MODULE 15
INHERITANCE II

My Training Period: hours

Abilities

Able to understand and use:

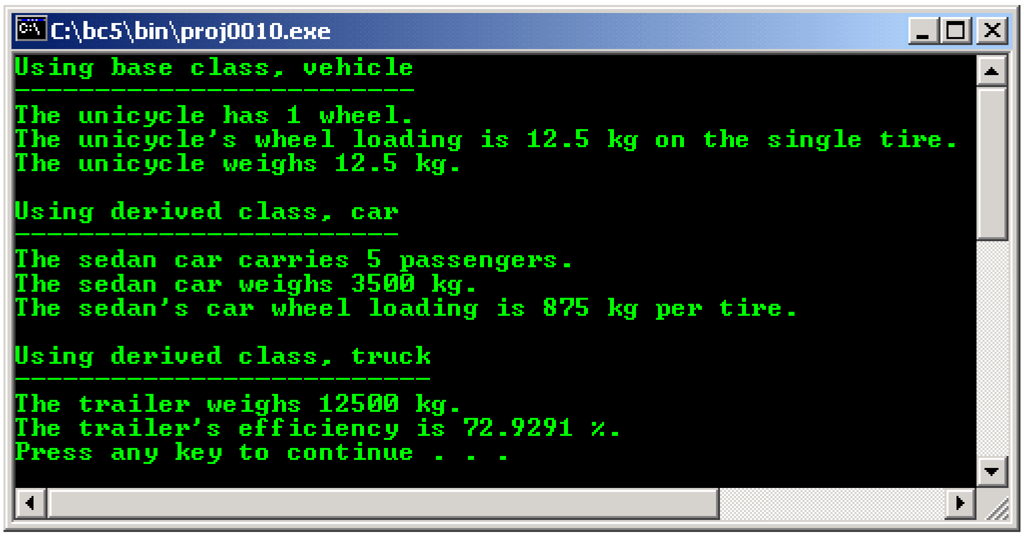
- Inheritance.
- Scope operator (**::**).
- `protected`, `private` and `public` keywords.
- Method vs function.
- Constructor Execution Order.
- Destructor Execution Order.
- Pointer, Array and Objects.
- Friend functions and classes, keyword `friend`.

15.1 Introduction

- This Module will illustrate some of the finer points of inheritance and what it can be used for.
- Examine `inherit1.cpp` program. It is identical to the program developed in Module 14 named `allvehicle.cpp` except the program code is rearranged.

```
//Program inherit1.cpp
#include <iostream.h>
#include <stdlib.h>

//-------class declaration part---------------------
class vehicle
{
  //This variable will be automatically
  //inherited by all the derived class but not outside the
  //base and derived class of vehicle
   protected:
  int    wheels;
  double weight;
 public:
    void  initialize(int input_wheels, double input_weight);
    int  get_wheels(void)        {return wheels;}
    double  get_weight(void)     {return weight;}
    double  wheel_load(void)  {return (weight/wheels);}
};

//--------------derived class declaration part-------------------
class car : public vehicle
{
  int passenger_load;
  public:
   void initialize(int input_wheels, double input_weight, int people = 4);
   int passengers(void)
   {
      return passenger_load;
    }
};

class truck : public vehicle
{
  int     passenger_load;
  double  payload;
  public:
  void    init_truck(int  how_many = 4, double  max_load = 24000.0);
  double  efficiency(void);
  int     passengers(void)        {return  passenger_load;}
};

//------------------------The main program------------------------
int main()
{
   vehicle  unicycle;
   unicycle.initialize(1, 12.5);

   cout<<"Using base class, vehicle\n";
   cout<<"-------------------------\n";
   cout<<"The unicycle has " <<unicycle.get_wheels()<<" wheel.\n";
   cout<<"The unicycle's wheel loading is "<<unicycle.wheel_load()<<" kg on the
                                                             single tire.\n";
   cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

   car  sedan_car;
   sedan_car.initialize(4, 3500.0, 5);

   cout<<"Using derived class, car\n";
   cout<<"------------------------\n";
   cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
   cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
   cout<<"The sedan's car wheel loading is "<<sedan_car.wheel_load()<<" kg per
                                                                   tire.\n\n";
   truck trailer;
   trailer.initialize(18, 12500.0);
   trailer.init_truck(1, 33675.0);

   cout<<"Using derived class, truck\n";
   cout<<"--------------------------\n";
   cout<<"The trailer weighs "<< trailer.get_weight()<< " kg.\n";
   cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";

   system("pause");
   return 0;
}

//------------base and derived class implementation part--------------
//initialize to any data desired, own by base class
void vehicle::initialize(int input_wheels, double input_weight)
{
   wheels = input_wheels;
   weight = input_weight;
}

//initialize() method own by derived car class
void car::initialize(int input_wheels, double input_weight, int people)
{
  //class base variables used by derived car class,
  //because of the protected keyword
    passenger_load = people;
    wheels = input_wheels;
    weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
     passenger_load = how_many;
     payload = max_load;
  }

  double truck::efficiency(void)
  {
     return payload / (payload + weight);
  }
```

106 Lines: Output:



```
Using base class, vehicle
-------------------------
The unicycle has 1 wheel.
The unicycle's wheel loading is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.

Using derived class, car
------------------------
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan's car wheel loading is 875 kg per tire.

Using derived class, truck
--------------------------
The trailer weighs 12500 kg.
The trailer's efficiency is 72.9291 %.
Press any key to continue . . .
```

- The difference is that some of the simpler methods in the classes have been changed to inline code.
- In a practical programming situation, shorter and simple methods should be programmed inline since the actual code just to return a simple value is shorter than the code required to send a message to a non-inline method.
- Other change is the **reordering** of the classes and related methods with the classes all defined first, followed by the main program.
- The implementations for the methods are deferred until the end of the file where they are available for quick reference but are not cluttering up the class definitions.
- This arrangement violates the C++ rules and the use of the separate compilation, but is only done here for convenience.
- The best way to package all of the program examples in this Module are like the class packaging explained in Module 12.
- As mentioned before, the two derived classes, **car** and **truck**, each have a variable named **passenger_load** which is legal because they are defined in different classes.
- The **car** class has a method of the same name, **initialize()**, as one declared in the base class named **vehicle**.

15.2 The Scope Operator

- Notice that the method **initialize()** is declared in the derived **car** class, so, it hides the method of the same name which is part of the base class, and there may be times you wish to send a message to the method in the base class for use in the derived class object.
- This can be done by using the **scope operator** (**::**) in the following manner in the main program:

```
void vehicle::initialize(int input_wheels, double input_weight)
```

- The **number** and **types of parameters** must agree with those of the method in the base class.

15.3 Protected Data

- If the data within a base class were totally available in all classes inheriting that base class, it would be a simple matter for a programmer to inherit the base class into a derived class and have free access to all data in the base class.
- This would completely override the protection achieved by using the information hiding. For this reason, the data in a class should not automatically available to the methods of an inheriting class.
- There are times when you may wish to automatically inherit all variables directly into the derived classes and have them act just as though they were declared as a part of those classes also. For this reason, C++ has provided the keyword **protected**.
- In this program example, the keyword **protected** is given in line 11 so that all of the data of the **vehicle** class can be directly imported into any derived classes but are **not available outside** of the base class or derived classes.

```
...
protected:
      int    wheels;
```

```
                double weight;
        ...
```

- As mentioned before, all data are automatically defaulted to **`private`** at the beginning of a class if no specifier is given.
- You will notice that the variables named **`wheels`** and **`weight`** are available for use in the method named **`initialize()`** in lines `87` through `94` as shown below, just as if they were declared as a part of the **`car`** class itself.

```
void car::initialize(int input_wheels, double input_weight, int people)
{
  //class base variables used by derived car class,
  //because of the protected keyword
    passenger_load = people;
    wheels = input_wheels;
    weight = input_weight;
}
```

- They are available because they were declared **`protected`** in the base class. They would be available here if they had been declared **`public`** in the base class, but then they would be available outside of both classes and we would lose our protection.
- We can now conclude the rules for the three means of defining **variables** and **methods** specifiers.

Specifier	Description
`private`	The variables and methods are not available to any outside calling routines, and they also are not available to any derived classes inheriting this class. Class members are private by default. You can override the default `struct` access with private or protected but you cannot override the default `union` access. `friend` declarations are not affected by this access specifier.
`protected`	The variables and methods are not available to any outside calling routines, but they are directly available to any derived class inheriting this class. You can override the default `struct` access with private or protected but you cannot override the default `union` access. `friend` declarations are not affected by this access specifier.
`public`	All variables and methods are freely available to all outside calling routines and to all derived classes. Members of a struct or union are public by default. You can override the default `struct` access with private or protected but you cannot override the default `union` access. `friend` declarations are not affected by this access specifier.

