Polymorphism

Polymorphism is implemented when you have (a) derived class(es) containing a member function with the same signature as a base class. A function invoked through a pointer or a reference to the base class, will execute the correct implementation regardless of whether the pointer is pointing at a base class object or a derived class object. Functions that behave in this way are called virtual functions. The determination of which function to call is not known at compile-time, so the correct function is selected during execution. This process is called late binding, or dynamic binding. The usual call of a function through an object, is known to the compiler, hence, early binding or static binding.

Non-virtual vs. Virtual Functions

Example 7-9 - Non virtual Functions

This example and the next one deomonstrate the difference between a virtual and a non-virtual function.

```
// File: ex7-9.cpp – Inheritance with a non-virtual function

#include <iostream>
using namespace std;

class B
{
public:
    B()
     {
         cout << "B ctor called for " << this << endl;
     }
     void funk1()
     {
         cout << "B::funk1() called for " << this << endl;
     }
     void funk2()
     {
         cout << "B::funk2() called for " << this << endl;
     }
 };

 class D : public B
 {
 public:
     D()
     {
         cout << "D ctor called for " << this << endl;
     }
     // Override funk1()
     void funk1()
```

```
    {
        cout << "D::funk1() called for " << this << endl;
    }
};

int main()
{
    B b;
    D d;
    cout << endl;

    b.funk1();
    d.funk1();
    cout << endl;

    b.funk2();
    d.funk2();
    cout << endl;

    B* pB;
    pB = &b;
    pB->funk1();
    cout << endl;

    pB = &d;
    pB->funk1();
    cout << endl;

    cout << "size of b = " << sizeof b << endl;
    cout << "size of d = " << sizeof d << endl;
}
```

```
******  Output  ******

B ctor called for 0x69fefb
B ctor called for 0x69fefa
D ctor called for 0x69fefa

B::funk1() called for 0x69fefb
D::funk1() called for 0x69fefa

B::funk2() called for 0x69fefb
B::funk2() called for 0x69fefa

B::funk1() called for 0x69fefb

B::funk1() called for 0x69fefa

size of b = 1
size of d = 1
```

✔ Why does a B and a D object have a size of 1?

Example 7-10 - Virtual Functions

This example is the same as the last one, except that funk1() is declared a virtual function. Hence, this program implements polymorphism.

```
// File: ex7-10.cpp - Inheritance with a virtual function

#include <iostream>
using namespace std;

class B
{
public:
   B() { cout << "B ctor called for " << this << endl;}
   void funk1() { cout << "B::funk1() called for " << this << endl; }
   virtual void funk2() { cout << "B::funk2() called for " << this <<
endl; }
 };

 class D : public B
 {
 public:
   D() { cout << "D ctor called for " << this << endl;}
   void funk1() { cout << "D::funk1() called for " << this << endl; }
   virtual void funk2() { cout << "D::funk2() called for " << this <<
endl; }
 };

 int main()
 {
   B b;
   D d;
   cout << endl;

   b.funk1();
   d.funk1();
   cout << endl;

   b.funk2();
   d.funk2();
   cout << endl;

   B* pB;
   pB = &b;
   pB->funk1();
   pB->funk2();
   cout << endl;

   pB = &d;
   pB->funk1();
   pB->funk2();

```

```
    cout << endl;

    cout << "size of b = " << sizeof b << endl;
    cout << "size of d = " << sizeof d << endl;
}
```

```
******  Output  ******

B ctor called for 0x69fef8
B ctor called for 0x69fef4
D ctor called for 0x69fef4

B::funk1() called for 0x69fef8
D::funk1() called for 0x69fef4

B::funk2() called for 0x69fef8
D::funk2() called for 0x69fef4

B::funk1() called for 0x69fef8
B::funk2() called for 0x69fef8

B::funk1() called for 0x69fef4
D::funk2() called for 0x69fef4

size of b = 4
size of d = 4
```

