

MODULE 16 MULTI INHERITANCE

My Training Period: hours

Abilities

Able to understand and use:

- Multiple inheritances.
- Duplicated methods issue.
- Duplicated member variables issue.
- Parameterized type - Function template.
- Parameterized type - Class template.

16.1 Introduction

- Multiple inheritances are the ability to inherit member variable and methods from more than one base class into a derived class.
- This feature will enhance the functionalities of the new class and its reusability but we have to deal with few problems regarding the multi inheritance later.
- The **biggest problem** with multiple inheritances involves the inheritance of variables or methods with **duplicated names** from two or more base classes. The question is: which variables or methods should be chosen as the inherited entities? This will be illustrated in the following program examples.
- Actually, nothing new trick to solve this problem. It is same solution as used in the previous program examples by using the scope operator (::) for duplicated functions name.

16.2 First Multiple Inheritance

- Study the following program named `mulinher1.cpp`, it will reveal the definition of two very simple classes in lines 7 through 32 named `moving_van` and `driver`.

```
1. //program mulinher1.cpp
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----class declaration and implementation part-----
6. //-----base class number one-----
7. class moving_van
8. {
9.     protected:
10.    float  payload;
11.    float  gross_weight;
12.    float  mpg;
13.    //this variable only available for derived class
14.    //because of the protected keyword
15.    public:
16.    void  initialize(float pl, float gw, float input_mpg)
17.    {
18.        payload = pl;
19.        gross_weight = gw;
20.        mpg = input_mpg;
21.    };
22.
23.    float  efficiency(void)
24.    {
25.        return (payload / (payload + gross_weight));
26.    };
27.
28.    float  cost_per_ton(float fuel_cost)
29.    {
30.        return (fuel_cost / (payload / 2000.0));
31.    };
32. };
33.
34. //-----base class number two-----
35. class driver
36. {
37.     protected:
38.    float  hourly_pay;
39.    public:
```

```

40. void initialize(float pay) {hourly_pay = pay; };
41. float cost_per_mile(void) {return (hourly_pay / 55.0); };
42. };
43.
44. //-----derived class-----
45. //inherit from two different base classes
46. class driven_truck : public moving_van, public driver
47. {
48. public:
49. void initialize_all(float pl, float gw, float input_mpg, float pay)
50. { payload = pl;
51.   gross_weight = gw;
52.   mpg = input_mpg;
53.   hourly_pay = pay;
54. };
55.
56. float cost_per_full_day(float cost_of_gas)
57. {
58.   return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg);
59. };
60. };
61.
62. //-----main program-----
63. int main()
64. {
65. //instantiate an object...
66. driven_truck john_merc;
67. john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);
68.
69. cout<<"The efficiency of the Merc truck is "<<john_merc.efficiency()<<
70.   " %\n";
71. cout<<"The cost per mile for John to drive Merc is
72.   $"<<john_merc.cost_per_mile()<<"\n";
73. cout<<"The cost per day for John to drive Merc is
74.   $"<<john_merc.cost_per_full_day(1.129)<<"\n";
75.
76. system("pause");
77.
78. return 0;
79. }

```

79 Lines: Output:

```

C:\bc5\bin\proj0010.exe
The efficiency of the Merc truck is 0.625 %
The cost per mile for John to drive Merc is $0.227273
The cost per day for John to drive Merc is $195.531
Press any key to continue . . .

```

- In order to keep the program as simple as possible for our study, all of the member methods are defined as **inline** functions.
- All variables in both classes are declared to be **protected** so they will be readily available for use in any class that inherits them only.
- Beginning in line 46, we define another class named **driven_truck** which inherits all of the variables and methods from both of the previously defined classes, **moving_van** and **driver**. In the last two Modules, we have discussed how to inherit a single class into another class, and to inherit two or more classes, the same technique is used except that we use a list of inherited classes separated by commas as illustrated in line 46, the class header.

```
class driven_truck : public moving_van, public driver
```

- We use the keyword **public** prior to the name of each inherited class in order to be able to freely use the methods within the derived class. In this case, we didn't define any new variables, but we did introduce two new methods into the derived class in lines 49 through 59 as shown below.

```

void initialize_all(float pl, float gw, float input_mpg, float pay)
{ payload = pl;
  gross_weight = gw;
  mpg = input_mpg;
  hourly_pay = pay;
};

```

```

float cost_per_full_day(float cost_of_gas)
{
    return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg);
};

```

- We define an object named **john_merc** which is composed of four variables, three from the **moving_van** class, and one from the **driver** class.
- Any of these four variables can be manipulated in any of the methods defined within the **driven_truck** class in the same way as in a singly inherited situation.
- A few examples are given in lines 67 through 74 of the main program as shown below.

```

john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);

cout<<"The efficiency of the Merc truck is "<<john_merc. efficiency()<<
" %\n";

cout<<"The cost per mile for John to drive Merc is
      $"<<john_merc.cost_per_mile()<<"\n";
cout<<"The cost per day for John to drive Merc is
      $"<<john_merc.cost_per_full_day(1.129)<<"\n";

```

- All of the rules for **private** or **protected** variables and **public** or **private** method inheritance as used with single inheritance extend to multiple inheritances.