Table 15.1: Member variable and method specifier

- These keywords when used are effective until one of the other keyword is found in a sequence manner. So, depend on your need, they can be reused in the same block of codes.
- These three definitions can also be used in a **`struct`** data type. The difference with a **`struct`** is that everything defaults to **`public`** until one of the other keywords is used.
- Compile and run this program before continuing on the next program example.

15.4 Private Data And Inheritance

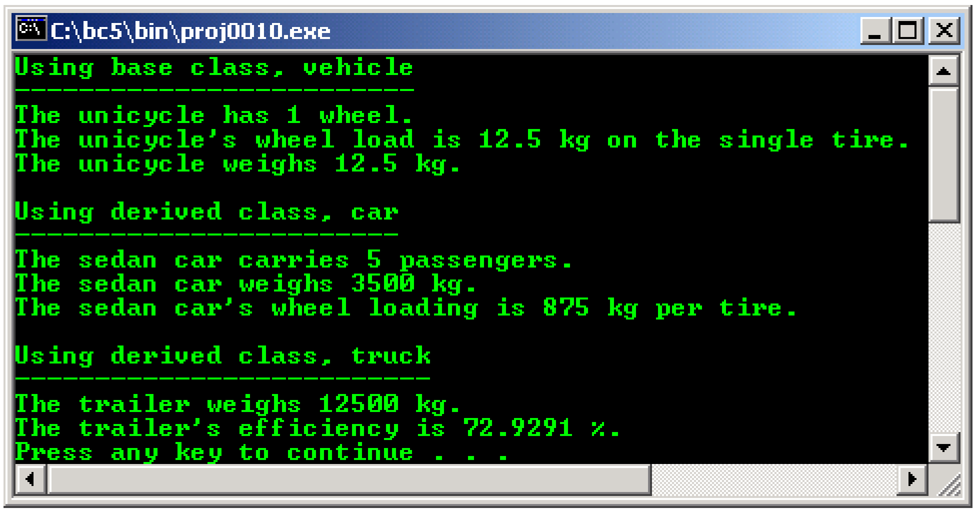
- Examine the program named `inherit2.cpp` where the data in the base class is permitted to use the default **`private`** because line `8` is commented out.

```
1.          //program inherit2.cpp
2.          #include  <iostream.h>
3.          #include  <stdlib.h>
4.
5.          //--------------base and derived class declaration part--------------
6.          class vehicle
7.          {
8.                   //protected:
9.               //Note this is removed, so it is private now
10.          int     wheels;
11.          double  weight;
12.          public:         //public specifier
13.          void    initialize(int input_wheels, double input_weight);
```

```
      int     get_wheels(void)        {return wheels;}
      double  get_weight(void)     {return weight;}
      double  wheel_load(void)  {return (weight/wheels);}
};

class car : public vehicle
{
   int   passenger_load;
   public:
   void  initialize(int input_wheels, double input_weight, int people = 4);
   int    passengers(void)        {return passenger_load;}
};

class truck : public vehicle
{
   int  passenger_load;
   double  payload;
   public:
   void  init_truck(int how_many = 4, double max_load = 24000.0);
   double efficiency(void);
   int passengers(void)        {return  passenger_load;}
};

//----------------main program------------------
int main()
{
   vehicle unicycle;
   unicycle.initialize(1, 12.5);

   cout<<"Using base class, vehicle\n";
   cout<<"-------------------------\n";
   cout<<"The unicycle has "<<unicycle.get_wheels()<<" wheel.\n";
   cout<<"The unicycle's wheel load is "<<unicycle.wheel_load()<<" kg
                                                    on the single tire.\n";
   cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

   car sedan_car;
   sedan_car.initialize(4, 3500.0, 5);

   cout<<"Using derived class, car\n";
   cout<<"-------------------------\n";
   cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
   cout<<"The sedan car weighs "<<sedan_car.get_weight()<< " kg.\n";
   cout<<"The sedan car's wheel loading is "<<sedan_car.wheel_load()<<
                                                        " kg per tire.\n\n";

   truck trailer;
   trailer.initialize(18, 12500.0);
   trailer.init_truck(1, 33675.0);

   cout<<"Using derived class, truck\n";
   cout<<"--------------------------\n";
   cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
   cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";

   system("pause");
   return 0;
}

//--------------base and derived class implementation part-------------
// initialize to any data desired, this method own by base class
void vehicle::initialize(int input_wheels, double input_weight)
{
     wheels = input_wheels;
     weight = input_weight;
}

//This method own by derived class
void car::initialize(int input_wheels, double input_weight, int people)
{
   passenger_load = people;

  //This variable are invalid anymore because both wheels
  //and weight are private now.
  //wheels = input_wheels;
  //weight = input_weight;
   vehicle::initialize(input_wheels, input_weight);
   //Added statement, using base class method instead of derived class
}

```

```
94.      void truck::init_truck(int how_many, double max_load)
95.      {
96.         passenger_load = how_many;
97.         payload = max_load;
98.      }
99.
100.    double truck::efficiency(void)
101.    {
102.      //Changed from program inherit1.cpp, from weight to get_weight()
103.       return payload / (payload + get_weight());
104.    }
```

104 Lines: Output:



```
C:\bc5\bin\proj0010.exe
Using base class, vehicle
-------------------------
The unicycle has 1 wheel.
The unicycle's wheel load is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.

Using derived class, car
-------------------------
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan car's wheel loading is 875 kg per tire.

Using derived class, truck
--------------------------
The trailer weighs 12500 kg.
The trailer's efficiency is 72.9291 %.
Press any key to continue . . .
```

- In this program, the data is not available directly for use in the derived classes, so the only way the data (member variables) in the base class can be used is by sending messages to methods in the base class, within the derived class.
- You should think about how the class you define will be used. If you think somebody should wish to inherit your class into a new class and expand it, you should make the data members **`protected`** so they can be easily used in the new derived class.
- Lines `88` and `89` are invalid now since the members are not visible, so they are commented out as shown below:

```
// wheels = input_wheels;
// weight = input_weight;
```

- But line `90` now does the job they did before they were hidden by calling the public method of the base class as shown below:

```
vehicle::initialize(input_wheels, input_weight);
//Added statement, using base class method instead of derived class
```

- You will notice that the data is still available in lines `77` and `78` as shown below, just as they were before because the member variables are protected in the **`vehicle`** class. Compile and run this program.

```
 wheels = input_wheels;
 weight = input_weight;
```