Example 7-11 - Virtual Functions

This example illustrates that

1) a virtual function does not have to be overridden in the derived class and
2) also that you may not execute a derived class function that is not defined in the base class through a base class pointer even if the pointer is pointing at a derived class object.

```
// File: ex7-11.cpp

#include <iostream>
using namespace std;

class B
{
protected:
    int b;
public:
    B()
    {
        cout << "B ctor called for " << this << endl;
        b = 0;
    }
    virtual void virt()
    {
        cout << "B::virt() called for " << this << endl;
    }
};

class D : public B
{
protected:
    int d;
public:
    D()
    {
        cout << "D ctor called for " << this << endl;
        d = 0;
    }
    void non_virt2()
    {
        cout << "D::non_virt2() called for " << this << endl;
    }
};

int main()
{
    B b;                    // declare a base object
    D d;                    // declare a derived object

    b.virt();               // invoke virt() through a base object
    d.virt();               // invoke virt() through a derived object

    B* pb;                  // pb is a pointer to a base class object
```

```
    pb = &b;       // pb points to b

    pb->virt();          // invoke virt() through a base pointer
    //   to a base object

    pb = &d;       // pb points to d
    pb->virt();          // invoke virt() through a base pointer
    //   to a derived object

    cout << "size of b = " << sizeof b << endl;
    cout << "size of d = " << sizeof d << endl;
    d.non_virt2(); // invoke non_virt2() through derived object
//  pb->non_virt2(); Error:  non_virt2() is not a member of B
}

```

```
******  Output  ******

B ctor called for 0x69fef4
B ctor called for 0x69fee8
D ctor called for 0x69fee8
B::virt() called for 0x69fef4
B::virt() called for 0x69fee8
B::virt() called for 0x69fef4
B::virt() called for 0x69fee8
size of b = 8
size of d = 12
D::non_virt2() called for 0x69fee8
```

Example 7-12 - Virtual Functions

This example shows that

1) "virtualness" is passed down to derived classes even if the immediate "parent" class does not name a function as virtual and
2) polymorphism may be implemented through references instead of pointers to base objects.

```
// File: ex7-12.cpp

// This example shows that "virtualness" is passed down to derived classes
// even if the immediate "parent" class does not name a function as virtual.
// It also illustrates polymorphism implemented through references instead
// of pointers to base objects.

#include <iostream>
#include <string>
 using namespace std;

 class person
```

```
{
public:
    virtual string who_am_i() const
    {
        return "person";
    }
    string non_virtual_who_am_i() const
    {
        return "non_virtual person";
    }
};

class child : public person
{
public:
    string who_am_i() const
    {
        return "child";
    }
    string non_virtual_who_am_i() const
    {
        return "non_virtual child";
    }
};
class grand_child : public child
{
public:
    string who_am_i() const
    {
        return "grand_child";
    }
    string non_virtual_who_am_i() const
    {
        return "non_virtual grand_child";
    }
};

void identify_yourself(const person& p)
{
    cout << "I am a " << (p.who_am_i()) << endl;
    cout << "I am a " << (p.non_virtual_who_am_i()) << endl;
}

int main()
{
    person P;
    child C;
    grand_child G;
    person* pp;
    pp = &P;
    cout << (pp->who_am_i()) << endl;
    cout << (pp->non_virtual_who_am_i()) << endl;
    pp = &C;
```

```
    cout << (pp->who_am_i()) << endl;
    cout << (pp->non_virtual_who_am_i()) << endl;
    pp = &G;
    cout << (pp->who_am_i()) << endl;
    cout << (pp->non_virtual_who_am_i()) << endl;
    cout << "sizeof(person) = " << sizeof(person) << endl;
    cout << "sizeof(child) = " << sizeof(child) << endl;
    cout << "sizeof(grand_child) = " << sizeof(grand_child) << endl;
    identify_yourself(P);
    identify_yourself(C);
    identify_yourself(G);
}
```

```
******  Output  ******

person
non_virtual person
child
non_virtual person
grand_child
non_virtual person
sizeof(person) = 4
sizeof(child) = 4
sizeof(grand_child) = 4
I am a person
I am a non_virtual person
I am a child
I am a non_virtual person
I am a grand_child
I am a non_virtual person
```

Why write a Virtual destructor?