16.3 Duplicate Method Names

- You will notice that both of the base classes have a method named **initialize()**, and both of these are inherited into the subclass. However, if we attempt to send a message to one of these methods, we will have a problem, because the system does not know which one we are referring to. This problem will be solved as illustrated in the next program example.
- Before going on to the next program example, it should be noted that we have not declared any objects of the two base classes in the main program. Be sure to compile and run this program after you understand its operation completely.
- The second program example in this Module named `mulinher2.cpp`, illustrates the use of classes with duplicate method names being inherited into a derived class.

```

1. //Program mulinher2.cpp
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----declaration and implementation class part-----
6. //-----base class number one-----
7. class moving_van
8. {
9.     protected:
10.         float   payload;
11.         float   gross_weight;
12.         float   mpg;
13.     public:
14.         void initialize(float pl, float gw, float input_mpg)
15.         {
16.             payload = pl;
17.             gross_weight = gw;
18.             mpg = input_mpg;
19.         };
20.
21.         float efficiency(void)
22.         {
23.             return (payload / (payload + gross_weight));
24.         };
25.
26.         float cost_per_ton(float fuel_cost)
27.         {
28.             return (fuel_cost / (payload / 2000.0));
29.         };
30.
31.         float cost_per_full_day(float cost_of_gas) //number one
32.         {
33.             return (8.0 * cost_of_gas * 55.0 / mpg);
34.         };
35. };
36.
37. //-----base class number two-----

```

```

38. class driver
39. {
40.     protected:
41.     float hourly_pay;
42.     public:
43.     //same method name as in moving van class..
44.     void initialize(float pay) {hourly_pay = pay; };
45.     float cost_per_mile(void) {return (hourly_pay / 55.0); } ;
46.     float cost_per_full_day(float overtime_premium) //number two
47.         {return (8.0 * hourly_pay); };
48. };
49.
50. //-----derived class-----
51. //notice also the duplicated method names used
52. class driven_truck : public moving_van, public driver
53. {
54.     public:
55.     void initialize_all(float pl, float gw, float input_mpg, float pay)
56.     {
57.         payload = pl;
58.         gross_weight = gw;
59.         mpg = input_mpg;
60.         hourly_pay = pay;
61.     };
62.
63.     float cost_per_full_day(float cost_of_gas) //number three
64.     {
65.         return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg);
66.     };
67. };
68.
69. //-----main program-----
70. int main()
71. {
72.     driven_truck john_merc;
73.
74.     john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);
75.     cout<<"The efficiency of the Merc is "<<john_merc. efficiency()<<" %\n";
76.     cout<<"The cost per mile for John to drive is
77.         $"<<john_merc.cost_per_mile()<<"\n\n";
78.
79.     cout<<"    calling the appropriate method using:\n";
80.     cout<<"    john_merc.moving_van::cost_per_full_day()\n";
81.     cout<<"-----\n";
82.     cout<<"The cost per day for the Merc is
83.         $"<<john_merc.moving_van::cost_per_full_day(1.129)<<"\n\n";
84.
85.     cout<<"    calling the appropriate method using:\n";
86.     cout<<"    john_merc.driver::cost_per_full_day()\n";
87.     cout<<"-----\n";
88.     cout<<"The cost of John for a full day is
89.         $"<<john_merc.driver::cost_per_full_day(15.75)<<"\n\n";
90.
91.     cout<<"    calling the appropriate method using:\n";
92.     cout<<"    john_merc.driven_truck::cost_per_full_day()\n";
93.     cout<<"-----\n";
94.     cout<<"The cost per day for John to drive Merc is
95.         $"<<john_merc.driven_truck::cost_per_full_day(1.129)<<"\n";
96.
97.     system("pause");
98.     return 0;
99. }

```

99 Lines: Output:

```

C:\bc5\bin\proj0010.exe
The efficiency of the Merc is 0.625 %
The cost per mile for John to drive is $0.227273

calling the appropriate method using:
john_merc.moving_van::cost_per_full_day()

-----
The cost per day for the Merc is $95.5308

calling the appropriate method using:
john_merc.driver::cost_per_full_day()

-----
The cost of John for a full day is $100

calling the appropriate method using:
john_merc.driven_truck::cost_per_full_day()

-----
The cost per day for John to drive Merc is $195.531
Press any key to continue . . .