15.5 Hidden Methods

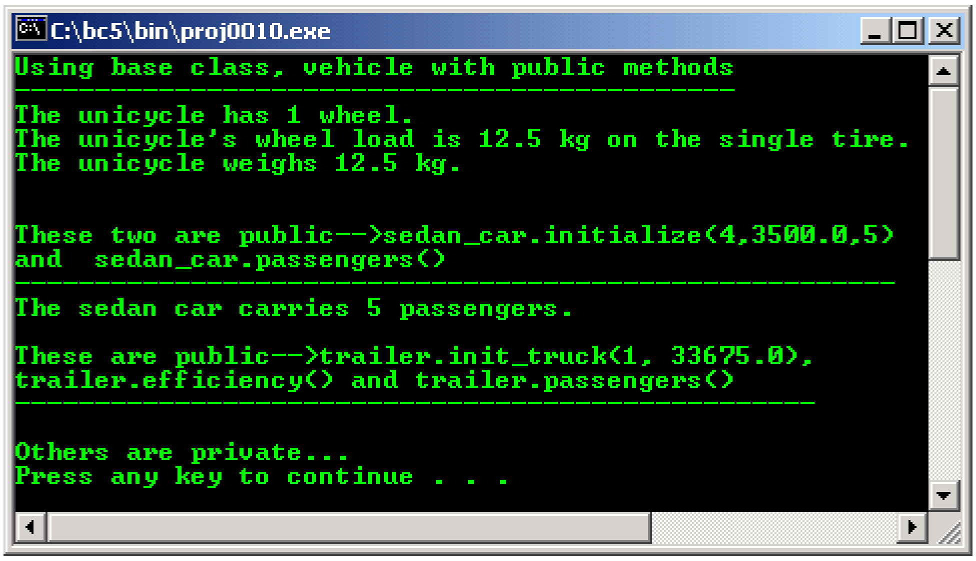
- Examine the program named `inherit3.cpp` carefully and you will see that it is a repeat of the `inherit1.cpp` with a few minor changes.

```
1.     //Program inherit3.cpp
2.     #include  <iostream.h>
3.     #include  <stdlib.h>
4.
5.     //-----------------base and derived class declaration part--------------
6.     class  vehicle
```

```
 {
  protected:
    int  wheels;
   double  weight;
  public:
   void   initialize(int input_wheels, double input_weight);
   int    get_wheels(void)      {return  wheels;}
   double get_weight(void)      {return  weight;}
   double wheel_load(void)      {return  (weight/wheels);}
};

//public keyword changed to private - private inheritance
class car : private vehicle
{
   int   passenger_load;
   public:
   void initialize(int input_wheels, double input_weight, int people = 4);
   int passengers(void)        {return  passenger_load;}
};

//public keyword change to private - private inheritance
class truck : private vehicle
{
   int  passenger_load;
   double payload;
   public:
   void init_truck(int how_many = 4, double max_load = 24000.0);
   double efficiency(void);
   int passengers(void)        {return  passenger_load;}
};

//-----------------------main program-----------------------------
int main()
{
   vehicle unicycle;
   unicycle.initialize(1, 12.5);

   cout<<"Using base class, vehicle with public methods\n";
   cout<<"---------------------------------------------\n";
   cout<<"The unicycle has "<<unicycle.get_wheels()<<" wheel.\n";
   cout<<"The unicycle's wheel load is "<<unicycle.wheel_load()<<" kg
                                                    on the single tire.\n";
   cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

   car sedan_car;
   sedan_car.initialize(4, 3500.0, 5);

   cout<<"\nThese two are public-->sedan_car.initialize(4,3500.0,5)\n";
   cout<<"and  sedan_car.passengers()\n";
   cout<<"-------------------------------------------------------\n";
   cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
   //methods get_weight() and wheel_load() not available
   //because we use private inheritance
   //cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
   //cout<<"The sedan car's wheel loading is "<<sedan_car.wheel_load()<<" kg per
   //                                             tire.\n\n";

  truck trailer;
  //trailer.initialize(18, 12500.0);
  //this method is private now
  trailer.init_truck(1, 33675.0);

  cout<<"\nThese are public-->trailer.init_truck(1, 33675.0),\n";
  cout<<"trailer.efficiency() and trailer.passengers()\n";
  cout<<"--------------------------------------------------\n";
  cout<<"\nOthers are private...\n";
  //methods get_weight() and efficiency() not available
  //because we use private inheritance
  //cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
  //cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";

  system("pause");
  return  0;
}

//------------class implementation part------------------------------
// initialize to any data desired, method own by base class
void vehicle::initialize(int input_wheels, double input_weight)
{
   wheels = input_wheels;
```

```
    weight = input_weight;
}

//method own by derived class
void car::initialize(int input_wheels, double input_weight, int people)
{ //wheels and weight still available because of the protected keyword
    passenger_load = people;
    wheels = input_wheels;
    weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
    passenger_load = how_many;
    payload = max_load;
}

double truck::efficiency(void)
{
     return (payload / (payload + weight));
}
```

107 Lines: Output:



```
Using base class, vehicle with public methods
---------------------------------------------
The unicycle has 1 wheel.
The unicycle's wheel load is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.

These two are public-->sedan_car.initialize(4,3500.0,5)
and  sedan_car.passengers()
-------------------------------------------------------
The sedan car carries 5 passengers.

These are public-->trailer.init_truck(1, 33675.0),
trailer.efficiency() and trailer.passengers()
--------------------------------------------------

Others are private...
Press any key to continue . . .
```

- You will notice that the derived classes named **`car`** and **`truck`** have the keyword **`private`** instead of the **`public`** prior to the name of the base class in the first line of each class declaration as shown below:

```
class car : private vehicle
...

class truck : private vehicle
...
```

- The keyword **`public`**, when included prior to the base class name, makes all of the methods defined in the base class available for use in the derived class at the **same security level** as they were defined in the base class.
- Therefore, in the previous program, we were permitted to call the methods defined as part of the base class from the main program even though we were working with an object of the derived classes.
- In this program, all entities are inherited as **`private`** due to the use of the keyword **`private`** prior to the name of the base class. They are therefore unavailable to any code outside of the derived class.
- The general rule is that all elements are inherited into the derived class at the most restrictive of the **two restrictions** placed on them,

  - The definition in the base class and
  - The restriction on inheritance.

- This defines the way the elements are viewed outside of the derived class.

- The elements are all inherited into the derived class such that they have the same level of protection they had in the base class, as far as their visibility restrictions within the derived class.
- We have returned to use the **protected** instead of **private** in the base class; therefore the member variables are available for use within the derived class only.
- In the this program, the only methods available for objects of the **car** class, are those that are defined as part of the class itself, and therefore we only have the methods named **initialize()** and **passengers()** available for use with objects of class **car** as shown below:

```
public:
   void initialize(int input_wheels, double input_weight, int people = 4);
   int  passengers(void)
        {return  passenger_load;}
```

- When we declare an object of type **car**, it contains three variables. It contains the one defined as part of its class named **passenger_load** and the two that are part of its base class, **wheels** and **weight** as shown below:

```
       ...
       int     wheels;
       double  weight;
        ...

    class car : private vehicle
    {
       int   passenger_load;
        ...
     }
```

- All are available for direct use within its methods because of the use of the keyword **protected** in the base class. The variables are part of an object of class **car** when it is declared and are stored as part of the object.
- You will notice that several of the output statements have been commented out in the main program since they are no longer legal operations.
- Lines 60 through 62 have been commented out as shown below, because the methods named **get_weight()** and **wheel_load()** are not available as members of the **car** class because we are using **private** inheritance.

```
//cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
//cout<<"The sedan car's wheel loading is "<<sedan_car.wheel_load()<<" kg per
//                                              tire.\n\n";
```

- You will notice that **initialize()** is still available but this is own by the **car** class, not the similar method of the same name in the **vehicle** class (base class).

15.6 Data initialization

- Moving on to the use of the **truck** class in the main program, we find that lines 65 and 66 are commented out as shown below, for the same reason as given above,

```
...
//trailer.initialize(18, 12500.0);
//this method is private now
...
```

- But lines 75 and 76 as shown below are commented out for an entirely different reason.

```
//cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
//cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<"%.\n";
 ...
```

- Even though the method named **efficiency()** is available and can be called as a part of the **truck** class, it cannot be used because we have no way to initialize the **wheels** or **weight** of the **truck** object.
- We can get the **weight** of the **truck** object, as we have done in line 87 as shown below,

```
weight = input_weight;
```

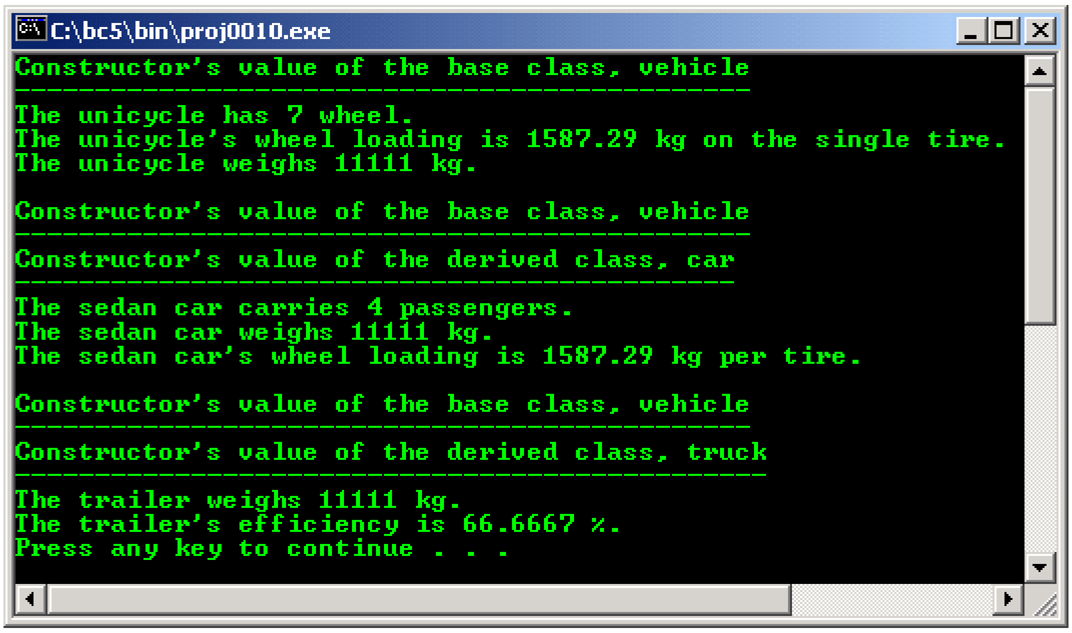
- But since the **`weight`** has no way to be initialized, the result is meaningless and lines `75` and `76` are commented out.
- The **`private`** inheritance is very similar to using an embedded object and, in fact, is rarely used. Until you gain a lot of experience with `C++` and the proper use of Object Oriented Programming, you should use **`public`** inheritance exclusively.
- There are probably not so many reasons to use **`private`** or **`protected`** inheritance. They were probably added to the language for completeness. Compile and run this program example.

