Reusability

- Reusability means that a class that has been designed, created and debugged once can be distributed to other programmers for use in their own programs.
- Similar to the idea of a library of functions in a procedural language.
- The concept of inheritance provides an important extension to the idea of reusability, as it allows a programmer to take an existing class without modifying it and adding additional features and functionality. This is done by inheriting a new sub-class from the exisiting base class.

Inheritance

- Inheritance is a tool to:
  - implement efficiently a set of related classes (mechanical)
  - organize coherently the concepts in an application domain (conceptual)
  - design for flexibility and extensibility in software systems (logical)
- Inheritance is the process of creating new classes, called *derived classes*, from existing classes, called *base classes*.
- The derived classes inherits all the properties and capabilities of the base class.

Relationships among Objects

- Attribute:
  One object uses as another object as an attribute, namely as member data, for example a Person contains an attribute Name. This type of relation-ship is also called a *weak association* or *has-a* relationship. Example: A Person has a Name
- Association:
  One object uses another to help it carry out a task. Classes that collaborate are usually related through associations. This type of relationship is also called a *uses* relationship.
  Example: The object Driver invokes the method Brake of the object BrakingSystem.

Relationships among Objects

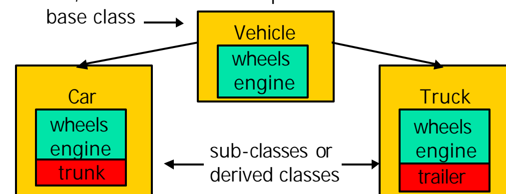
- Aggregation:
  Aggregation means that one object contains other objects. Aggregation is also called part-of relationship. Example: The class Adressbook contains many People Objects.
- Composition:
  Composition is building objects from parts. It is a stronger type of aggregation in which the parts are necessary to the whole, namely they are permanently bound to the object and do not exist outside the object.
  A class Processor contains a CPU, Memory and I/O-Ports.

Relationships among Objects

- Generalization
  Generalization is a relationship defined at the class level not the object level, which means that all objects of this class must obey the relationship. This is type of relationship is also called a *is-a-kind-of* relationship. Generalization in object oriented languages is realized by means of *inheritance*.
  Example: A car is a kind of vehicle.

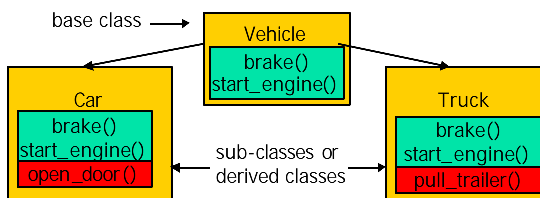
Inheritance

- In our daily lives we use the concept of classes divided into subclasses, for example vehicles are divided into cars, trucks, buses and motor cycles.
- The principle in this sort of division is that each sub-class shares some common features with the base class from which it is derived, but also has its own particular features.



Inheritance

- A sub-class also shares common methods with its super-class but can add its own methods or overwrite the methods of its super-class.



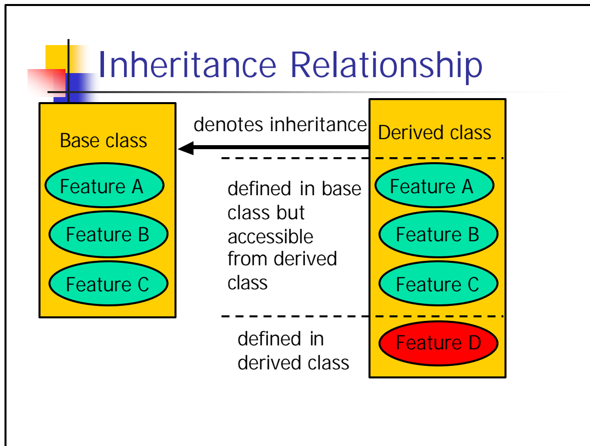
Inheritance

Terminology:

- Car is a *sub-class* (or *derived class*) of Vehicle
- Car *inherits* from Vehicle
- Car is a *specialization* of Vehicle
- Vehicle is a *super-class* (or *base class*) of Car
- Vehicle is a *generalization* of Car

- In C++ an object of a sub-class is substitutable for an object of the super-class, in other words an object of class Car can be used whenever an object of class Vehicle is required.