```

- A new method has been added to all three of the classes named `cost_per_full_day()`. This was done intentionally to illustrate how the same method name can be used in all three classes. The class definitions are no problem at all; the methods are simply named and defined as shown.
- The problem comes when we wish to use one of the methods since they all have **same name, numbers and types of parameters and identical return types**. This prevents some sort of an overloading rule to disambiguate the message sent to one or more of the methods.
- The method used to disambiguate the method calls are illustrated in lines 82–83, 88–89, and 94–95 of the main program as shown below:

```

cout<<"The cost per day for the Merc is
$" << john_merc.moving_van::cost_per_full_day(1.129) << "\n\n";
...
...
cout<<"The cost of John for a full day is
$" << john_merc.driver::cost_per_full_day(15.75) << "\n\n";
...
...
cout<<"The cost per day for John to drive Merc is
$" << john_merc.driven_truck::cost_per_full_day(1.129) << "\n";
...
...

```

- The solution is simple by **prepending the class name to the method name with the double colon (::, scope operator)** as used in the method implementation definition. This is referred to as **qualifying the method name**. Actually, qualification is not necessary in line 95 since it is the method in the derived class and it will take precedence over the other method names.
- Actually, you could qualify all method calls, but if the names are unique, the compiler can do it for you and make your code easier to be written and read. Be sure to compile and run this program and study the output and the source code.

16.4 Duplicated Variable Names

- Examine the program example named `mulinher3.cpp`, you will notice that each base class has a variable with the same name, named `weight`.

```

1. //Program mulinher3.cpp
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----class declaration and implementation part-----
6. //-----class number one-----
7. class moving_van
8. {
9.     protected:
10.     float payload;
11.     float weight; //note this variable
12.     float mpg;
13.     public:
14.     void initialize(float pl, float gw, float input_mpg)

```

```

15.     {
16.         payload = pl;
17.         weight = gw;
18.         mpg = input_mpg;
19.     };
20.
21.     float efficiency(void)
22.     { return(payload / (payload + weight)); };
23.
24.     float cost_per_ton(float fuel_cost)
25.     { return (fuel_cost / (payload / 2000.0)); };
26. };
27.
28. //-----class number two-----
29. class driver
30. {
31.     protected:
32.     float hourly_pay;
33.     float weight; //another weight variable
34.     //variable with same name as in class number one
35.     public:
36.     void initialize(float pay, float input_weight)
37.     //same method name but different number of parameter
38.     {
39.         hourly_pay = pay;
40.         weight = input_weight;
41.     };
42.     float cost_per_mile(void) {return (hourly_pay / 55.0); } ;
43.     float drivers_weight(void) {return (weight); };
44. };
45.
46. //-----derived class with multi inheritance-----
47. //-----declaration and implementation-----
48. class driven_truck : public moving_van, public driver
49. {
50.     public:
51.     void initialize_all(float pl, float gw, float input_mpg, float pay)
52.     //another same method name but different number of parameter
53.     {
54.         payload = pl;
55.         moving_van::weight = gw;
56.         mpg = input_mpg;
57.         hourly_pay = pay;
58.     };
59.
60.     float cost_per_full_day(float cost_of_gas)
61.     { return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg); };
62.
63.     float total_weight(void)
64.     //see, how to call different variables with same name
65.     {
66.         cout<<"\nCalling appropriate member variable\n";
67.         cout<<"---->(moving_van::weight)+(driver::weight)\n";
68.         cout<<"-----\n";
69.         return ((moving_van::weight) + (driver::weight));
70.     };
71. };
72.
73. //-----the main program-----
74. int main()
75. {
76.     driven_truck john_merc;
77.
78.     john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);
79.     //accessing the derived class method
80.     john_merc.driver::initialize(15.50, 250.0);
81.     //accessing the base class number two
82.
83.     cout<<"The efficiency of the Merc is "<<john_merc.efficiency()<<" %\n";
84.     cout<<"The cost per mile for John to drive is
85.         $"<<john_merc.cost_per_mile()<<"\n";
86.     cout<<"The cost per day for John to drive Merc is
87.         $"<<john_merc.cost_per_full_day(1.129)<<"\n";
88.     cout<<"The total weight is "<<john_merc.total_weight()<<" ton\n";
89.
90.     system("pause");
91.     return 0;
92. }

```

92 Lines: Output:

- According to the rules of inheritance, an object of the **driven_truck** class will have two variables with the same name, **weight**. This would be a problem if it weren't for the fact that C++ has defined a method of accessing each one in a well defined way. We have to use **qualification** to access each variable.
- Line 69 as shown below, illustrates the use of the variables. It may be obvious, but it should be explicitly stated, the derived class itself can have a variable of the same name as those inherited from the base classes. In order to access it, no qualification would be required, but qualification with the derived class name is permitted.

```
return ((moving_van::weight) + (driver::weight));
```

- It should be apparent to you that once you understand single inheritance, multiple inheritances is nothing more than an extension of the same rules. Of course, if you inherit two methods or variables of the same name, you must use qualification to allow the compiler to select the correct one.
- Constructors are called for both classes before the derived class constructor is executed. The constructors for the base classes are called in the order they are declared in the class header.
- Compile and run this program.

