

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 demonstrate the difference between a virtual and a non-virtual function.

```
1 // File: ex7-9.cpp - Inheritance with a non-virtual function
2
3 #include <iostream>
4 using namespace std;
5
6 class B
7 {
8 public:
9     B()
10    {
11        cout << "B ctor called for " << this << endl;
12    }
13    void funk1()
14    {
15        cout << "B::funk1() called for " << this << endl;
16    }
17    void funk2()
18    {
19        cout << "B::funk2() called for " << this << endl;
20    }
21 };
22
23 class D : public B
24 {
25 public:
26     D()
27     {
28         cout << "D ctor called for " << this << endl;
29     }
30     // Override funk1()
31     void funk1()
```

```
32     {
33         cout << "D::funk1() called for " << this << endl;
34     }
35 };
36
37 int main()
38 {
39     B b;
40     D d;
41     cout << endl;
42
43     b.funk1();
44     d.funk1();
45     cout << endl;
46
47     b.funk2();
48     d.funk2();
49     cout << endl;
50
51     B* pB;
52     pB = &b;
53     pB->funk1();
54     cout << endl;
55
56     pB = &d;
57     pB->funk1();
58     cout << endl;
59
60     cout << "size of b = " << sizeof b << endl;
61     cout << "size of d = " << sizeof d << endl;
62 }
```

***** Output *****

```
B ctor called for 0x69fefb
B ctor called for 0x69fefafa
D ctor called for 0x69fefafa
```

```
B::funk1() called for 0x69fefb
D::funk1() called for 0x69fefafa
```

```
B::funk2() called for 0x69fefb
B::funk2() called for 0x69fefafa
```

```
B::funk1() called for 0x69fefb
```

```
B::funk1() called for 0x69fefafa
```

```
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.

```
1 // File: ex7-10.cpp - Inheritance with a virtual function
2
3 #include <iostream>
4 using namespace std;
5
6 class B
7 {
8 public:
9     B() { cout << "B ctor called for " << this << endl;}
10    void funk1() { cout << "B::funk1() called for " << this << endl; }
11    virtual void funk2() { cout << "B::funk2() called for " << this <<
    endl; }
12 };
13
14 class D : public B
15 {
16 public:
17     D() { cout << "D ctor called for " << this << endl;}
18     void funk1() { cout << "D::funk1() called for " << this << endl; }
19     virtual void funk2() { cout << "D::funk2() called for " << this <<
    endl; }
20 };
21
22 int main()
23 {
24     B b;
25     D d;
26     cout << endl;
27
28     b.funk1();
29     d.funk1();
30     cout << endl;
31
32     b.funk2();
33     d.funk2();
34     cout << endl;
35
36     B* pB;
37     pB = &b;
38     pB->funk1();
39     pB->funk2();
40     cout << endl;
41
42     pB = &d;
43     pB->funk1();
44     pB->funk2();
45
```

```
46     cout << endl;
47
48     cout << "size of b = " << sizeof b << endl;
49     cout << "size of d = " << sizeof d << endl;
50 }
```

***** 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.

```
1 // File: ex7-11.cpp
2
3 #include <iostream>
4 using namespace std;
5
6 class B
7 {
8 protected:
9     int b;
10 public:
11     B()
12     {
13         cout << "B ctor called for " << this << endl;
14         b = 0;
15     }
16     virtual void virt()
17     {
18         cout << "B::virt() called for " << this << endl;
19     }
20 };
21
22 class D : public B
23 {
24 protected:
25     int d;
26 public:
27     D()
28     {
29         cout << "D ctor called for " << this << endl;
30         d = 0;
31     }
32     void non_virt2()
33     {
34         cout << "D::non_virt2() called for " << this << endl;
35     }
36 };
37
38 int main()
39 {
40     B b;           // declare a base object
41     D d;           // declare a derived object
42
43     b.virt();      // invoke virt() through a base object
44     d.virt();      // invoke virt() through a derived object
45
46     B* pb;        // pb is a pointer to a base class object
```

```

47     pb = &b;           // pb points to b
48
49     pb->virt();        // invoke virt() through a base pointer
50     //   to a base object
51
52     pb = &d;           // pb points to d
53     pb->virt();        // invoke virt() through a base pointer
54     //   to a derived object
55
56     cout << "size of b = " << sizeof b << endl;
57     cout << "size of d = " << sizeof d << endl;
58     d.non_virt2();    // invoke non_virt2() through derived object
59 // pb->non_virt2(); Error: non_virt2() is not a member of B
60 }
61

```

***** 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.