Inheritance Relationship



Base Class Date

```
class Date                          // declares class name
{
    private:
      int day, month, year;         // member data
    public:                         // member functions
      void Date(int d, int m, int y);  // constructor
      Date operator+(int days);
      Date& operator++();
      int operator-(Date &date);
      bool LeapYear();
      int DaysInMonth();
      void Display();
};
```

Derived Class DateTime

- Suppose we would like to have a class that handles dates with times such as 12.3.1998 17:24, for example for a lecture schedule.
- Since the functionality for date has already been implemented we can reuse it and do not have to rewrite the code again.
- Our new class DateTime inherits all the data members and membership functions of class Date
- It adds the data members and membership functions for time that are missing

Derived Class DateTime

```
class DateTime : public Date  // derived from base class Date
{
    private:
      int hour, minutes;        // additional member data
    public:                     // additional member functions
      void DateTime(int d, int m, int y, int h, int mi);  // constructor
      void SetTime(int h, int m);
      void AddMinutes(int m);
      void AddHours(int h);
      void Display();    // overrides Date::Display()
};
```

Derived Class DateTime

```
Date justdt(12,6,1998);   // constructor Date
DateTime  dt(13,2,1997,20,15);  //constructor DateTime
dt=dt+5;   // operator+() of class Date
if (dt.LeapYear())    // LeapYear() of class Date
   cout << "Date is in a leapyear" << endl;
else
   cout << "Date is not in a leapyear" << endl;
dt.AddHours(2);   // AddHours() of class DateTime
dt.Display();   // Display() of class DateTime
justdt.Display();  // Display() of class Date
```

Derived Class DateTime

```
DateTime::DateTime(int d, int m, int y, int h, int mi)
  : Date(d,m,y) , hours(h) , minutes(mi)
    // invokes the Date Constructor
{
}

void DateTime::Display()
{
    Date::Display();    // first call Display() of class Date
    cout << " " << hours << ":" << minutes;
}
```

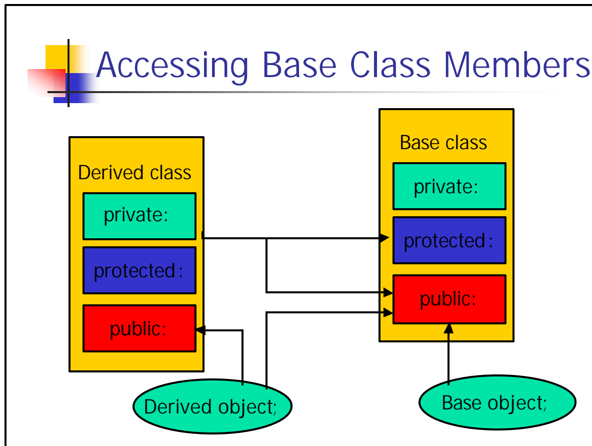
Derived Class DateTime

```
void DateTime::AddHours(int h)
{
   hours+=h;
   if (hours > 23)
   {
      hours-=24;
      *this++;       // add one day  Date operator++
   }
}
```

Accessing Base Class Members

- The access specifier private, protected and public specify which objects are allowed to access members of the base class
- private:
  - This member is only accessible within the base class.
- protected:
  - This member is accessible within the base class and its derived classes.
- public:
  - Every object of the class outside the class definition has access to this member

Accessing Base Class Members



Accessing Base Class Members

Access Specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside class
Public:	Yes	Yes	Yes
Protected:	Yes	Yes	No
Private;	Yes	No	No

Access Specifiers

```
class Date
{
    protected:              // member data can be
     int day, month, year;  // accessed by derived class
     …
};
void DateTime::Display()
{
   cout << day << "/" << month << "/" year << " " << hours <<
   ":" << minutes;    // DateTime can access day, month, year
}
```

Derived Class Constructors

```
void DateTime::DateTime(int d, int m, int y, int h, int m)
  : Date(d,m,y) , hours(h) , minutes(m)
    // invokes the Date() constructor
{
}
DateTime dt(12,4,1999,17,30);
```

- The compiler will create an object of type DateTime and then call the DateTime() constructor to initialize it.
- The DateTime() constructor in turn calls the Date() constructor which initializes the day, month, year
- The DateTime() constructor initializes its own member data hours and minutes

Overloading Member Functions

- You can use member functions in a derived class that override – have the same name- as member functions in the base class.

```
class Date
{
 public:
    voidDisplay();     // defines Date::Display()
};

class DateTime : public Date  // derived from base class Date
{
public:
    voidDisplay();     // overrides Date::Display()
};
```

Overloading Member Functions

- When the same function exists in both the base class and the derived class, the function in the derived class will be executed.

```
DateTime dt(12,3,2001,12,30);
dt.Display();  // Display of derived class DateTime is called
```

- One can use the scope resolution operator to specify exactly what class the function is a member of

```
void DateTime::Display()
{
    Date::Display();    // explicitly calls Display() of class Date
    cout << " " << hours << ":" << minutes;
}
```