16.5 Parameterized Types

- Many times, when developing a program, you wish to perform some operation on **more than one data type with same program routine**. For example you may wish to sort a list of integers, another list of floating point numbers, and a list of alphabetic strings but the sorting routine should be same.
- It seems silly to write a separate sort function for each of the three types when all three are sorted in the same logical way.
- With **parameterized types**, you will be able to write a single sort routine that is capable of sorting all the three different types of data.
- This is already available in other programming language such as **Ada** language as the **generic package** or **procedure**.
- There is already a library of these components available, as a part of the ANSI-C++ standard and most of the C++ compilers available in the market. It is called the **Standard Template Library**, usually referred to as **STL**.
- It will be very beneficial for you to study this and learn how to use it in your programs. The detail discussion about STL is presented in Part III of this Tutorial. Assume this section as an introduction of the **generic programming**.

16.6 Template – A Function Template

- The following program example named `template1.cpp` is our first example of the template and their usage. This simple program just to illustrate the use of the parameterized type.

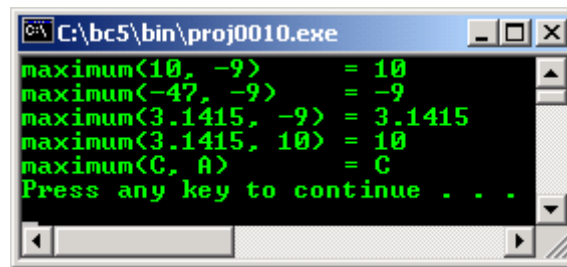
```
1.      //Using function template
2.      #include <iostream.h>
3.      #include <stdlib.h>
4.
5.      //-----template declaration-----
6.      template
7.          <class ANY_TYPE> ANY_TYPE maximum(ANY_TYPE a, ANY_TYPE b)
8.      {
9.          return (a > b) ? a : b;
10.     }
11.
12.     //-----main program-----
```

```

13.     int  main(void)
14.     {
15.     int    x = 10, y = -9;
16.     float  real = 3.1415;
17.     char   ch = 'C';
18.
19.     cout<<"maximum("<<x<<", "<<y<<"      = "<<maximum(x, y)<<"\n";
20.     cout<<"maximum(-47, "<<y<<"      = "<<maximum(-47,y)<<"\n";
21.     cout<<"maximum("<<real<<", "<<float(y)<<" =
22.                                     "<<maximum(real,float(y))<<"\n";
23.     cout<<"maximum("<<real<<", "<<float(x)<<" =
24.                                     "<<maximum(real,float(x))<<"\n";
25.     cout<<"maximum("<<ch<<", "<<'A'<<"      = "<<maximum(ch, 'A')<<"\n";
26.
27.     system("pause");
28.
29.     return 0;
30.     }

```

30 Lines: Output:



```

C:\bc5\bin\proj0010.exe
maximum(10, -9)      = 10
maximum(-47, -9)    = -9
maximum(3.1415, -9) = 3.1415
maximum(3.1415, 10) = 10
maximum(C, A)       = C
Press any key to continue . . .

```

- The template is given in lines 6 through 10 as shown below with the first line indicating that it is a template with a single type to be replaced, the type ANY_TYPE.

```

//-----template declaration-----
template
<class ANY_TYPE> ANY_TYPE maximum(ANY_TYPE a, ANY_TYPE b)
{
    return (a > b) ? a : b;
}

```

- This type can be replaced by any type which can be used in the comparison operation in line 9. If you have defined a class, and you have overloaded the operator ">", then this template can be used with objects of your class. Thus, you do not have to write a **maximum** function for each type or class in your program.
- This function is included automatically for each type it is called with in the program, and the code itself should be very easy to understand.
- You should realize that nearly the same effect can be achieved through the use of a **macro**, except that when a macro is used, the strict **type checking** is not done. Because of this and because of the availability of the inline method capability in C++, the use of macros is essentially been considered non-existent by experienced programmers.

