# Chapter 8

**Objective**

- Polymorphism
- Virtual Functions
- Single Inheritance and Virtual Functions
- Virtual Destructor
- Abstract Class

# Unit 1

## Polymorphism

- Multiple implementation single interface.
- Polymorphism is supported in two ways:
  1. Compile time, by using function overloading and operators overloading
  2. Run time, by using virtual functions.

- Run-time polymorphism provides the greatest flexibility
- Polymorphism increases the level of abstraction in a program.
- Virtual functions are used for supporting polymorphism in C++ language.
- It allows a single operation (method) to behave differently with each derived object.

### Virtual Functions

- A pointer of type base class can be used to refer to any derived class object.
- A pointer to the derived type cannot be used to access an object of the base class
- Virtual function must be a class member function.
- Constructors can not be a virtual function.
- Destructor function may be a virtual function.

# Unit 2

## Using Pointer to Base Class without Virtual Functions

```
1   #include <iostream>   // Example 8-1
2   using namespace std;
3   class CFruit {
4       public:
5               void SetFruitColor(int i) { m_x = i; }
6               int GetFruitColor() { return m_x; }
7       private:
8               int m_x;
9   };
10
11   class CApple : public CFruit {
12      public:
13              void SetAppleColor(int i) { m_y = i; }
14              int GetAppleColor() { return m_y; }
15      private:
16              int m_y;
17   };
18
```

# Unit 2

```
20 int main()
21{
22      CFruit *pFruit;  // pointer of type base class
23      CFruit FruitObj; // object of type base class
24      CApple AppleObj; // object of type derived class
25
26      // assign base object address to the base pointer type
27      pFruit = &FruitObj;
28
29      // use pFruit to access base members
30      pFruit->SetFruitColor(10);
31      cout << "Fruit object gets CFruits data member x: ";
32      cout << pFruit->GetFruitColor() << '\n';
33
```

# Unit 2

```
34        // assign derived object address to base pointer pFruit
35        pFruit = &AppleObj;
36
37        // use pFruit to access derived members
38  //access derived member function using base type pointer
39        pFruit->SetFruitColor(99);
40
41  // !!ERROR SetAppleColor is not a member of CFruit class
42        // pFruit->SetAppleColor(100);
43
44  // can't use pFruit to set CApple class members (i.e y),
45  // so do it directly
46        AppleObj.SetAppleColor(88);
47        cout << "Apple object gets CFruits data member x: "
48        cout << pFruit->GetFruitColor() << '\n';
49        cout << "Apple object gets CApple data member y: "
50        cout << AppleObj.GetAppleColor() << "\n";
51  }
```

# Unit 3

## Virtual Function

```
1 #include <iostream>     // Example 8-2
2 using namespace std;
3
4 class CFruit {
5       public:
6       CFruit(int x) { m_Color = x; }
7       virtual void SetColor()
8       {
9        cout << "Using Fruit version of SetColor(): ";
10       cout << m_Color << "\n";
11      }
12      protected:
13      int m_Color;
14 };
15
```

# Unit 3

```
16 class CApple : public CFruit {
17       public:
18       CApple(int x) : CFruit(x) {}
19       void SetColor()
20       {
21        cout << "Using CApple version of SetColor(): ";
22        cout << m_Color * m_Color << "\n";
23       }
24 };
25
26 class COrange : public CFruit {
27       public:
28       COrange(int x) : CFruit(x) {}
29       void SetColor()
30       {
31        cout << "Using COrange version of SetColor(): ";
32        cout << m_Color + m_Color << "\n";
33       }
34 };
```

# Unit 3

```
35 int main()
36 {
37       CFruit *pFruit;
38       CFruit FruitObj(10);
39       CApple AppleObj(10);
40       COrange OrangeObj(10);
41
42       pFruit = &FruitObj;
43       pFruit->SetColor(); // use CFruit's SetColor()
44
45       pFruit = &AppleObj;
46       pFruit->SetColor();  // use CApple's SetColor()
47
48       pFruit = &OrangeObj;
49       pFruit->SetColor();  // use COrange's SetColor()
50 }
```

# Unit 4

## Simplified Code Design by Using Virtual Function

- Use virtual functions to respond to random events occurring at run time.

```
0 #include <iostream>     // Example 8-3
1 #include <cstdlib>
2 using namespace std;
3
4 class CFruit {
5  public:
6  CFruit(int x) { m_Color = x; }
7  virtual void SetColor()
8  {
9   cout << "Using base version of SetColor(): ";
10  cout << m_Color << "\n";
11 }
12 protected:
13 int m_Color;
14};
15
```