Example 7-13
This example illustrates why you might want to write a virtual destructor.

```
// File: ex7-13.cpp - Why a Virtual destructor?

#include <iostream>
using namespace std;

class X
{
public:
    X()
    {
        cout << "X constructor called\n";
    }
    ~X()
    {
        cout << "X destructor called\n";
    }
};


class A : public X
{
public:
    A()
    {
        cout << "A constructor called\n";
    }
    ~A()
    {
        cout << "A destructor called\n";
    }
};


int main()
{
    X* ptrX;

    ptrX = new X;
    delete ptrX;

    cout << endl;

    ptrX = new A;
    delete ptrX;
}
```

```
******  Output  ******
```

```
X constructor called
X destructor called

X constructor called
A constructor called
X destructor called
```

- What's the problem?

Example 7-14
This example shows how to write a virtual destructor. Compare the output with the last example.

```
// File: ex7-14.cpp - Why a Virtual destructor? Here's why!

#include <iostream>
using namespace std;


class X
{
public:
     X()
     {
         cout << "X constructor called\n";
     }
     virtual ~X()
     {
         cout << "X destructor called\n";
     }
};


class A : public X
{
public:
     A()
     {
         cout << "A constructor called\n";
     }
     ~A()
     {
         cout << "A destructor called\n";
     }
};


int main()
{
     X* ptrX;

     ptrX = new X;
     delete ptrX;

     cout << endl;

     ptrX = new A;
     delete ptrX;
}
```

```
******  Output  ******
```

```
X constructor called
X destructor called

X constructor called
A constructor called
A destructor called
X destructor called
```

Note: it is not necessary to repeat the **virtual** for the destructor in the derived class.

Non-Virtual, Virtual, and Pure Virtual Functions

The following notes differentiate these three types of class member functions:

Non-Virtual

- This is the default type of class member function. The keyword ***virtual*** does not appear in the function prototype.
- Non-virtual functions, as a rule, are not usually overridden in the derived class.

Virtual

- The keyword ***virtual*** appears at the beginning of the function prototype in the base class. It doesn't have to be used in derived class function prototypes, but it's not a bad idea to use it.
- Virtual functions, as a rule, are usually overridden in the derived class.
- Virtual functions make polymorphism possible.

Pure Virtual

- The keyword ***virtual*** appears at the beginning and **= 0** at the end of the function prototype in the base class. The **= 0** is not repeated in derived classes unless that class is intended to serve as a base class for other derived classes.
- Pure virtual functions must be overridden in the derived class, unless, that class is also a base class for other classes.
- Pure virtual functions are not defined in the class in which they are declared as pure vitual.
- The presence of a pure virtual function in a class makes it an abstract class. Abstract classes may not be instantiated.

Abstract Classes and Pure Virtual Functions

The following example is the traditional shape class example, illustrating the abstract base class, shape, with pure vitual functions.