15.6.1 Initializing All Data

- Examine the program example named `inherit4.cpp`, you will find that we have fixed the initialization problem that we left dangling in the last program example.

```
//Program inherit4.cpp
#include <iostream.h>
#include <stdlib.h>

//------------base and derived class declaration part-------------
class vehicle
{
   protected:
   int     wheels;
   double  weight;
   public:
   vehicle(void)     {wheels = 7; weight = 11111.0;
   cout<<"Constructor's value of the base class, vehicle"<<'\n';
   cout<<"----------------------------------------------\n";}
   void    initialize(int input_wheels, double input_weight);
   int     get_wheels(void)       {return  wheels;}
   double  get_weight(void)     {return  weight;}
   double  wheel_load(void)  {return  (weight/wheels);}
};

//public inheritance
class car : public vehicle  //public inheritance
{
  int     passenger_load;
  public:
  car(void)       {passenger_load = 4;
  cout<<"Constructor's value of the derived class, car"<<'\n';
  cout<<"----------------------------------------------\n";}
  void    initialize(int input_wheels, double input_weight, int people = 4);
  int     passengers(void)        {return  passenger_load;}
};

class truck : public vehicle         //public inheritance
{
   int      passenger_load;
   double   payload;
   public:
   truck(void)      {passenger_load = 3;payload = 22222.0;
   cout<<"Constructor's value of the derived class, truck"<<'\n';
   cout<<"----------------------------------------------\n";}
   void     init_truck(int how_many = 4, double max_load = 24000.0);
   double   efficiency(void);
   int      passengers(void)        {return passenger_load;}
};

//----------------------------main program---------------------------
int main()
{
   vehicle unicycle;

  //unicycle.initialize(1, 12.5);
  //use default constructor value, so no need the
  //initialization code for object unicycle anymore.
     cout<<"The unicycle has "<<unicycle.get_wheels()<<" wheel.\n";
     cout<<"The unicycle's wheel loading is "<<unicycle.wheel_load()<<" kg
                                                  on the single tire.\n";
     cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

  car sedan_car;
  // use base class initialize() method
  // sedan_car.initialize(4, 3500.0, 5);
  cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
  cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
  cout<<"The sedan car's wheel loading is "<<sedan_car.wheel_load() <<
                                                         " kg per tire.\n\n";

  truck trailer;
  //use base class initialize() method with default data
  //trailer.initialize(18, 12500.0);
  //trailer.init_truck(1, 33675.0);
  cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
  cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";

  system("pause");
  return 0;
}

//----------------class implementation part------------------------
// initialize to any data desired
void vehicle::initialize(int input_wheels, double input_weight)
//base class method
{
   wheels = input_wheels;
   weight = input_weight;
}

void car::initialize(int input_wheels, double input_weight, int people)
//derived class method
{
   passenger_load = people;
   wheels = input_wheels;
   weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
    passenger_load = how_many;
    payload = max_load;
}

double truck::efficiency(void)
{
    return (payload / (payload + weight));
}
```

104 Lines: Output:



```
Constructor's value of the base class, vehicle
----------------------------------------------
The unicycle has 7 wheel.
The unicycle's wheel loading is 1587.29 kg on the single tire.
The unicycle weighs 11111 kg.

Constructor's value of the base class, vehicle
----------------------------------------------
Constructor's value of the derived class, car
---------------------------------------------
The sedan car carries 4 passengers.
The sedan car weighs 11111 kg.
The sedan car's wheel loading is 1587.29 kg per tire.

Constructor's value of the base class, vehicle
----------------------------------------------
Constructor's value of the derived class, truck
-----------------------------------------------
The trailer weighs 11111 kg.
The trailer's efficiency is 66.6667 %.
Press any key to continue . . .
```

- We also added default constructors to each of the classes so we can study how they are used when we use inheritance; and we have returned to the use of **`public`** inheritance.
- When we create an object of the base class **`vehicle`**, there is no problem since inheritance is not a factor. The constructor for the base class operates in exactly the same as all the constructors have in previous Module.

- You will notice that we create the **`unicycle`** object in line `49` as shown below, using the **default constructor** and the object is initialized to the values contained in the constructor.

```
vehicle unicycle;
```

- Line `51` is commented out because we no longer need the initialization code for the object.

```
//unicycle.initialize(1, 12.5);
```

- When we define an object of the derived classes in line `59`, as shown below, it is a little different because not only do we need to call a constructor for the derived class; we have to worry about how we get the base class initialized through its constructor also.

```
car sedan_car;
```

- Actually, it is no problem because the compiler will automatically call the default constructor for the base class unless the derived class explicitly calls another constructor for the base class.
- We will explicitly call another constructor in the next program example, but for now we will only be concerned about the default constructor for the base class that is called automatically.

15.7 Constructor Execution Order

- The next problem we need to be concerned about is the order of constructor execution, and it is easy to remember if you remember the following statement, "**`C++` classes honor their base class by calling their base constructor before they call their own**".
- In the previous program output, you can see that the base class constructor will be called before the derived class constructor. This makes sense because it guarantees that the base class is properly constructed when the constructor for the derived class is executed.
- This allows you to use some of the data from the base class during construction of the derived class.
- In this case, the **`vehicle`** part of the **`sedan_car`** object is constructed, and then the local portions of the **`sedan_car`** object will be constructed, so that all member variables are properly initialized. This is why we can comment out the **`initialize()`** method in line `61`. It is not needed.

```
//sedan_car.initialize(4, 3500.0, 5);
```

- When we define a **`trailer`** object in line `67`, it will also be constructed in the same manner. The constructor for the base class is executed, and then the constructor for the derived class will be executed.

```
truck trailer;
```

- The object is now fully defined and useable with default data in each member. Lines `69` and `70` are therefore not needed and commented out as shown below:

```
//trailer.initialize(18, 12500.0);
//trailer.init_truck(1, 33675.0);
```

15.8 Destructor Execution Order

- As the objects go out of scope, they must have their destructors executed also, and since we didn't define any, the **default destructors** will be executed.
- Once again, the destruction of the base class object named **`unicycle`** is no problem, its destructor is executed and the object is gone.
- The **`sedan_car`** object however, must have two destructors executed to destroy each of its parts, the base class part and the derived class part. The destructors for this object are executed in **reverse order** from the order in which they were constructed.
- In other words, the object is dismantled in the opposite order from the order in which it was assembled. The derived class destructor is executed first, then the base class destructor and the object is removed from the allocation.
- Remember that every time an object is instantiated, every portion of it must have a constructor executed on it. Every object must also have a destructor executed on each of its parts when it is destroyed in order to properly dismantle the object and free up the allocation. Compile and run this program.