```

1 // File: ex7-12.cpp
2
3 // This example shows that "virtualness" is passed down to derived
4 // classes
5 // even if the immediate "parent" class does not name a function as
6 // virtual.
7 // It also illustrates polymorphism implemented through references
8 // instead
9 // of pointers to base objects.
10
11 #include <iostream>
12 #include <string>
13 using namespace std;
14
15 class person

```

```
13 {
14 public:
15     virtual string who_am_i() const
16     {
17         return "person";
18     }
19     string non_virtual_who_am_i() const
20     {
21         return "non_virtual person";
22     }
23 };
24
25 class child : public person
26 {
27 public:
28     string who_am_i() const
29     {
30         return "child";
31     }
32     string non_virtual_who_am_i() const
33     {
34         return "non_virtual child";
35     }
36 };
37 class grand_child : public child
38 {
39 public:
40     string who_am_i() const
41     {
42         return "grand_child";
43     }
44     string non_virtual_who_am_i() const
45     {
46         return "non_virtual grand_child";
47     }
48 };
49
50 void identify_yourself(const person& p)
51 {
52     cout << "I am a " << (p.who_am_i()) << endl;
53     cout << "I am a " << (p.non_virtual_who_am_i()) << endl;
54 }
55
56 int main()
57 {
58     person P;
59     child C;
60     grand_child G;
61     person* pp;
62     pp = &P;
63     cout << (pp->who_am_i()) << endl;
64     cout << (pp->non_virtual_who_am_i()) << endl;
65     pp = &C;
```



```
66     cout << (pp->who_am_i()) << endl;
67     cout << (pp->non_virtual_who_am_i()) << endl;
68     pp = &G;
69     cout << (pp->who_am_i()) << endl;
70     cout << (pp->non_virtual_who_am_i()) << endl;
71     cout << "sizeof(person) = " << sizeof(person) << endl;
72     cout << "sizeof(child) = " << sizeof(child) << endl;
73     cout << "sizeof(grand_child) = " << sizeof(grand_child) << endl;
74     identify_yourself(P);
75     identify_yourself(C);
76     identify_yourself(G);
77 }
```

***** 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.

```
1 // File: ex7-13.cpp - Why a Virtual destructor?
2
3 #include <iostream>
4 using namespace std;
5
6 class X
7 {
8 public:
9     X()
10    {
11        cout << "X constructor called\n";
12    }
13    ~X()
14    {
15        cout << "X destructor called\n";
16    }
17 };
18
19
20 class A : public X
21 {
22 public:
23     A()
24     {
25         cout << "A constructor called\n";
26     }
27     ~A()
28     {
29         cout << "A destructor called\n";
30     }
31 };
32
33
34 int main()
35 {
36     X* ptrX;
37
38     ptrX = new X;
39     delete ptrX;
40
41     cout << endl;
42
43     ptrX = new A;
44     delete ptrX;
45 }
```

***** 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.