# Unit 4

```
16 class CApple : public CFruit {
17       public:
18       CApple(int x) : CFruit(x) {}
19       void SetColor()
20       {
21        cout << "Using Apple's version of SetColor(): ";
22        cout << m_Color * m_Color << "\n";
23       }
24};
25
26 class COrange : public CFruit {
27       public:
28       COrange(int x) : CFruit(x) {}
29       void SetColor()
30       {
31        cout << "Using Orange's version of SetColor() : ";
32        cout << m_Color + m_Color << "\n";
33       }
34 };
```

# Unit 4

```
35 int main()
36 {
37      CFruit *pFruit;
38
39      for (int i = 0; i < 10; i++) {
40              if (rand() % 2) {
41      // if odd use AppleObj
42                      pFruit = new CApple(10);
43              }
44              else {
45                      // if even use OrangeObj
46                      pFruit = new COrange(10);
47              }
48
49              pFruit->SetColor();      // call appropriate function
50
51              delete pFruit;   // Delete appropriate object
52
53      }       // end for loop
54 }
```

# Unit 5

## Single Inheritance and Virtual Functions

- Virtual functions are called through a vector of functions pointers called “virtual table”
- Each class containing virtual functions has virtual table.
- Each object of that class contains a pointer of that virtual table.

```
0 #include <iostream>     // Example 8-4
1 using namespace std;
2
3 enum Birds {BIRD,PARROT,PARAKEET};
4
5 class CBird {
6   public:
7   virtual void GetData();
8   virtual void History() { }
9   virtual void ShowData() { cout << m_Color << '\n'; }
10
11 protected:
12  int m_Color;
13 };
14
```

# Unit 5

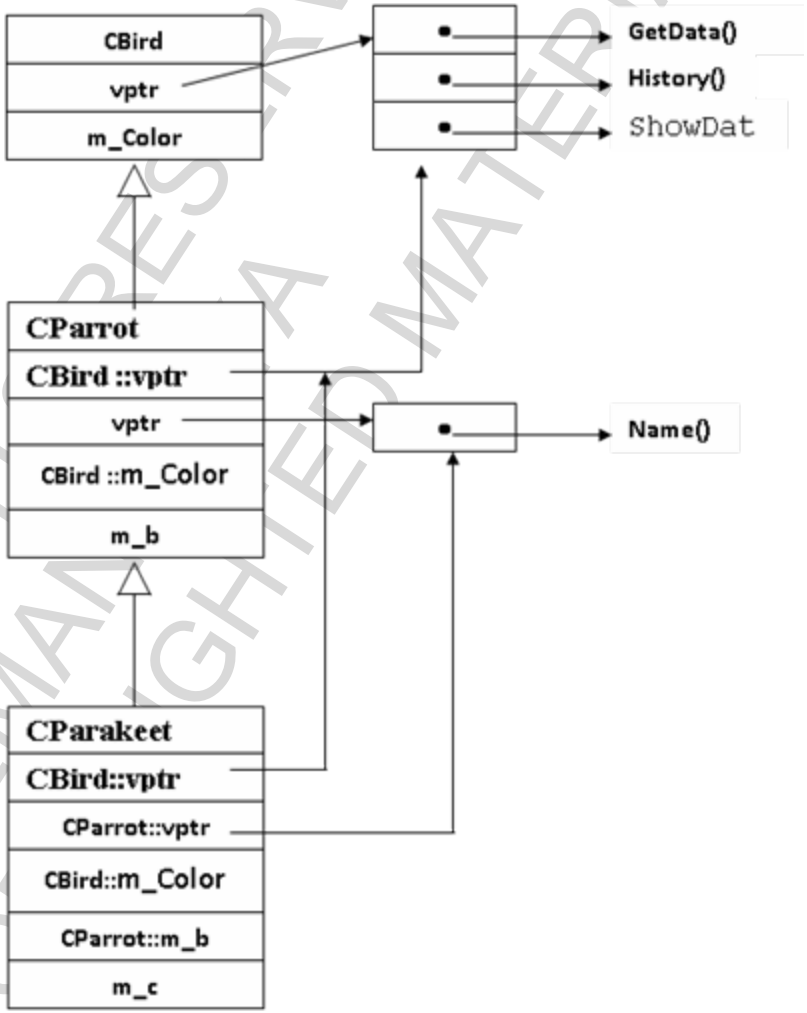
```
15 void CBird::GetData()
16 {
17       cout << "Enter Data For Bird: ";
18       cin >> m_Color;
19 }
20 // Inherit direct base class.
21 class CParrot : public CBird {
22 public:
23        void GetData();
24        void ShowData() { cout << m_b << '\n'; }
25        virtual void Name() { }
26
27 protected:
28       int m_b;
29 };
30 void CParrot::GetData()
31{
32       cout << "Enter Data For Parrot: ";
34       cin >>  m_b;
35 }
```