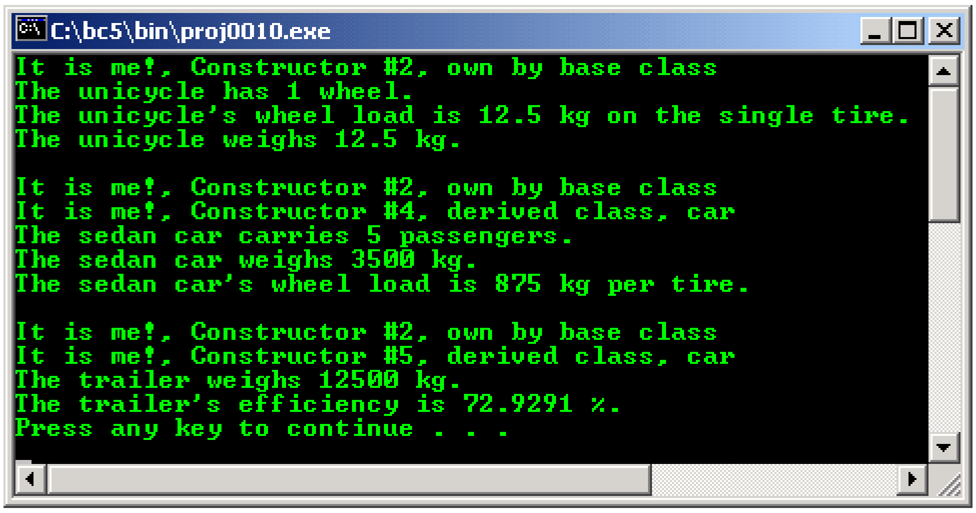
15.9 Inheritance And Constructors

- Examine the program example named `inherit5.cpp` for another variation to our basic program, this time using constructors that are more than just the default constructors.
- You will notice that each class has another constructor declared within it.

```
//Program inherit5.cpp
#include  <iostream.h>
#include  <stdlib.h>

//----------base and derived class declaration part------------
class  vehicle
{
   protected:
   int      wheels;
   double   weight;
   public:
   vehicle(void)
    {
      wheels = 7; weight = 11111.0;
      cout<<"It is me!, Constructor #1, own by base class"<<'\n';
    }

   vehicle(int input_wheels, double input_weight)
    {
      wheels = input_wheels; weight = input_weight;
      cout<<"It is me!, Constructor #2, own by base class"<<'\n';
    }

   void initialize(int input_wheels, double input_weight);
   int get_wheels(void)        {return wheels;}
   double get_weight(void)     {return weight;}
   double wheel_load(void)     {return (weight/wheels);}
};

class car : public vehicle
{
   int passenger_load;
   public:
   car(void)
    {
    passenger_load = 4; cout<<"It is me!, Constructor #3, derived class,
                                                        car"<<'\n';
    }

   car(int people, int input_wheels, double input_weight):vehicle(input_wheels,
                                       input_weight), passenger_load(people)
   {
   cout<<"It is me!, Constructor #4, derived class, car"<<'\n';
   }

  void initialize(int input_wheels, double input_weight, int people = 4);
  int passengers(void)        {return passenger_load;}
};

class truck : public vehicle
{
   int    passenger_load;
   double   payload;
   public:
   truck(void)
    {
       passenger_load = 3;
       payload = 22222.0;
    }
   //the following code should be in one line....
   truck(int people, double load, int input_wheels, double
                                input_weight):vehicle(input_wheels,
                                   input_weight),passenger_load(people),
                                       payload(load)
       {

        cout<<"It is me!, Constructor #5, derived class, car"<<'\n';
     }
   void    init_truck(int  how_many = 4, double  max_load = 24000.0);
   double  efficiency(void);
   int     passengers(void)        {return  passenger_load;}
};

```

```
//----------------------------main program--------------------------
int  main()
{
   vehicle  unicycle(1, 12.5);

  // unicycle.initialize(1, 12.5);
  cout<<"The unicycle has "<<unicycle.get_wheels()<<" wheel.\n";
  cout<<"The unicycle's wheel load is "<<unicycle.wheel_load()<<
                                              " kg on the single tire.\n";
  cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

  //Constructor in the car class called to construct an object,
  //after base class constructor called
  car  sedan_car(5, 4, 3500.0);

  //constructor in the car class called to construct object
  //sedan_car.initialize(4, 3500.0, 5);
  cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
  cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
  cout<<"The sedan car's wheel load is "<<sedan_car.wheel_load()<<
                                                     " kg per tire.\n\n";

  //Constructor in the base class called to construct an object
  truck  trailer(1, 33675.0, 18, 12500.0);

     //trailer.initialize(18, 12500.0);
  //trailer.init_truck(1, 33675.0);
  cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
  cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";

  system("pause");
  return  0;
}

//----------base and derived class implementation part------
// initialize to any data desired
void vehicle::initialize(int input_wheels, double input_weight)
{
   wheels = input_wheels;
   weight = input_weight;
}

void car::initialize(int input_wheels, double input_weight, int people)
{
   passenger_load = people;
   wheels = input_wheels;
   weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
    passenger_load = how_many;
    payload = max_load;
}

double truck::efficiency(void)
{
    return (payload / (payload + weight));
}
```

132 Lines: Output:



```
It is me!, Constructor #2, own by base class
The unicycle has 1 wheel.
The unicycle's wheel load is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.

It is me!, Constructor #2, own by base class
It is me!, Constructor #4, derived class, car
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan car's wheel load is 875 kg per tire.

It is me!, Constructor #2, own by base class
It is me!, Constructor #5, derived class, car
The trailer weighs 12500 kg.
The trailer's efficiency is 72.9291 %.
Press any key to continue . . .
```

- The additional constructor added to the **vehicle** class in lines 12 through 22 as shown below is nothing special; it is just like some of the constructors we have studied earlier.

```
...
vehicle(void)
{
   wheels = 7; weight = 11111.0;
   cout<<"It is me!, Constructor #1, own by base class"<<'\n';
}

vehicle(int input_wheels, double input_weight)
{
   wheels = input_wheels; weight = input_weight;
   cout<<"It is me!, Constructor #2, own by base class"<<'\n';
}
 ...
```

- It is used in line 77 of the main program as shown below, where we define **unicycle** with two values passed in to be used when executing this constructor.

```
vehicle    unicycle(1, 12.5);
```

- The constructor for the **car** class which is declared in lines 34 through 44 as shown below is a little bit different, because we pass in three values. One of the values, named **people,** is used within the derived class itself to initialize the member variable named **passenger_load**.

```
 ...
 car(void)
 {
   passenger_load = 4; cout<<"It is me!, Constructor #3, derived class,
car"<<'\n';
 }

 car(int people, int input_wheels, double input_weight):vehicle(input_wheels,
input_weight), passenger_load(people)
 {
   cout<<"It is me!, Constructor #4, derived class, car"<<'\n';
 }
...
```

- The other two literal values however, must be passed to the base class somehow in order to initialize the number of **wheels** and the **weight**.
- This is done by using a **member initializer**, and is illustrated in this constructor. The colon near the end of line 40 as shown below indicates that a member initializer list follows, and all entities between the colon and the opening brace of the constructor body are member initializers.

```
car(int people, int input_wheels, double input_weight):vehicle(input_wheels,
                                          input_weight), passenger_load(people)
```

- The first member initializer as shown below and looks like a constructor call to the **`vehicle`** class that requires two input parameters.

```
...vehicle(input_wheels, input_weight)...
```

- That is exactly what it is, and it calls the constructor for the **`vehicle`** class and initializes that part of the **`sedan_car`** object that is inherited from the **`vehicle`** class. We can therefore control which base class initializer gets called when we construct an object of the derived class.
- The next member initializer, as shown below acts like a constructor for a simple variable. By mentioning the name of the variable, `passenger_load` and including a value, `people` of the correct type within the parentheses, that value is assigned to that variable even though the variable is not a class, but a simple predefined type.

```
...passenger_load(people)
```

- This technique can be used to initialize all members of the derived class or any portion of them. When all of the members of the member initializer list are executed, the code within the braces is executed.
- In this case, there is no code within the executable block of the constructor. The code within the braces would be written in a normal manner for the constructor.

15.10 Execution Order

- This may seem to be very strange, but the elements of the member initializer list are not executed in the order in which they appear in the list. The constructors for the inherited classes are executed first, in the order of their declaration in the class header.
- When using **multiple inheritance**, (will be discussed in next Module) several classes can be listed in the header line, but in this program, only one is used.
- The member variables are then initialized, but not in the order as given in the list, but in the order in which they are declared in the class. Finally, the code within the constructor block is executed, if there is any code in the block.
- The destructors must be executed in reverse order from the construction order, but if there are two constructors with different construction order defined, which should define the destruction order? The correct answer is neither. The system uses the **declaration order** for construction order and reverses it for the destruction order.
- You will notice that the **`truck`** class uses one initializer for the base class constructor and two member initializers, one to initialize the **`passenger_load`,** and another one to initialize the **`payload`**. The body of the constructor, much like the **`car`** class, is almost empty. This should be put in one line or in contiguous when you do the compiling.

```
truck(int people, double load, int input_wheels, double
                                   input_weight):vehicle(input_wheels,
                                      input_weight), passenger_load(people),
                                          payload(load)
```

- The two constructors in the **`car`** class and the **`truck`** class are called to construct objects in lines `87` and `97` as shown below for a **`car`** and a **`truck`** object respectively as illustrations in this program example.

```
...
car    sedan_car(5, 4, 3500.0);
...
truck  trailer(1, 33675.0, 18, 12500.0);
...
```