Example 7-15 - Abstract classes and pure virtual functions

```
// File: ex7-15.cpp - Abstract classes

#include <iostream>
#include <cmath>
#include <cstdlib>
using namespace std;

const double pi = 3.141592654;

class Shape
{
protected:
    double x;
    double y;
public:
    Shape(double = 0,double = 0);
    double get_x() const
    {
        return x;
    }
    double get_y() const
    {
        return y;
    }
    virtual double area() const = 0;    // pure virtual function
    virtual double girth() const = 0;   // pure virtual function
};

Shape::Shape(double c_x, double c_y) : x(c_x),y(c_y) {}

ostream& operator<<(ostream& out, const Shape& object)
{
    cout << '(' << object.get_x() << ',' << object.get_y() << ')';
    return out;
}

class Square : public Shape
{
private:
    double side;
public:
    Square(double c_x,double c_y, double s);
    double get_side()
    {
        return side;
```

```
    }
    double area() const;
    double girth() const;
};

Square::Square(double c_x, double c_y, double s) : Shape(c_x,c_y), side(s)
{ }

double Square::area() const
{
    return side * side;
}

double Square::girth() const
{
    return 4.*side;
}

class Triangle : public Shape
{
private:
    double a,b,c; // length of 3 sides
public:
    Triangle(double c_x,double c_y, double s1, double s2, double s3);
    void print_sides();
    double area() const;
    double girth() const;
};

Triangle::Triangle(double c_x, double c_y, double s1, double s2, double s3)
    : Shape(c_x,c_y), a(s1), b(s2), c(s3)
{ }

void Triangle::print_sides()
{
    cout << a << ' ' << b << ' ' << c;
}

double Triangle::area() const
{
    double s = (a + b + c) / 2.;   // semiperimeter
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

double Triangle::girth() const
{
    return a+b+c;
}

class Circle : public Shape
{
```

```
private:
    double radius;
public:
     Circle(double c_x, double c_y, double r);
     double get_radius()
     {
         return radius;
     }
     double area() const;
     double girth() const;
 };

 Circle::Circle(double c_x, double c_y, double r) : Shape(c_x,c_y), radius(r)
 { }

 double Circle::area() const
 {
     return pi*radius*radius;
 }

 double Circle::girth() const
 {
     return 2.*pi*radius;
 }

 int main()
 {
     //  Shape sh(6,7); can't create instance of abstract class
     Circle c(3,4,5);
     cout << "Circle c - center: ";
     cout << c << endl;
     cout << "  radius = " << c.get_radius();
     cout << "  area = " << c.area();
     cout << "  circumference = " << c.girth() << endl;

     Square s(5.,2.,1.);
     cout << "Square s - center: ";
     cout << s << endl;
     cout << "  side = " << s.get_side();
     cout << "  area = " << s.area();
     cout << "  perimeter = " << s.girth() << endl;

     Triangle t(0,0,3,4,5);
     cout << "Triangle t - center: ";
     cout << t << endl;
     cout << "  sides = ";
     t.print_sides();
     cout << "  area = " << t.area();
     cout << "  perimeter = " << t.girth() << endl;

     cout << "sizeof(double)=" << sizeof(double) << endl;
     cout << "sizeof(Shape)=" << sizeof(Shape) << endl;
```

```
    cout << "sizeof(Square)=" << sizeof(Square) << endl;
    cout << "sizeof(Triangle)=" << sizeof(Triangle) << endl;
    cout << "sizeof(Circle)=" << sizeof(Circle) << endl;
}
```

```
******  Output  ******

circle c - center: (3,4)  radius = 5  area = 78.5398  circumference = 31.4159
square s - center: (5,2)  side = 1  area = 1  perimeter = 4
triangle t - center: (0,0)  sides = 3 4 5  area = 6  perimeter = 12
sizeof(double)=8
sizeof(shape)=24
sizeof(square)=32
sizeof(triangle)=48
sizeof(circle)=32
```