Virtual Functions

- Virtual functions allow the re-definition of base class functions in the derived class
- The program decides at run-time not at compile time which version of a function is called (dynamic binding)
- When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class

Static Binding

```
class Animal
{
   protected:
     string str;
   public:
     Animal(const string s) : str(s) {};
     void Display() { cout << "Animal : " << str << endl; }
     // non-virtual function, static binding
};
```

Derived Classes

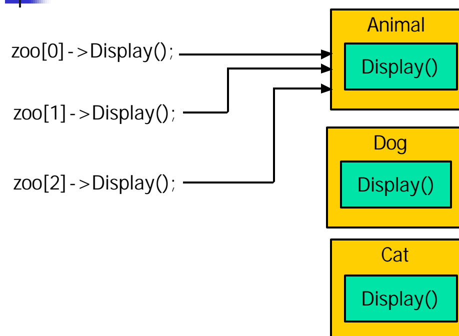
```
class Dog : public Animal
{
    public:
     Dog(const string s) : Animal(s)  {};
      voidDisplay() { cout << "Dog : " << str << endl; }
};
class Cat : public Animal
{
    public:
      Cat(const string s) : Animal(s)  {};
      voidDisplay() { cout << "Cat : " << str << endl ; }
};
```

Static Binding

```
Animal *zoo[3];  // defines an array of pointers to Animal
Animal  animal("Flipper");   // creates an Animal object
Dog  dog("Lassie");       // creates a Dog object
Cat   cat("Jerry");        // creates a Cat object
zoo[0] = &animal;     // pointer zoo[0] points to animal
zoo[1] = &dog;     // pointer zoo[0] points to animal
zoo[2] = &cat;     // pointer zoo[0] points to animal
for (int i=0; i<3; i++)
    zoo[i]->Display();    // static binding to Animal::Display()
```

Output:
Animal : Flipper
Animal : Lassy
Animal : Jerry

Static Binding



Virtual Function

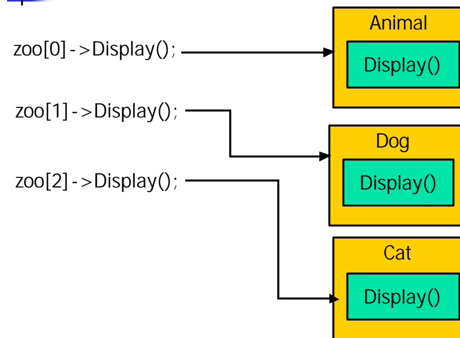
```
class Animal
{
    protected;
      string str;
    public:
      Animal(const string s) : str(s) {};
      virtual void  Display() { cout << "Animal : " << str << endl; }
      // virtual function, dynamic binding
};
```

Dynamic Binding

```
Animal *zoo[3];  // defines an array of pointers to Animal
zoo[0] = new Animal("Flipper");   // creates an Animal object
zoo[1] = new Dog("Lassie");       // creates a Dog object
zoo[2] = new Cat("Jerry");        // creates a Cat object
for (int i=0; i<3; i++)
   zoo[i]->Display();    // dynamic binding to either
       // Animal::Display(), Dog::Display() or Cat::Display()
```

Output:
Animal : Flipper
Dog : Lassy
Cat : Jerry

Dynamic Binding



Dynamic Binding with References

```
void f(Animal& a)
{
   cout << "This is an ";
   a.Display();
}

Animal animal("Flipper");
Dog dog("Lassie");
Cat cat("Jerry");
f(animal);
f(dog);
f(cat);
```

Dynamic Binding

- The compiler defers the decision about which function Display() to invoke until runtime.
- At runtime when it is known what class the pointer or reference is pointing to the appropriate function Display() for the object of the corresponding class is called
- This mechanism is called *late binding* or *dynamic binding*.
- Late binding requires some overhead but provides increased power and flexibility

Lab1 Reading Matrix from File

- Cin and the UNIX redirection operator < can be used to read in the data from the file testdata into the Matrix objects

```
int main()
{
Matrix m1;m2;
cin >> m1;
cin >> m2;
}
```

- On the UNIX prompt you redirect the input to testdata:

```
$ a.out < testdata
```

Lab2 Recursion in Class Definition

- Instead of including menuitem.h into menu.h on just declares the class MenuItem in menu.h without defining it. The same for the class Menu in menuitem.h
- Example of menuitem.h

```
class Menu;   // class declaration of Menu
class MenuItem  // class definition of MenuItem
{
  ...
  void f (Menu &menu);  // refers to declared class Menu
  ...
}
```