16.7 Template - A Class template

- The following program example named `template2.cpp` provides a template for an entire class rather than a single function.
- The template code is given in lines 8 through 17 and actually it is an entire class definition.

```

1. //Class template
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. const int MAXSIZE = 128;
6.
7. //-----class template-----
8. template <class ANY_TYPE> class stack
9. {
10.     ANY_TYPE array[MAXSIZE];
11.     int stack_pointer;
12.     public:
13.     stack(void) { stack_pointer = 0; };

```



```

14.     void push(ANY_TYPE input_data){ array[stack_pointer++] = input_data; };
15.     ANY_TYPE pop(void)      { return array[--stack_pointer]; };
16.     int empty(void)        { return (stack_pointer == 0); };
17. };
18.
19. char name[] = "Testing, this is an array, name[]";
20.
21. //-----main program-----
22. int main(void)
23. {
24.     int     x = 30, y = -10;
25.     float   real = 4.2425;
26.
27.     stack<int>      int_stack;
28.     stack<float>   float_stack;
29.     stack<char *> string_stack;
30.
31. //storing data
32. int_stack.push(x);
33. int_stack.push(y);
34. int_stack.push(67);
35.
36. float_stack.push(real);
37. float_stack.push(-20.473);
38. float_stack.push(107.03);
39.
40. string_stack.push("This is the first line of string");
41. string_stack.push("This is the second line of string");
42. string_stack.push("This is the third line of string");
43. string_stack.push(name);
44.
45. //displaying data
46. cout<<"-----Displaying data-----\n";
47. cout<<"\nInteger stack\n";
48. cout<<"-----\n";
49. cout<<"Access using int_stack.pop(), first time : "<<int_stack.pop()<<"\n";
50. cout<<"Access using int_stack.pop(), second time: "<<int_stack.pop()<<"\n";
51. cout<<"Access using int_stack.pop(), third time : "<<int_stack.pop()<<"\n";
52.
53. cout<<"\nFloat stack\n";
54. cout<<"-----\n";
55. cout<<"Access using float_stack.pop(), first time :
56.                                     "<<float_stack.pop()<<"\n";
57. cout<<"Access using float_stack.pop(), second time:
58.                                     "<<float_stack.pop()<<"\n";
59. cout<<"Access using float_stack.pop(), third time :
60.                                     "<<float_stack.pop()<<"\n";
61.
62. cout<<"\nString stack\n";
63. cout<<"-----\n";
64.     do
65.     {
66.         cout<<"Access using string_stack.pop(): "<<string_stack.pop()<<"\n";
67.     } while (!string_stack.empty());
68.
69.
70.     system("pause");
71.     return 0;
72. }

```

72 Lines: Output:

```

C:\bc5\bin\proj0010.exe
-----Displaying data-----
Integer stack
-----
Access using int_stack.pop(), first time : 67
Access using int_stack.pop(), second time: -10
Access using int_stack.pop(), third time : 30

Float stack
-----
Access using float_stack.pop(), first time : 107.03
Access using float_stack.pop(), second time: -20.473
Access using float_stack.pop(), third time : 4.2425

String stack
-----
Access using string_stack.pop(): Testing, this is an array, name[]
Access using string_stack.pop(): This is the third line of string
Access using string_stack.pop(): This is the second line of string
Access using string_stack.pop(): This is the first line of string
Press any key to continue . . .

```

- In the main program we create an object named `int_stack` in line 27 which will be a stack designed to store integers, and another object named `float_stack` in line 28 which is designed to store `float` type values as shown below:

```

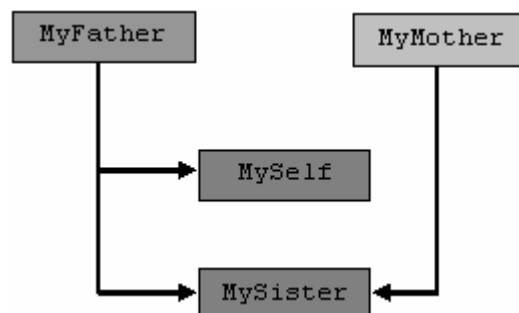
stack<int>      int_stack;
stack<float>    float_stack;

```

- In both cases, we enclose the type we desire this object to work with in "<>" brackets and the system creates the object by first replacing all instances of **ANY_TYPE** with the desired type, then creating the object of that type.
- You will note that any type can be used that has an assignment capability since lines 13 and 14 use the assignment operator on the parameterized type. The assignment operator is used in line 14 because it returns an object of that type which must be assigned to something in the calling program.
- Even though the strings are all of different lengths, we can even use the stack to store the strings if we only store a pointer to the strings and not the entire string. This is illustrated in the object named `string_stack` defined in line 29 and used later in the program.
- Compile and run this program and appreciate it :o).
- You will learn a lot more things about templates, the Standard Template Library in Tutorial #4.

