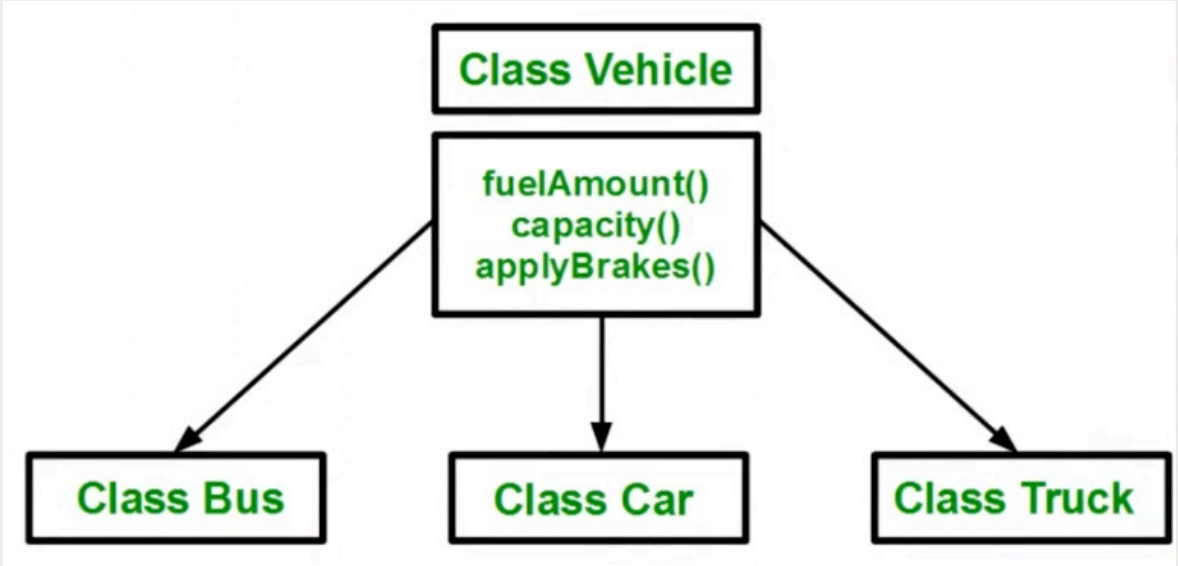
Inheritance

- Class and struct can inherit data and functions from other classes.
- There are three types of inheritance in C++:
  - public
  - protected
  - private
- Public inheritance keeps all access specifiers of the base class.



inheritance_example.cpp inheritance_example.cpp/Vehicle

```cpp
  int capacity_ = 0;  // amount of fuel of the gas tank
  string brand_;      // make of the vehicle

 public:
  Vehicle(int seats, int capacity, string brand)
      : seats_(seats), capacity_(capacity), brand_(std::move(brand)) {}

  int get_seats() const { return seats_; }
  int get_capacity() const { return capacity_; }
  string get_brand() const { return brand_; }

  void Print() const {
    cout << "Vehicle seats available = " << seats_ << endl
         << "Vehicle fuel available = " << capacity_ << endl
         << "Vehicle brand = " << brand_ << endl;
  }
};

////////////////////////////////////////////////////////////////////////////////
class Bus : public Vehicle {
 public:
  Bus(int seats, int capacity, const string& brand, string brakes)
      : Vehicle(seats, capacity, brand), special_brakes_(std::move(brakes)) {}

  string special_brakes() const { return special_brakes_; }

 private:
  string special_brakes_;
};

////////////////////////////////////////////////////////////////////////////////
class Car : public Vehicle {
 public:
  Car(int seats, int capacity, const string& brand, string stereo)
      : Vehicle(seats, capacity, brand), stereo_brand_(std::move(stereo)) {}