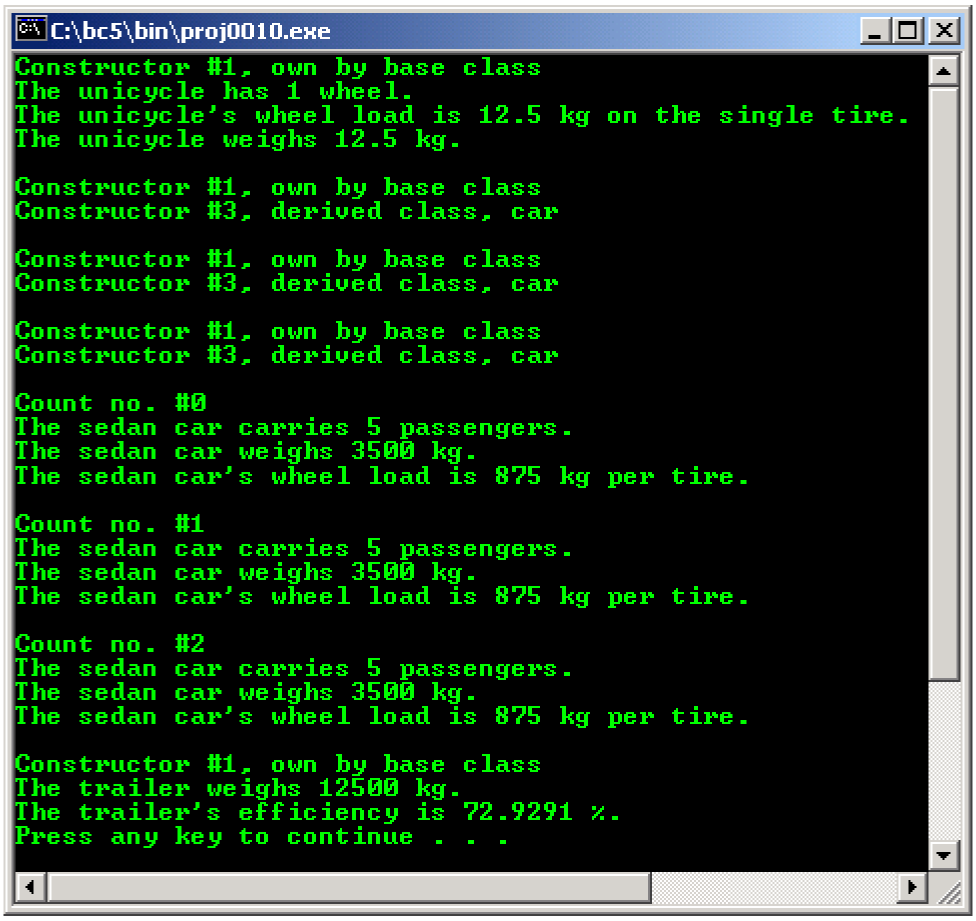
15.11 Pointer, Array And Objects

- Examine the program example named `inherit6.cpp` for examples of the use of an array of objects and a pointer to an object.
- In this program, the objects are instantiated from an inherited class and the purpose of this program is to illustrate that there is nothing special about a derived class. A class acts the same whether it is a base class or a derived class.

```
//Program inherit6.cpp
#include  <iostream.h>
```

```

#include  <stdlib.h>

//------------base and derived class declaration part------------
class  vehicle
{
  protected:
  int      wheels;
  double   weight;
  public:
  vehicle(void)
  {    wheels = 7; weight = 11111.0;
       cout<<"Constructor #1, own by base class"<<'\n';}
  vehicle(int input_wheels, double input_weight)
  {    wheels = input_wheels; weight = input_weight;
       cout<<"Constructor #2, own by base class"<<'\n';}

  void initialize(int input_wheels, double input_weight);
  int get_wheels(void)        {return wheels;}
  double get_weight(void)     {return weight;}
  double wheel_load(void)     {return (weight/wheels);}
};

class car : public vehicle
{
   int passenger_load;
   public:
   car(void)
   {passenger_load = 4; cout<<"Constructor #3, derived class, car"<<"\n\n";}

   car(int people, int input_wheels, double input_weight):vehicle(input_wheels,
                                           input_weight),passenger_load(people)
    {cout<<"Constructor #4 derived class, car"<<'\n'; }

   void initialize(int input_wheels, double input_weight, int people = 4);
   int passengers(void)        {return passenger_load;}
};

class truck : public vehicle
{
   int   passenger_load;
   double   payload;
   public:
   truck(void)
   {passenger_load = 3;
    payload = 22222.0;}

   truck(int people, double load, int input_wheels, double
                                    input_weight):vehicle(input_wheels,
                                      input_weight),passenger_load(people),
                                            payload(load)
          {        }
   void   init_truck(int  how_many = 4, double  max_load = 24000.0);
   double efficiency(void);
   int    passengers(void)        {return  passenger_load;}
};

//---------------------------main program-------------------------
int  main()
{
  vehicle    unicycle;

  unicycle.initialize(1, 12.5);

  cout<<"The unicycle has " <<unicycle.get_wheels()<<" wheel.\n";
  cout<<"The unicycle's wheel load is "<<unicycle.wheel_load()<<
                                                  " kg on the single tire.\n";
  cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

  car sedan_car[3];
  //an array of object with 3 elements
  int index;
  //variable used for counter
  for (index = 0 ; index < 3 ; index++)
  //count and execute
   {
     sedan_car[index].initialize(4, 3500.0, 5);
     cout<<"Count no. #" <<index<<'\n';
     cout<<"The sedan car carries "<<sedan_car[index].passengers()<<"
                                                                passengers.\n";
     cout<<"The sedan car weighs "<<sedan_car[index].get_weight()<<" kg.\n";
     cout<<"The sedan car's wheel load is "<<sedan_car[index].wheel_load()<<
                                                          " kg per tire.\n\n";
   }

  truck  *trailer;   //pointer

  trailer = new truck;
  //initialize to point to something...point to an object

   if (trailer == NULL)
     {
         cout<<"Memory allocation failed\n";
         exit(EXIT_FAILURE);
     }
      trailer->initialize(18, 12500.0);
      trailer->init_truck(1, 33675.0);
  cout<<"The trailer weighs " << trailer->get_weight()<<" kg.\n";
  cout<<"The trailer's efficiency is "<<100.0 * trailer->efficiency()<<
                                       " %.\n";

  delete trailer;
  //de-allocate the object


  system("pause");
  return  0;
 }

//---------base and derived class implementation part----------
// initialize to any data desired
void vehicle::initialize(int input_wheels, double input_weight)
{
   wheels = input_wheels;
   weight = input_weight;
}

void car::initialize(int input_wheels, double input_weight, int people)
{
    passenger_load = people;
    wheels = input_wheels;
    weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
    passenger_load = how_many;
    payload = max_load;
}

double truck::efficiency(void)
{
    return (payload / (payload + weight));
}
```

136 Lines: Output:



```
Constructor #1, own by base class
The unicycle has 1 wheel.
The unicycle's wheel load is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.

Constructor #1, own by base class
Constructor #3, derived class, car

Constructor #1, own by base class
Constructor #3, derived class, car

Constructor #1, own by base class
Constructor #3, derived class, car

Count no. #0
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan car's wheel load is 875 kg per tire.

Count no. #1
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan car's wheel load is 875 kg per tire.

Count no. #2
The sedan car carries 5 passengers.
The sedan car weighs 3500 kg.
The sedan car's wheel load is 875 kg per tire.

Constructor #1, own by base class
The trailer weighs 12500 kg.
The trailer's efficiency is 72.9291 %.
Press any key to continue . . .
```

- This program is identical to the previous program until we get to the **`main()`** program where we find an array of 3 objects of class **`car`** declared in line `72` as shown below:

```
car  sedan_car[3];
```

- It should be obvious that any operation that is legal for a simple object is also legal for an object that is part of an array, but we must tell the system which object of the array we are interested in by adding the array subscript as we do in lines `79, 81, 83` and `84` as shown below:

```
     sedan_car[index].initialize(4, 3500.0, 5);
     cout<<"Count no. #" <<index<<'\n';
     cout<<"The sedan car carries "<<sedan_car[index].passengers()<<" passengers.\n";
     cout<<"The sedan car weighs "<<sedan_car[index].get_weight()<<" kg.\n";
     cout<<"The sedan car's wheel load is "<<sedan_car[index].wheel_load()<<" kg per 
tire.\n\n";
```

- You will notice, in line `88`, we do not declare an object of type **`truck`** but a pointer to an object of type **`truck`**. In order to use the pointer, we must give it something to point at which we do in line `90` by dynamically allocating an object size by using `new` keyword as shown below:

```
truck  *trailer;   //pointer

trailer = new truck;
//initialize to point to something...point to an object
```

- Once the pointer has an object to point to, we can use the object in the same way we would use any object, but we must use the pointer notation to access any of the methods of the object. This is illustrated for you in lines `98` through `101` as shown below:

```
trailer->initialize(18, 12500.0);
trailer->init_truck(1, 33675.0);
cout<<"The trailer weighs " << trailer->get_weight()<<" kg.\n";
cout<<"The trailer's efficiency is "<<100.0 * trailer->efficiency()<<" %.\n";
```

- Finally, we de-allocate the object in line `104` as shown below.

```
delete   trailer;
//de-allocate the object from memory
```

- Compile and run this program.