Example 7-16 - Life

The following example is a practical application which make use of polymorphism and an abstract class.

```
// File: ex7-16.cpp - Life and polymorphism

#include <iostream>
#include <cstdlib>
using namespace std;

enum Bool { FALSE, TRUE};
enum LifeForm {VACANT, WEED, RABBIT, HAWK};

const int GridSize = 10;
const int Cycles = 10;
const int NumberLifeForms = 4;
const int HawkLifeExpectancy = 8;
const int HawkOvercrowdingLimit = 3;
const int RabbitLifeExpectancy = 3;

class Grid;

class LivingThing
{
protected:
    int x,y;
    void AssessNeighborhood(const Grid& G, int sm[]);
public:
    LivingThing(int _x, int _y): x(_x), y(_y) {}
    virtual ~LivingThing() {}
    virtual LifeForm WhoAmI() const = 0;
    virtual LivingThing* next(const Grid& G) = 0;
};

class Grid
{
private:
    LivingThing* cell[GridSize][GridSize];
public:
    Grid();
    ~Grid()
    {
        if (cell[1][1]) release();
    }
    void update(Grid&);
    void release();
    void print();
    LivingThing* get_cell(int row, int col) const;
};

/* This function counts the number of each LivingThing thing in
  the neighborhood.  A neighborhood is a square and the 8
  adjacent squares on each side of it */
```

```
void LivingThing::AssessNeighborhood(const Grid& G, int count[])
{
    int i, j;
    count[VACANT] = count[WEED] = count[RABBIT] = count[HAWK] = 0;
    for (i = -1; i <= 1; ++i)
        for (j = -1; j <= 1; ++j)
            count[G.get_cell(x+i,y+j) -> WhoAmI()]++;
}

LivingThing* Grid::get_cell(int row, int col) const
{
    return cell[row][col];
}

class Vacant : public LivingThing
{
public:
    Vacant(int _x, int _y):LivingThing(_x,_y) {}
    LifeForm WhoAmI() const
    {
        return (VACANT);
    }
    LivingThing* next(const Grid& G);
};

class Weed : public LivingThing
{
public:
    Weed(int _x, int _y): LivingThing(_x,_y) {}
    LifeForm WhoAmI() const
    {
        return (WEED);
    }
    LivingThing* next(const Grid& G);
};

class Rabbit : public LivingThing
{
protected:
    int age;
public:
    Rabbit(int x, int y, int a = 0) : LivingThing(x,y), age(a) {}
    LifeForm WhoAmI() const
    {
        return (RABBIT);
    }
    LivingThing* next(const Grid& G);
};

class Hawk : public LivingThing
{
protected:
    int age;
```

```
public:
    Hawk(int x, int y, int a = 0): LivingThing(x,y), age(a) {}
    LifeForm WhoAmI() const
    {
        return (HAWK);
    }
    LivingThing* next(const Grid& G);
};

// This function determines what will be in an Vacant square in the next cycle
LivingThing* Vacant::next(const Grid& G)
{
    int count[NumberLifeForms];
    AssessNeighborhood(G,count);

// If there is more than one Rabbit in the neighborhood, a new Rabbit
//     is born.
    if (count[RABBIT] > 1)   return (new Rabbit(x,y));

// otherwise, if there is more than one Hawk, a Hawk will be born
    else if (count[HAWK] > 1) return (new Hawk(x, y));

// otherwise, if there is Weed in the neighborhood, Weed will grow
    else if (count[WEED])   return (new Weed(x, y));

// otherwise the square will remain Vacant
    else return (new Vacant(x, y));
}

// if there is more Weeds than Rabbits, then new Weed will grow,
// otherwise Vacant
LivingThing* Weed::next(const Grid& G)
{
    int count[NumberLifeForms];
    AssessNeighborhood(G, count);
    if (count[WEED] > count[RABBIT])   return (new Weed(x, y));
    else return (new Vacant(x, y));
}

/* The Rabbit dies if:
 there's more Hawks in the neighborhood than Rabbits
 not enough to eat
 or if it's too old
 otherwise a new Rabbit is born */
LivingThing* Rabbit::next(const Grid& G)
{
    int count[NumberLifeForms];
    AssessNeighborhood(G, count);
    if (count[HAWK] >= count[RABBIT] ) return (new Vacant(x, y));
    else if (count[RABBIT] > count[WEED]) return (new Vacant(x, y));