```
1 // File: ex7-14.cpp - Why a Virtual destructor? Here's why!
2
3 #include <iostream>
4 using namespace std;
5
6
7 class X
8 {
9 public:
10     X()
11     {
12         cout << "X constructor called\n";
13     }
14     virtual ~X()
15     {
16         cout << "X destructor called\n";
17     }
18 };
19
20
21 class A : public X
22 {
23 public:
24     A()
25     {
26         cout << "A constructor called\n";
27     }
28     ~A()
29     {
30         cout << "A destructor called\n";
31     }
32 };
33
34
35 int main()
36 {
37     X* ptrX;
38
39     ptrX = new X;
40     delete ptrX;
41
42     cout << endl;
43
44     ptrX = new A;
45     delete ptrX;
46 }
```

***** 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 virtual.
- 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 virtual functions.

Example 7-15 - Abstract classes and pure virtual functions

```
1 // File: ex7-15.cpp - Abstract classes
2
3 #include <iostream>
4 #include <cmath>
5 #include <cstdlib>
6 using namespace std;
7
8 const double pi = 3.141592654;
9
10 class Shape
11 {
12 protected:
13     double x;
14     double y;
15 public:
16     Shape(double = 0, double = 0);
17     double get_x() const
18     {
19         return x;
20     }
21     double get_y() const
22     {
23         return y;
24     }
25     virtual double area() const = 0;    // pure virtual function
26     virtual double girth() const = 0;  // pure virtual function
27 };
28
29 Shape::Shape(double c_x, double c_y) : x(c_x), y(c_y) {}
30
31 ostream& operator<<(ostream& out, const Shape& object)
32 {
33     cout << '(' << object.get_x() << ',' << object.get_y() << ')';
34     return out;
35 }
36
37 class Square : public Shape
38 {
39 private:
40     double side;
41 public:
42     Square(double c_x, double c_y, double s);
43     double get_side()
44     {
45         return side;
```

```
46     }
47     double area() const;
48     double girth() const;
49 };
50
51 Square::Square(double c_x, double c_y, double s) : Shape(c_x,c_y),
side(s)
52 { }
53
54 double Square::area() const
55 {
56     return side * side;
57 }
58
59 double Square::girth() const
60 {
61     return 4.*side;
62 }
63
64 class Triangle : public Shape
65 {
66 private:
67     double a,b,c; // length of 3 sides
68 public:
69     Triangle(double c_x,double c_y, double s1, double s2, double s3);
70     void print_sides();
71     double area() const;
72     double girth() const;
73 };
74
75 Triangle::Triangle(double c_x, double c_y, double s1, double s2, double
s3)
76     : Shape(c_x,c_y), a(s1), b(s2), c(s3)
77 { }
78
79 void Triangle::print_sides()
80 {
81     cout << a << ' ' << b << ' ' << c;
82 }
83
84 double Triangle::area() const
85 {
86     double s = (a + b + c) / 2.; // semiperimeter
87     return sqrt(s*(s-a)*(s-b)*(s-c));
88 }
89
90 double Triangle::girth() const
91 {
92     return a+b+c;
93 }
94
95 class Circle : public Shape
96 {
```



```
97 private:
98     double radius;
99 public:
100     Circle(double c_x, double c_y, double r);
101     double get_radius()
102     {
103         return radius;
104     }
105     double area() const;
106     double girth() const;
107 };
108
109 Circle::Circle(double c_x, double c_y, double r) : Shape(c_x,c_y),
    radius(r)
110 { }
111
112 double Circle::area() const
113 {
114     return pi*radius*radius;
115 }
116
117 double Circle::girth() const
118 {
119     return 2.*pi*radius;
120 }
121
122 int main()
123 {
124     // Shape sh(6,7); can't create instance of abstract class
125     Circle c(3,4,5);
126     cout << "Circle c - center: ";
127     cout << c << endl;
128     cout << "    radius = " << c.get_radius();
129     cout << "    area = " << c.area();
130     cout << "    circumference = " << c.girth() << endl;
131
132     Square s(5.,2.,1.);
133     cout << "Square s - center: ";
134     cout << s << endl;
135     cout << "    side = " << s.get_side();
136     cout << "    area = " << s.area();
137     cout << "    perimeter = " << s.girth() << endl;
138
139     Triangle t(0,0,3,4,5);
140     cout << "Triangle t - center: ";
141     cout << t << endl;
142     cout << "    sides = ";
143     t.print_sides();
144     cout << "    area = " << t.area();
145     cout << "    perimeter = " << t.girth() << endl;
146
147     cout << "sizeof(double)=" << sizeof(double) << endl;
148     cout << "sizeof(Shape)=" << sizeof(Shape) << endl;
```

```
149     cout << "sizeof(Square)=" << sizeof(Square) << endl;
150     cout << "sizeof(Triangle)=" << sizeof(Triangle) << endl;
151     cout << "sizeof(Circle)=" << sizeof(Circle) << endl;
152 }
```

***** 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.