15.12 Friend Functions and class

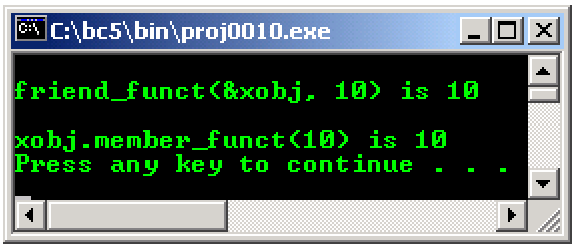
- A function outside of a class can be defined to be a friend function by the class which gives the friend function free access to the private or protected members of the class.
- This is done by preceding the function prototype in the class declaration with keyword `friend`. For example:

```
private:
friend void set(int new_length, int new_width);
//friend method
friend int get_area(void) {return (length * width);}
//friend method
```

- So, `set()` and `get_area()` functions still can be used to access members of the class.
- This in effect, opens a small hole in the protective shield of the class, so it should be used very carefully.
- A single isolated function can be declared as a **friend**, as well as members of other classes, and even entire classes can be given friend status if needed in a program. Neither a constructor nor a destructor can be a friend function.
- Friendship is granted, so for class Y to be a friend of class X, class X must declare that class Y is its friend.
- Class friendship is not transitive: X friend of Y and Y friend of Z does not imply X friend of Z also is not inherited.
- By using `friend`, you can see that it has weakened the data hiding. You should implement this only when there is no way to solve your programming problem.
- Simple program example:

```
//Using friend function...
#include <iostream.h>
#include <stdlib.h>

class SampleFriend
{
   //private member variable
   int i;
   friend int friend_funct(SampleFriend *, int);
   //friend_funct is not private,
   //even though it's declared in the private section
   public:
   //constructor
   SampleFriend(void) { i = 0;};
   int member_funct(int);
};

//implementation part, both functions access private int i
int friend_funct(SampleFriend *xptr, int a)
{
  return xptr->i = a;
}

int SampleFriend::member_funct(int a)
{
    return i = a;
}

main()
{
    SampleFriend xobj;
    //note the difference in function calls

    cout<<"\nfriend_funct(&xobj, 10) is "<<friend_funct(&xobj, 10)<<"\n\n";
    cout<<"xobj.member_funct(10) is "<<xobj.member_funct(10)<<endl;
    system("pause");
}
```

Output:



```
friend_funct(&xobj, 10) is 10

xobj.member_funct(10) is 10
Press any key to continue . . .
```

- You can make all or part of the functions of class, let say, Y, into friends of class X. For example,

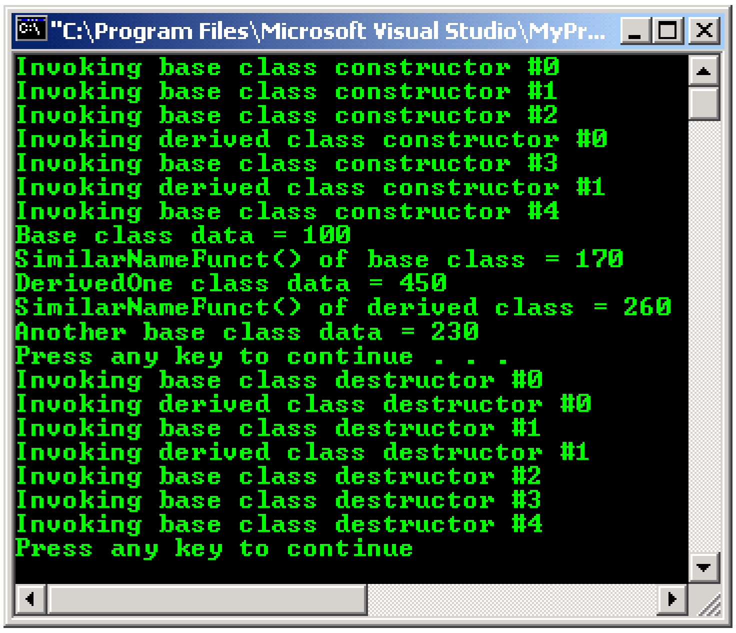
```
class One
{
   friend Two;
   int i;
   void member_funcOne();
};

class Two;
{
   void friend_One1(One&);
   void friend_One2(One*);
  ...
  ...
  ...
};
```

- The functions declared in `Two` are friends of `One`, although they have no friend specifiers. They can access the private members of `One`, such as `i` and `member_funcOne`.

Program Examples and Experiments

```
//inheritance again...
//notice the sequence of the constructor
//and destructor, and private, public,
//protected usage...
#include <iostream.h>
#include <stdlib.h>

class Base
{
      //available for this class member functions ONLY...
     private:
      int BaseVar;
      int NewX;
      int ExtraBaseVar;

      //available to this and derived classes...
      protected:
      int BaseVarOne;

      //available to the derived and outside classes...
      public:
      Base();
      Base(int NewX);
      ~Base();
      public:
      int SetBaseData();
      int ShowBaseData(){return BaseVar;}
      int SimilarNameFunct();
};

class DerivedOne:public Base
{
      //available to this class member functions ONLY...
     private:
      int DerivedOneVar;
      int ExtraDerivedVar;

     //available to the derived and outside classes...
     public:
      DerivedOne();
      ~DerivedOne();
     //available to the derived and outside classes...
     public:
      void SetDerivedOneData();
      int ShowDerivedOneData()
      {
           //BaseVarOne is base class protected member
           //variable, available to this derived class...
           return (DerivedOneVar + BaseVarOne);
      }
      int SimilarNameFunct();
};

//base class constructor...
Base::Base()
{
  BaseVar = 100;
  //constructor counter...
  static int p;
  cout<<"Invoking base class constructor #"<<p<<endl;
  p++;
}

//another base class constructor...
Base::Base(int)
{
  //constructor counter...
  static int t;
  cout<<"Invoking 2nd base class constructor #"<<t<<endl;
  t++;
  BaseVar = NewX;
}

//base class member function...
int Base::SetBaseData()
{ return NewX = 230;}

int Base::SimilarNameFunct()
{return ExtraBaseVar = 170;}

//base class destructor...
Base::~Base()
{
  //destructor counter...
  static int q;
  cout<<"Invoking base class destructor #"<<q<<endl;
  q++;
}

//derived class constructor...
DerivedOne::DerivedOne()
{
  DerivedOneVar = 200;
  //this member variable is inherited from protected base class
  BaseVarOne = 250;

  //constructor counter...
  static int r;
  cout<<"Invoking derived class constructor #"<<r<<endl;
  r++;
}

//derived class destructor...
DerivedOne::~DerivedOne()
{
  //destructor counter...
  static int s;
  cout<<"Invoking derived class destructor #"<<s<<endl;
  s++;
}

void DerivedOne::SetDerivedOneData()
{}

//same member function name as base class
//it is valid since they are from different class
int DerivedOne::SimilarNameFunct()
{return ExtraDerivedVar = 260;}

void main()
{
      //instantiate objects with class types...
      Base ObjOne, ObjFour;
      DerivedOne ObjTwo, ObjFive;

      Base ObjThree;

      cout<<"Base class data = "<<ObjOne.ShowBaseData()<<endl;
      cout<<"SimilarNameFunct() of base class = "<<ObjFour.SimilarNameFunct()<<endl;
     cout<<"DerivedOne class data = "<<ObjTwo.ShowDerivedOneData()<<endl;
      cout<<"SimilarNameFunct() of derived class = "<<ObjFive.SimilarNameFunct()<<endl;
      cout<<"Another base class data = "<<ObjThree.SetBaseData()<<endl;

      system("pause");
}
```

Output:



```
Invoking base class constructor #0
Invoking base class constructor #1
Invoking base class constructor #2
Invoking derived class constructor #0
Invoking base class constructor #3
Invoking derived class constructor #1
Invoking base class constructor #4
Base class data = 100
SimilarNameFunct() of base class = 170
DerivedOne class data = 450
SimilarNameFunct() of derived class = 260
Another base class data = 230
Press any key to continue . . .
Invoking base class destructor #0
Invoking derived class destructor #0
Invoking base class destructor #1
Invoking derived class destructor #1
Invoking base class destructor #2
Invoking base class destructor #3
Invoking base class destructor #4
Press any key to continue
```

- If you cannot see the full output in Borland, you have to use its debugger (Turbo Debugger). From the output snapshot screen you can see that the destructors were invoked in the reverse order of the constructors.
- From the output screen also we can conclude that the processes involved for objects are:

  1. Allocating storage for the objects instantiation.
  2. Processing the objects such as storing, assigning, displaying the objects data etc.
  3. Finally destroy all the allocation and objects.

