VIRTUAL FUNCTIONS

- A **virtual function** is dynamically bound (i.e., choose which class’ version of the function to call at runtime).

- To make a function virtual, place the `virtual` key word before the return type in the base class's declaration:
  
  ```c++
  virtual double getArea() const;
  ```

- **Why**: For polymorphism to work
  
  ```c++
  Rectangle * varRec = new Box(4,2,1);
  cout << varRec.getArea() << endl;
  ```
**Pure Virtual Functions**

What if you want to write a base class that defines the public interface for a certain hierarchy, but does not implement the virtual functions?

This can be done by making your virtual functions **pure virtual**, or **abstract**:

- `virtual double getArea() const = 0;`

This function has no **body**, and **must** be implemented by the derived classes.
PURE VIRTUAL FUNCTIONS

- But why??
- Consider:

```cpp
class Pet {
    public:
        Pet(string);
        string getName();
        void setName(string);
        virtual void saySomething() = 0;
    protected:
        string name;
};
```
**Pure Virtual Functions**

- Let’s derive a cow:

```cpp
class Cow : public Pet {
    public:
        Cow(string);
        void giveMilk();
        virtual void saySomething();
};

void Cow::saySomething() { // implement!
    cout << "Mooo!" << endl;
}
```
Pure Virtual Functions

Now let’s derive a dog:

class Dog : public Pet {
    public:
        Dog(string);
        void guardProperty();
        virtual void saySomething();
};

void Dog::saySomething() { // implement!
    cout << “Bark! Bark! Bark!” << endl;
}
Pure Virtual Functions

Now:

```c
int main() {

    Pet * cow = new Cow("Belle");
    Pet * dog = new Dog("Spot");

    cow.saySomething(); // Mooo!
    dog.saySomething(); // Bark! Bark! Bark!

    delete cow;
    delete dog;
    return 0;
}
```
**Abstract Classes**

- Now `Pet` provides no implementation of the `saySomething()` function.
- Can we still create a non-polymorphic pet?

```
Pet myPet("Mr Snuffles"); // does this make sense?
```

- **No.** A class containing at least one pure virtual function becomes an **abstract class**

- Abstract classes can not be instantiated, they can only be used with polymorphism

- Abstract classes are used to define a public interface to be shared by the derived classes
**Interface Classes**

- You can write a class that contains only pure virtual functions.
- Such a class is sometimes referred to as an interface class.
- It provides no implementation, but declares the functions that every derived class must implement.

```cpp
class Function { // interface class
    public:
        virtual double computeFunction(double) = 0;
        virtual double computeDerivative(double) = 0;
        virtual double* getRange() = 0;
        virtual double* getDomain() = 0;
};
```
MULTIPLE INHERITANCE

- Up till now, we have only dealt with **single inheritance**
  - Every derived class had exactly one parent
- C++ also allows **multiple inheritance**
  - A derived class can inherit from more than one base class
MULTIPLE INHERITANCE

Why: Because sometimes you want to create a class that combines data and functionality of two unrelated parents

```cpp
class Date {
    public:
        Date(int, int, int);
        int getYear();
        int getMonth();
        int getDay();
    protected:
        int day;
        int month;
        int year;
};

class Time {
    public:
        Time(int, int, int);
        int getHours();
        int getMinutes();
        int getSeconds();
    protected:
        int hour;
        int min;
        int secs;
};
```
MULTIPLE INHERITANCE

Why: Because sometimes you want to create a class that combines data and functionality of two unrelated parents

class DateTime : public Date, public Time {
    public:
        DateTime(int, int, int, int, int, int, int);
        string getDate();
        string getTime();
        void printDateTime();
};

Now every DateTime object will have all data & functions of Date, as well as all data & functions of Time
MUTIPLE INHERITANCE

- Calling both parent class constructors:

  ```cpp
  DateTime::DateTime(int d, int mon, int y,
                     int h, int m, int s)
  : Date(d, mon, y), Time(h, m, s)
  {}
  ```

- Order of execution: same as the order of inheritance

- `class DateTime : public Date, public Time`

- In this case: first Date, then Time
MULTIPLE INHERITANCE

Problem: What if both parent classes define a function with the same name?

class Date {
    public:
        Date(int, int, int);
        int getYear();
        int getMonth();
        int getDay();
        void print();

    protected:
        int day;
        int month;
        int year;
};

class Time {
    public:
        Time(int, int, int);
        int getHours();
        int getMinutes();
        int getSeconds();
        void print();

    protected:
        int hour;
        int min;
        int secs;
};
MULTIPLE INHERITANCE

- Which print() will be invoked here?

```java
int main() {
    DateTime dt(31, 8, 2015, 15, 20, 0);
    // 31 Aug 2015, 15:20:00
    dt.print(); // Which print??
    return 0;
}
```

- This function call is ambiguous and will not compile
- To resolve ambiguity, use scope resolution operator (::)
Multiple Inheritance

- Use scope resolution:

```cpp
int main() {
    DateTime dt(31, 8, 2015, 15, 20, 0);
    // 31 Aug 2015, 15:20:00
    dt.Date::print(); // print the date
    dt.Time::print(); // print the time
    return 0;
}
```

- Now the compiler knows exactly which function to call in each case
MULTIPLE INHERITANCE

Was multiple inheritance really necessary here, though?

class DateTime {
    public:
    DateTime(int, int, int, int, int, int, int);
    string getDate();
    string getTime();
    void printDateTime();

    private:
    Date date;
    Time time;
};

Most of the time, multiple inheritance can be replaced with single inheritance and aggregation. Use multiple inheritance with caution, and only when it truly makes sense.