```
1 // File: ex7-16.cpp - Life and polymorphism
2
3 #include <iostream>
4 #include <cstdlib>
5 using namespace std;
6
7 enum Bool { FALSE, TRUE};
8 enum LifeForm {VACANT, WEED, RABBIT, HAWK};
9
10 const int GridSize = 10;
11 const int Cycles = 10;
12 const int NumberLifeForms = 4;
13 const int HawkLifeExpectancy = 8;
14 const int HawkOvercrowdingLimit = 3;
15 const int RabbitLifeExpectancy = 3;
16
17 class Grid;
18
19 class LivingThing
20 {
21 protected:
22     int x,y;
23     void AssessNeighborhood(const Grid& G, int sm[]);
24 public:
25     LivingThing(int _x, int _y): x(_x), y(_y) {}
26     virtual ~LivingThing() {}
27     virtual LifeForm WhoAmI() const = 0;
28     virtual LivingThing* next(const Grid& G) = 0;
29 };
30
31 class Grid
32 {
33 private:
34     LivingThing* cell[GridSize][GridSize];
35 public:
36     Grid();
37     ~Grid()
38     {
39         if (cell[1][1]) release();
40     }
41     void update(Grid&);
42     void release();
43     void print();
44     LivingThing* get_cell(int row, int col) const;
45 };
46
47 /* This function counts the number of each LivingThing thing in
48 the neighborhood. A neighborhood is a square and the 8
49 adjacent squares on each side of it */
```

```
50 void LivingThing::AssessNeighborhood(const Grid& G, int count[])
51 {
52     int i, j;
53     count[VACANT] = count[WEED] = count[RABBIT] = count[HAWK] = 0;
54     for (i = -1; i <= 1; ++i)
55         for (j = -1; j <= 1; ++j)
56             count[G.get_cell(x+i,y+j) -> WhoAmI()]++;
57 }
58
59 LivingThing* Grid::get_cell(int row, int col) const
60 {
61     return cell[row][col];
62 }
63
64 class Vacant : public LivingThing
65 {
66 public:
67     Vacant(int _x, int _y):LivingThing(_x,_y) {}
68     LifeForm WhoAmI() const
69     {
70         return (VACANT);
71     }
72     LivingThing* next(const Grid& G);
73 };
74
75 class Weed : public LivingThing
76 {
77 public:
78     Weed(int _x, int _y): LivingThing(_x,_y) {}
79     LifeForm WhoAmI() const
80     {
81         return (WEED);
82     }
83     LivingThing* next(const Grid& G);
84 };
85
86 class Rabbit : public LivingThing
87 {
88 protected:
89     int age;
90 public:
91     Rabbit(int x, int y, int a = 0) : LivingThing(x,y), age(a) {}
92     LifeForm WhoAmI() const
93     {
94         return (RABBIT);
95     }
96     LivingThing* next(const Grid& G);
97 };
98
99 class Hawk : public LivingThing
100 {
101 protected:
102     int age;
```