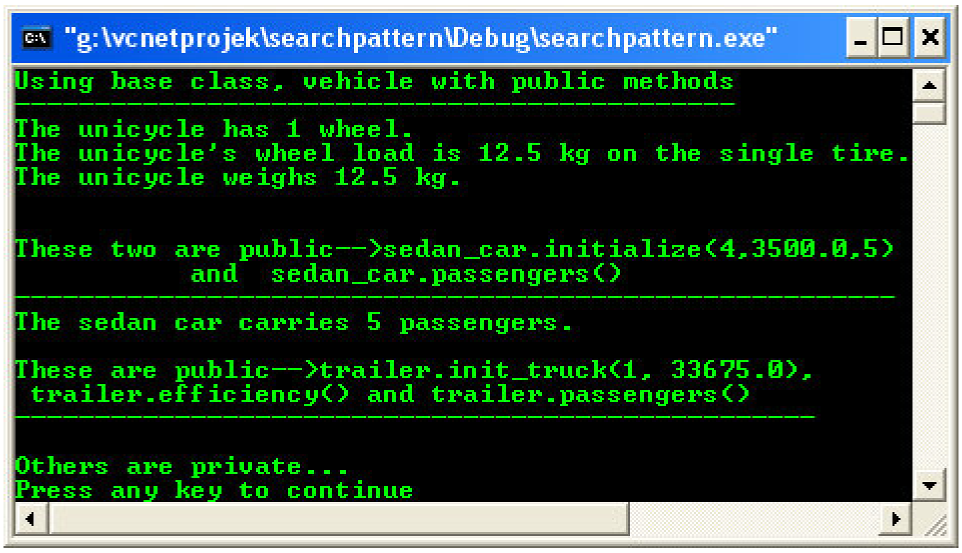
- Very nice huh!!!
- Program example compiled using VC++/VC++ .Net.

```
//Program inherit3.cpp
#include  <iostream>
using namespace std;

//--------base and derived class declaration part--------------
class vehicle
{
  protected:
  int wheels;
  double weight;

  public:
   void   initialize(int input_wheels, double input_weight);
   int    get_wheels(void)      {return  wheels;}
   double get_weight(void)      {return  weight;}
   double wheel_load(void)      {return  (weight/wheels);}
};

//public keyword changed to private - private inheritance
class car : private vehicle
```

```
{
   int   passenger_load;

   public:
   void initialize(int input_wheels, double input_weight, int people = 4);
   int passengers(void)        {return  passenger_load;}
};

//public keyword change to private - private inheritance
class truck : private vehicle
{
   int  passenger_load;
   double payload;
   public:
   void init_truck(int how_many = 4, double max_load = 24000.0);
   double efficiency(void);
   int passengers(void)        {return  passenger_load;}
};

//--------------main program-------------------
int main()
{
   vehicle unicycle;
   unicycle.initialize(1, 12.5);

   cout<<"Using base class, vehicle with public methods\n";
   cout<<"---------------------------------------------\n";
   cout<<"The unicycle has "<<unicycle.get_wheels()<<" wheel.\n";
   cout<<"The unicycle's wheel load is "<<unicycle.wheel_load()<<" kg on the single
                                                         tire.\n";
   cout<<"The unicycle weighs "<<unicycle.get_weight()<<" kg.\n\n";

   car sedan_car;
   sedan_car.initialize(4, 3500.0, 5);

   cout<<"\nThese two are public-->sedan_car.initialize(4,3500.0,5)\n";
   cout<<"           and  sedan_car.passengers()\n";
   cout<<"-----------------------------------------------------\n";
   cout<<"The sedan car carries "<<sedan_car.passengers()<<" passengers.\n";
   //methods get_weight() and wheel_load() not available
   //because we use private inheritance
   //cout<<"The sedan car weighs "<<sedan_car.get_weight()<<" kg.\n";
   //cout<<"The sedan car's wheel loading is "<<sedan_car.wheel_load()<<" kg per
   //                                              tire.\n\n";

   truck trailer;
  //trailer.initialize(18, 12500.0);
  //this method is private now
  trailer.init_truck(1, 33675.0);

  cout<<"\nThese are public-->trailer.init_truck(1, 33675.0),\n";
  cout<<" trailer.efficiency() and trailer.passengers()\n";
  cout<<"--------------------------------------------------\n";
  cout<<"\nOthers are private...\n";
  //methods get_weight() and efficiency() not available
  //because we use private inheritance
  //cout<<"The trailer weighs "<<trailer.get_weight()<<" kg.\n";
  //cout<<"The trailer's efficiency is "<<100.0 * trailer.efficiency()<<" %.\n";
  return 0;
}

//------------class implementation part-------------------------------
// initialize to any data desired, method own by base class
void vehicle::initialize(int input_wheels, double input_weight)
{
   wheels = input_wheels;
   weight = input_weight;
}

//method own by derived class
void car::initialize(int input_wheels, double input_weight, int people)
{  //wheels and weight still available because of the protected keyword
   passenger_load = people;
   wheels = input_wheels;
   weight = input_weight;
}

void truck::init_truck(int how_many, double max_load)
{
   passenger_load = how_many;
```

```
    payload = max_load;
}

double truck::efficiency(void)
{
    return (payload / (payload + weight));
}
```

Output:



```
"g:\vcnetprojek\searchpattern\Debug\searchpattern.exe"
Using base class, vehicle with public methods
---------------------------------------------
The unicycle has 1 wheel.
The unicycle's wheel load is 12.5 kg on the single tire.
The unicycle weighs 12.5 kg.


These two are public-->sedan_car.initialize(4,3500.0,5)
              and  sedan_car.passengers()
-------------------------------------------------------
The sedan car carries 5 passengers.

These are public-->trailer.init_truck(1, 33675.0),
 trailer.efficiency() and trailer.passengers()
--------------------------------------------------

Others are private...
Press any key to continue
```

- Program example compiled using **g++**.

```
/////-herit.cpp-//////////////////////
//notice the sequence of the constructor
//and destructor, and the use of private,
//public and protected. The inheritance...
//////-FEDORA 3, g++ x.x.x-////////////
#include <iostream>
using namespace std;

class Base
{
      //available for this class member functions ONLY...
      private:
      int BaseVar;
      int NewX;
      int ExtraBaseVar;

      //available to this and derived classes...
      protected:
      int BaseVarOne;

      //available to the derived and outside classes...
      public:
      Base();
      Base(int NewX);
      ~Base();
      public:
      int SetBaseData();
      int ShowBaseData(){return BaseVar;}
      int SimilarNameFunct();
};

class DerivedOne:public Base
{
      //available to this class member functions ONLY...
      private:
      int DerivedOneVar;
      int ExtraDerivedVar;

      //available to the derived and outside classes...
      public:
      DerivedOne();
      ~DerivedOne();
      //available to the derived and outside classes...
      public:
      void SetDerivedOneData();
      int ShowDerivedOneData()
      {
        //BaseVarOne is base class protected member
        //variable, available to this derived class...
        return (DerivedOneVar + BaseVarOne);
      }
      int SimilarNameFunct();
};

//base class constructor...
Base::Base()
{
        BaseVar = 100;
        //constructor counter...
        static int p;
        cout<<"Invoking base class constructor #"<<p<<endl;
        p++;
}

//another base class constructor...
Base::Base(int)
{
        //constructor counter...
        static int t;
        cout<<"Invoking 2nd base class constructor #"<<t<<endl;
        t++;
        BaseVar = NewX;
}

//base class member function...
int Base::SetBaseData()
{return NewX = 230;}

int Base::SimilarNameFunct()
{return ExtraBaseVar = 170;}

//base class destructor...
Base::~Base()
{
        //destructor counter...
        static int q;
        cout<<"Invoking base class destructor #"<<q<<endl;
        q++;
}

//derived class constructor...
DerivedOne::DerivedOne()
{
        DerivedOneVar = 200;
        //this member variable is inherited from protected base class
        BaseVarOne = 250;

        //constructor counter...
        static int r;
        cout<<"Invoking derived class constructor #"<<r<<endl;
        r++;
}

//derived class destructor...
DerivedOne::~DerivedOne()
{
        //destructor counter...
        static int s;
        cout<<"Invoking derived class destructor #"<<s<<endl;
        s++;
}

void DerivedOne::SetDerivedOneData()
{}

//same member function name as base class
//it is valid since they are from different class
int DerivedOne::SimilarNameFunct()
{return ExtraDerivedVar = 260;}

int main()
{
      //instantiate objects with class types...
      Base ObjOne, ObjFour;
      DerivedOne ObjTwo, ObjFive;

      Base ObjThree;

      cout<<"Base class data = "<<ObjOne.ShowBaseData()<<endl;
      cout<<"SimilarNameFunct() of base class = "<<ObjFour.SimilarNameFunct()<<endl;
      cout<<"DerivedOne class data = "<<ObjTwo.ShowDerivedOneData()<<endl;
      cout<<"SimilarNameFunct() of derived class = "<<ObjFive.SimilarNameFunct()<<endl;
      cout<<"Another base class data = "<<ObjThree.SetBaseData()<<endl;
      return 0;
}
```

```
[bodo@bakawali ~]$ g++ herit.cpp -o herit
[bodo@bakawali ~]$ ./herit
```

```
Invoking base class constructor #0
Invoking base class constructor #1
Invoking base class constructor #2
Invoking derived class constructor #0
Invoking base class constructor #3
Invoking derived class constructor #1
Invoking base class constructor #4
Base class data = 100
SimilarNameFunct() of base class = 170
DerivedOne class data = 450
SimilarNameFunct() of derived class = 260
Another base class data = 230
Invoking base class destructor #0
Invoking derived class destructor #0
Invoking base class destructor #1
Invoking derived class destructor #1
Invoking base class destructor #2
Invoking base class destructor #3
Invoking base class destructor #4
```

--o0o--

Further reading and digging:

1. Check the best selling C/C++ and object oriented books at Amazon.com.