Program Examples and Experiments

- Let have program examples demonstrating the multi inheritance. The simple class hierarchy for this program example is depicted below.



```

//multiple inheritance program example...
#include <iostream.h>
#include <stdlib.h>

//-----class declaration part-----
//base class...
class MyFather
{

```

```

protected:
char* EyeColor;
char* HairType;
double FamSaving;
int FamCar;

public:
MyFather(){}
~MyFather(){}
char* ShowEyeColor();
char* ShowHairType();
long double FamilySaving();
int FamilyCar();
};

//another base class...
class MyMother
{
//notice the same member variables names
//as in MyFather class...
protected:
char* EyeColor;
char* HairType;
int FamHouse;

public:
MyMother(){}
~MyMother(){}
char* ShowMotherEye();
char* ShowMotherHair();
int FamilyHouse();
};

//single inheritance derived class...
//aaahhh!!! my class :- ) finally!!!
class MySelf:public MyFather
{
//another member variables with same names...
char* HairType;
char* Education;

public:
MySelf(){}
~MySelf(){}
char* ShowMyHair();
char* ShowMyEducation();
};

//multiple inheritance derived class...
//notice the keyword public must follow every
//parent class list as needed...
class MySister:public MyFather,public MyMother
{
char* SisEye;
float MonAllowance;

public:
MySister(){}
~MySister(){}
char* ShowSisEye();
float ShowSisAllownace();
};

//-----class implementation part-----
char* MyFather::ShowEyeColor()
{return EyeColor = "Brown";}

char* MyFather::ShowHairType()
{return HairType = "Bald";}

long double MyFather::FamilySaving()
{return FamSaving = 100000L;}

int MyFather::FamilyCar()
{return FamCar = 4;}

char* MyMother::ShowMotherEye()
{return EyeColor = "Blue";}

char* MyMother::ShowMotherHair()

```

```

{return HairType = "Curly Blonde";}

int MyMother::FamilyHouse()
{return FamHouse = 3;}

char* MySelf::ShowMyHair()
{return HairType = "Straight Black";}

char* MySelf::ShowMyEducation()
{return Education = "Post Graduate";}

char* MySister::ShowSisEye()
{return SisEye = "Black";}

float MySister::ShowSisAllownace()
{return MonAllowance = 1000.00;}

//-----main program-----
int main()
{
    //instantiate the objects...
    MyFather ObjFat;
    MyMother ObjMot;
    MySelf ObjSelf;
    MySister ObjSis;

    cout<<"--My father's data--"<<endl;
    cout<<"His eye:          "<<ObjFat.ShowEyeColor()<<"\n"
        <<"His hair:          "<<ObjFat.ShowHairType()<<"\n"
        <<"Family Saving: USD"<<ObjFat.FamilySaving()<<"\n"
        <<"Family Car:         "<<ObjFat.FamilyCar()<<" cars.\n";

    cout<<"\n--My mother's data--"<<endl;
    cout<<"Her eye: "<<ObjMot.ShowMotherEye()<<endl;
    cout<<"Her hair: "<<ObjMot.ShowMotherHair()<<endl;
    cout<<"Our family house: "<<ObjMot.FamilyHouse()<<" houses."<<endl;

    //notice how to access the base/parent class member functions
    //through the child or derived objects...
    cout<<"\n--My data--"<<endl;
    cout<<"My Hair: "<<ObjSelf.ShowMyHair()<<endl;
    cout<<"My family saving: USD"<<ObjSelf.MySelf::FamilySaving()<<endl;
    cout<<"My family car: "<<ObjSelf.MySelf::FamilyCar()<<" cars."<<endl;
    cout<<"My education: "<<ObjSelf.ShowMyEducation()<<endl;

    cout<<"\n--My sister's data--"<<endl;
    cout<<"Her eye: "<<ObjSis.ShowSisEye()<<endl;
    cout<<"Our family saving: USD"<<ObjSis.MySister::FamilySaving()<<endl;
    cout<<"Our family car: "<<ObjSis.MySister::FamilyCar()<<" cars."<<endl;
    cout<<"Our family house: "<<ObjSis.MySister::FamilyHouse()<<" houses."<<endl;
    cout<<"Her monthly allowances: USD"<<ObjSis.ShowSisAllownace()<<endl;

    system("pause");
    return 0;
}