# Unit 5

```
37// Inherit a CParrot class and an indirect CBird class.
38 class CParakeet : public CParrot {
39       public:
40       void GetData();
41       void ShowData()  { cout << m_c << '\n'; }
42       void Name() { }
43
44
45       private:
46       int m_c;
47 };
48
49 void CParakeet::GetData()
50 {
51       cout << "Enter Data For Parakeet: ";
52       cin >> m_c;
53 }
```

# Unit 5

## Virtual Function Control Flow



# Unit 5

```
54 int main()
55 {
56       CBird *pBird;  // Base type pointer
57       int nBirdType = 0;
58       cout << "Enter Bird Type: ";
59       cin >> nBirdType;
60       switch (nBirdType) {
61       case BIRD:
62               pBird = new CBird;
63               break;
64       case PARROT:
65               pBird = new CParrot;
66               break;
67       case PARAKEET:
68               pBird = new CParakeet;
69               break;
70       }
71       pBird->GetData();
72       pBird->ShowData();
73       delete pBird;
74 }
```

# Unit 6

## Virtual Destructor

- If base class destructor is virtual then every destructor in the derivation hierarchy will be virtual.
- Specifying the destructors in a derivation hierarchy as virtual guarantees that the appropriate destructors are invoked whenever delete is applied to a base class pointer.

```
1  #include <iostream>    // Example 8-5
2  using namespace std;
3  class CFruit { // base class
4       public:
5            CFruit();
6            virtual ~CFruit();
7       private:
8            char *pFruit;
9  };
10
11  CFruit::CFruit() // Base class constructor
12  {
13     pFruit = new char[5];
14     cout << "Fruit class allocates 5 bytes\n";
15  }
```

# Unit 6

```
16  CFruit::~CFruit() // Base class destructor
17  {
18     delete[] pFruit;
19     cout << "Fruit class frees 5 bytes\n";
20  }
21
22  class CApple : public CFruit { // derived class
23       public:
24            CApple();
25            ~CApple(); // derived destructor
26       private:
27             char *pApple;
28  };
29
30
31  CApple::CApple() : CFruit() // Derived class constructor
32  {
33      pApple = new char[1000];
34      cout << "CApple class allocates 1000 bytes\n";
35  }
```

# Unit 6

```
36  CApple::~CApple() // derived class destructor
37  {
38     delete[] pApple;
39     cout << "CApple class frees 1000 bytes\n";
40  }
41
42  int main()
43  {
44     const int Forever = 1;
45
46     while (Forever) {
47         CFruit *pFruit = new CApple;
48
49         delete pFruit;
50     }
51  }
```

# Unit 7

## Abstract Class

- A class with one or more **pure** virtual functions is an abstract class.
- Pure virtual functions have **no** body.
- Object **cannot** be created for an abstract class.
- The derived class of an abstract class **must** provide a body for a pure virtual function.

```
1  #include <iostream>    // Example 8-6
2  using namespace std;
3
4  class CArea {
5      protected:
6            double m_fDim1;// dimensions of figure
7            double m_fDim2;
8      public:
9            void Setarea(double d1, double d2) { m_fDim1=d1; m_fDim2=d2; }
10           virtual double Getarea() = 0;
11  };
```

# Unit 7

```
12  class CTriangle : public CArea {
13       public:
14            double Getarea() { return(0.5 * m_fDim1 * m_fDim2); }
15  };
16
17  class CRectangle : public CArea {
18      public:
19           double Getarea() { return (m_fDim1 * m_fDim2); }
20  };
```

# Unit 7

```
21  int main()
22  {
23      CArea *pArea;
24      CRectangle *pR = new CRectangle;
25      CTriangle *pT = new CTriangle;
26
27      pR->Setarea(3.3,4.5);
28      pT->Setarea(4.0,5.0);
29
30      pArea = pR;
31      cout << "Rectangle has area: " << pArea->Getarea() << "\n";
32
33      pArea = pT;
34      cout << "Triangle has area: " << pArea->Getarea() << "\n";
35
36      delete pR;
37      delete pT;
38  }
```

OUTPUT:

**Rectangle has area: 14.85**
**Triangle has area: 10**