```

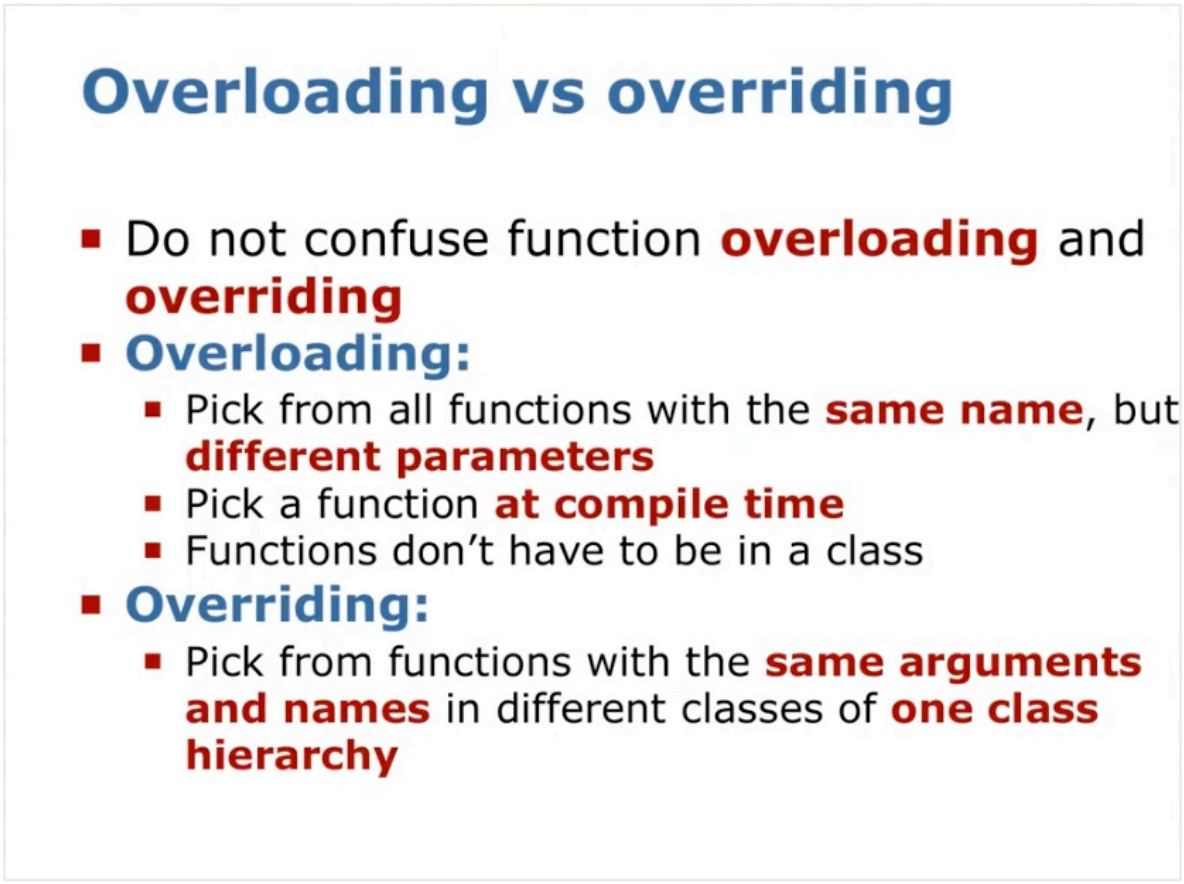
Public Inheritance

- Public Inheritance stands for “is a” relationship, if class Derived inherits publicly from class Base, we say that, Derived is a kind of Base.
- Allows Derived to use all public and protected members of Base.
- Derived still gets its own special functions:
  Constructors, Destructor, Assignment operators

```cpp
#include <iostream>
using std::cout; using std::endl;
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }
 protected:
  int width_ = 0;
  int height_ = 0;
};
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
int main() {
  Square sq(10);  // Short name to save space.
  cout << sq.width() << " " << sq.height() << endl;
  return 0;
}
```

Function Overriding

- A function can be declared virtual
  virtual Func(<params>);
- If function is virtual in base class it can be overridden in Derived class
  Func(<params>) override;
- Base can force all Derived classes to override a function by making it pure virtual
  virtual Func(<params>) = 0;



Abstract classes and Interfaces

- Abstract class: class has at least one pure virtual function.
- Interface: class that has only pure virtual functions and no data members.

How virtual works

- A class with virtual functions has a virtual table.
- When calling a function the class checks which of the virtual function that match the signature should be called.
- Called runtime polymorphism.
- Costs some time but is very convenient.

Using Interfaces

- Use interfaces when you must enforce other classes to implement some functionality.
- Allow thinking about classes in terms of abstract functionality.
- Hide implementation from the caller.
- Allow to easily extend functionality by simply adding a new class.

```cpp
#include <iostream>
using std::cout;
using std::endl;
struct Printable {  // Saving space. Should be a class.
  virtual void Print() const = 0;
};
struct A : public Printable {
  void Print() const override { cout << "A" << endl; }
};
struct B : public Printable {
  void Print() const override { cout << "B" << endl; }
};
void Print(const Printable& var) { var.Print(); }
int main() {
  Print(A());
  Print(B());
  return 0;
}
```