```

Output:

```

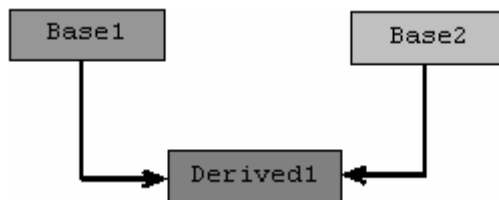
C:\bc5\bin\test.exe
--My father's data--
His eye:      Brown
His hair:     Bald
Family Saving: USD100000
Family Car:   4 cars.

--My mother's data--
Her eye: Blue
Her hair: Curly Blonde
Our family house: 3 houses.

--My data--
My Hair: Straight Black
My family saving: USD100000
My family car: 4 cars.
My education: Post Graduate

--My sister's data--
Her eye: Black
Our family saving: USD100000
Our family car: 4 cars.
Our family house: 3 houses.
Her monthly allowances: USD1000
Press any key to continue . . .

```



```

//another simple multiple
//inheritance program example...
#include <iostream.h>
#include <stdlib.h>

//-----class declaration and implementation-----
//-----part-----
//base class...
class Base1
{
protected:
int SampleDataOne;
public:
Base1(){SampleDataOne = 100;}
~Base1(){}

int SampleFuncOne()
{return SampleDataOne;}
};

//another base class...
class Base2
{
protected:
int SampleDataTwo;
public:
Base2(){SampleDataTwo = 200;}
~Base2(){}

int SampleFuncTwo()
{return SampleDataTwo;}
};

//derived class...
class Derived1:public Base1,public Base2
{
int MyData;
public:
Derived1(){MyData = 300;}
};

```

```

~Derived1(){}
int MyFuncnt()
{
    //the protected data of the base classes are available
    //for this derived class...
    return (MyData + SampleDataOne + SampleDataTwo);
}
};

//-----main program-----
int main()
{
    //instantiate objects...
    Base1 SampleObjOne;
    Base2 SampleObjTwo;
    Derived1 SampleObjThree;

    cout<<"Normal access Base1 class data: "<<SampleObjOne.SampleFuncntOne()<<endl;
    cout<<"Normal access Base2 class data: "<<SampleObjTwo.SampleFuncntTwo()<<endl;

    cout<<"Normal access Derived1 class data: "<<SampleObjThree.MyFuncnt()<<endl;
    cout<<"\nExtracting the Base1 data through the derived class:"<<endl;
    cout<<"Base1 data: "<<SampleObjThree.Base1::SampleFuncntOne()<<endl;

    cout<<"\nExtracting the Base2 data through the derived class:"<<endl;
    cout<<"Base2 data: "<<SampleObjThree.Base2::SampleFuncntTwo()<<endl;

    system("pause");
    return 0;
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Normal access Base1 class data: 100
Normal access Base2 class data: 200
Normal access Derived1 class data: 600

Extracting the Base1 data through the derived class:
Base1 data: 100

Extracting the Base2 data through the derived class:
Base2 data: 200
Press any key to continue . . .

```

- You can try other simple examples of inheritance/multi inheritance and polymorphism (next [Module](#)) in [Typecasting Module](#).
- Example compiled using VC++/VC++ .Net.

```

//Program mulinher3.cpp
#include <iostream>
using namespace std;

//-----class declaration and implementation part-----
//-----class number one-----
class moving_van
{
protected:
double payload;
double weight; //note this variable
double mpg;
public:
void initialize(double pl, double gw, double input_mpg)
{
    payload = pl;
    weight = gw;
    mpg = input_mpg;
};

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

double cost_per_ton(double fuel_cost)
{ return (fuel_cost / (payload / 2000.0)); };

```

```

};

//-----class number two-----
class driver
{
protected:
double hourly_pay;
double weight; //another weight variable
//variable with same name as in class number one
public:
void initialize(double pay, double input_weight)
//same method name but different number of parameter
{
    hourly_pay = pay;
    weight = input_weight;
};
double cost_per_mile(void) {return (hourly_pay / 55.0); };
double drivers_weight(void) {return (weight); };
};

//-----derived class with multi inheritance-----
//-----declaration and implementation-----
class driven_truck : public moving_van, public driver
{
public:
void initialize_all(double pl, double gw, double input_mpg, double pay)
//another same method name but different number of parameter
{
    payload = pl;
    moving_van::weight = gw;
    mpg = input_mpg;
    hourly_pay = pay;
};

double cost_per_full_day(double cost_of_gas)
{ return ((8.0 * hourly_pay) + (8.0 * cost_of_gas * 55.0) / mpg); };

double total_weight(void)
//see, how to call different variables with same name
{
    cout<<"\nCalling appropriate member variable\n";
    cout<<"---->(moving_van::weight)+(driver::weight)\n";
    cout<<"-----\n";
    return ((moving_van::weight) + (driver::weight));
};
};

//-----the main program-----
int main()
{
    driven_truck john_merc;

    john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);
    //accessing the derived class method
    john_merc.driver::initialize(15.50, 250.0);
    //accessing the base class number two

    cout<<"The efficiency of the Merc is "<<john_merc.efficiency()<<" %\n";
    cout<<"The cost per mile for John to drive is $"<<john_merc.cost_per_mile()<<"\n";
    cout<<"The cost per day for John to drive Merc is
    $"<<john_merc.cost_per_full_day(1.129)<<"\n";
    cout<<"The total weight is "<<john_merc.total_weight()<<" ton\n";
    return 0;
}