    else if (age > RabbitLifeExpectancy) return (new Vacant(x, y));
    else return (new Rabbit(x,y, age + 1));
```

```
}

// Hawk die of overcrowding, starvation, or old age
LivingThing* Hawk::next(const Grid& G)
{
    int count[NumberLifeForms];
    AssessNeighborhood(G, count);
    if (count[HAWK] > HawkOvercrowdingLimit) return (new Vacant(x, y));
    else if (count[RABBIT] < 1) return (new Vacant(x,y));
    else if (age > HawkLifeExpectancy) return (new Vacant (x, y));
    else return (new Hawk(x, y, age + 1));
}

Grid::Grid()
{
    LifeForm creature;
    int i, j;
    for (i = 0; i < GridSize; i++)
        for (j = 0; j < GridSize; j++)
        {
            if (i == 0 || i == GridSize - 1 || j ==0 || j == GridSize - 1)
                creature = VACANT;
            else
                creature = LifeForm(rand() % NumberLifeForms);
            switch (creature)
            {
            case HAWK:
                cell[i][j] = new Hawk(i,j);
                break;
            case RABBIT:
                cell[i][j] = new Rabbit(i,j);
                break;
            case WEED:
                cell[i][j] = new Weed(i,j);
                break;
            case VACANT:
                cell[i][j] = new Vacant(i,j);
            }
        }
}

void Grid::release()
{
    int i, j;
    for (i = 1; i < GridSize - 1; ++i)
        for (j = 1; j < GridSize - 1; ++j) delete cell[i][j];
    cell[1][1] = 0;
}

void Grid::update(Grid& old)
{
    int i, j;
```

```
    for (i = 1; i < GridSize - 1; ++i)
        for (j = 1; j < GridSize - 1; ++j)
            cell[i][j] = old.cell[i][j] -> next(old);
}

void Grid::print()
{
    LifeForm creature;
    int i, j;
    for (i = 1; i < GridSize - 1; i++)
    {
        for (j = 1; j < GridSize - 1; j++)
        {
            creature = cell[i][j]->WhoAmI();
            switch (creature)
            {
            case HAWK:
                cout << "H";
                break;
            case RABBIT:
                cout << "R";
                break;
            case WEED:
                cout << "W";
                break;
            case VACANT:
                cout << "0";
            }
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    Grid G1, G2;
    G1.print();

    for (int i = 1; i <= Cycles; i++)
    {
        cout << "Cycle " << i << endl;
        if (i % 2)
        {
            G2.update(G1);
            G2.print();
            G1.release();
        }
        else
        {
            G1.update(G2);
            G1.print();
            G2.release();
```

```
            }
        }
    }
```

```
******  Output  ******

   WHR0W0RR
   R0WWWHWH
   HRH0H0RW
   0RW0RHHR
   HRW0HHHR
   HHRWW0WH
   W0H0RWWH
   HWWR0HRR

   Cycle 1
   0H0WWW00
   0R0WWH0H
   H0HHHR00
   R00H0000
   H00RH000
   H00WWHWH
   WHHRRWWH
   0WW0RH00

   Cycle 2
   0HWWWWH0
   H0HWWHH0
   HRH000H0
   0HH0RH00
   HHW0HHHW
   0HR00HW0
   W0HRRWW0
   WWWR0HHW

   Cycle 3
   H0WWWW0H
   HHHWW00H
   H00HHH0H
   H00H00HH
   000R000W
   H00RR0WW
   WHH00WWW
   WW00RH0W

   Cycle 4
   0HWWWWH0
   000WWHH0
   0HH000H0
   0HHHHH00
   HHR0RWHW
   0HR00WWW
   W0HRRWWW
   WWH00HWW

   Cycle 5
   00WWWW0H
   HHHWW00H
   H00HHH0H
   H000H0HH
   000R0W0W
   H00RRWWW
   WHH0RWWW
   WWHRRHWW

   Cycle 6
   HHWWWWH0
   000WWHH0
   0HH000H0
   0H0H0H00
   HHR0RWHW
   0HR00WWW
   W0HR0WWW
   WWH00HWW

   Cycle 7
   00WWWW0H
   HHHWW00H
   H00HHH0H
   H0HHHHHH
   000R0W0W
   H00RRWWW
   WHH0WWWW
   WWHHW0WW

   Cycle 8
   HHWWWWH0
   000WWHH0
   0HH000H0
   0HH00000
   HHR0RWHW
   0HR0RWWW
   W00RWWWW
   WW00WWWW

   Cycle 9
   00WWWW0H
   HHHWW00H
   H00HWH0H
   H00RWHHH
   000RRW0W
   HH0RRWWW
   WWR0WWWW
   WWWWWWWW

   Cycle 10
   HHWWWWH0
   000WWWH0
```

```
0HHHW0H0
0HR00H00
HHR00WHW
0HR0RWWW
WWRRWWWW
WWWWWWW
```