```

Output :

```

C:\ "g:\vcnetprojek\searchpattern\Debug\searchpattern.e...
The efficiency of the Merc is 0.625 %
The cost per mile for John to drive is $0.281818
The cost per day for John to drive Merc is $219.531

Calling appropriate member variable
-----><moving_van::weight>+<driver::weight>
-----

The total weight is 12250 ton
Press any key to continue

```

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

```

////////- multiherit.cpp-/////////
////////-multi inheritance-/////////
#include <iostream>
using namespace std;

//-----declaration and implementation class part-----
//-----base class number one-----
class moving_van
{
protected:
float   payload;
float   gross_weight;
float   mpg;

public:
void initialize(float pl, float gw, float input_mpg)
{
    payload = pl;
    gross_weight = gw;
    mpg = input_mpg;
};

float efficiency(void)
{return (payload / (payload + gross_weight));};

float cost_per_ton(float fuel_cost)
{return (fuel_cost / (payload / 2000.0));};

float cost_per_full_day(float cost_of_gas) //number one
{return (5.5 * cost_of_gas * 55.0 / mpg);};
};

//-----base class number two-----
class driver
{
protected:
float   hourly_pay;

public:
//same method name as in moving van class.
void initialize(float pay) {hourly_pay = pay;};
float cost_per_mile(void) {return (hourly_pay / 55.0); };
float cost_per_full_day(float overtime_premium) //number two
{return (7.0 * hourly_pay); };
};

//-----derived class-----
//notice also the duplicated method names used
class driven_truck : public moving_van, public driver
{
public:
void initialize_all(float pl, float gw, float input_mpg, float pay)
{
    payload = pl;
    gross_weight = gw;
    mpg = input_mpg;
    hourly_pay = pay;
};

float cost_per_full_day(float cost_of_gas) //number three
{
    return ((7.0 * hourly_pay) + (5.5 * cost_of_gas * 55.0) / mpg);
};
};

```



```

};

//-----main program-----
int main()
{
    driven_truck    john_merc;

    john_merc.initialize_all(20000.0, 12000.0, 5.2, 12.50);
    cout<<"Merc's efficiency is "<<john_merc. efficiency()<<" %\n";
    cout<<"Cost per mile for John to drive is USD"<<john_merc.cost_per_mile()<<"\n\n";

    cout<<"Calling the appropriate method using:\n";
    cout<<"john_merc.moving_van::cost_per_full_day()\n";
    cout<<"-----\n";
    cout<<"Merc's cost per day is
USD"<<john_merc.moving_van::cost_per_full_day(1.129)<<"\n\n";

    cout<<"Calling the appropriate method using:\n";
    cout<<"john_merc.driver::cost_per_full_day()\n";
    cout<<"-----\n";
    cout<<"John's full day cost is
USD"<<john_merc.driver::cost_per_full_day(15.75)<<"\n\n";

    cout<<"Calling the appropriate method using:\n";
    cout<<"john_merc.driven_truck::cost_per_full_day()\n";
    cout<<"-----\n";
    cout<<"Merc's cost per day for John to drive is USD"
        <<john_merc.driven_truck::cost_per_full_day(1.129)<<"\n";

    return 0;
}

```

```

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

```

```

Merc's efficiency is 0.625 %
Cost per mile for John to drive is USD0.227273

Calling the appropriate method using:
john_merc.moving_van::cost_per_full_day()
-----
Merc's cost per day is USD65.6774

Calling the appropriate method using:
john_merc.driver::cost_per_full_day()
-----
John's full day cost is USD87.5

Calling the appropriate method using:
john_merc.driven_truck::cost_per_full_day()
-----
Merc's cost per day for John to drive is USD153.177

```

Further reading and digging:

1. [Check the best selling C/C++ and object oriented books at Amazon.com.](#)
2. Subjects, topics or books related to the following item:
 - a. Object Oriented Design.
 - b. Object Oriented Analysis.
 - c. [Standard Template Library –Tutorial #4 tenouk.com.](#)
3. **Typecasting** that discussed in [Module 22](#) of this Tutorial extensively deals with Inheritance and Multi inheritance.