```

103 public:
104     Hawk(int x, int y, int a = 0): LivingThing(x,y), age(a) {}
105     LifeForm WhoAmI() const
106     {
107         return (HAWK);
108     }
109     LivingThing* next(const Grid& G);
110 };
111
112 // This function determines what will be in an Vacant square in the
    next cycle
113 LivingThing* Vacant::next(const Grid& G)
114 {
115     int count[NumberLifeForms];
116     AssessNeighborhood(G,count);
117
118 // If there is more than one Rabbit in the neighborhood, a new Rabbit
119 //   is born.
120     if (count[RABBIT] > 1) return (new Rabbit(x,y));
121
122 // otherwise, if there is more than one Hawk, a Hawk will be born
123     else if (count[HAWK] > 1) return (new Hawk(x, y));
124
125 // otherwise, if there is Weed in the neighborhood, Weed will grow
126     else if (count[WEED]) return (new Weed(x, y));
127
128 // otherwise the square will remain Vacant
129     else return (new Vacant(x, y));
130 }
131
132 // if there is more Weeds than Rabbits, then new Weed will grow,
133 // otherwise Vacant
134 LivingThing* Weed::next(const Grid& G)
135 {
136     int count[NumberLifeForms];
137     AssessNeighborhood(G, count);
138     if (count[WEED] > count[RABBIT]) return (new Weed(x, y));
139     else return (new Vacant(x, y));
140 }
141
142 /* The Rabbit dies if:
143     there's more Hawks in the neighborhood than Rabbits
144     not enough to eat
145     or if it's too old
146     otherwise a new Rabbit is born */
147 LivingThing* Rabbit::next(const Grid& G)
148 {
149     int count[NumberLifeForms];
150     AssessNeighborhood(G, count);
151     if (count[HAWK] >= count[RABBIT] ) return (new Vacant(x, y));
152     else if (count[RABBIT] > count[WEED]) return (new Vacant(x, y));
153     else if (age > RabbitLifeExpectancy) return (new Vacant(x, y));
154     else return (new Rabbit(x,y, age + 1));

```

```
155 }
156
157 // Hawk die of overcrowding, starvation, or old age
158 LivingThing* Hawk::next(const Grid& G)
159 {
160     int count[NumberLifeForms];
161     AssessNeighborhood(G, count);
162     if (count[HAWK] > HawkOvercrowdingLimit) return (new Vacant(x, y));
163     else if (count[RABBIT] < 1) return (new Vacant(x,y));
164     else if (age > HawkLifeExpectancy) return (new Vacant (x, y));
165     else return (new Hawk(x, y, age + 1));
166 }
167
168 Grid::Grid()
169 {
170     LifeForm creature;
171     int i, j;
172     for (i = 0; i < GridSize; i++)
173         for (j = 0; j < GridSize; j++)
174             {
175                 if (i == 0 || i == GridSize - 1 || j ==0 || j == GridSize -
176 1)
177                     creature = VACANT;
178                 else
179                     creature = LifeForm(rand() % NumberLifeForms);
180                 switch (creature)
181                 {
182                     case HAWK:
183                         cell[i][j] = new Hawk(i,j);
184                         break;
185                     case RABBIT:
186                         cell[i][j] = new Rabbit(i,j);
187                         break;
188                     case WEED:
189                         cell[i][j] = new Weed(i,j);
190                         break;
191                     case VACANT:
192                         cell[i][j] = new Vacant(i,j);
193                 }
194             }
195
196 void Grid::release()
197 {
198     int i, j;
199     for (i = 1; i < GridSize - 1; ++i)
200         for (j = 1; j < GridSize - 1; ++j) delete cell[i][j];
201     cell[1][1] = 0;
202 }
203
204 void Grid::update(Grid& old)
205 {
206     int i, j;
```

```
207     for (i = 1; i < GridSize - 1; ++i)
208         for (j = 1; j < GridSize - 1; ++j)
209             cell[i][j] = old.cell[i][j] -> next(old);
210 }
211
212 void Grid::print()
213 {
214     LifeForm creature;
215     int i, j;
216     for (i = 1; i < GridSize - 1; i++)
217     {
218         for (j = 1; j < GridSize - 1; j++)
219         {
220             creature = cell[i][j]->WhoAmI();
221             switch (creature)
222             {
223                 case HAWK:
224                     cout << "H";
225                     break;
226                 case RABBIT:
227                     cout << "R";
228                     break;
229                 case WEED:
230                     cout << "W";
231                     break;
232                 case VACANT:
233                     cout << "0";
234             }
235         }
236         cout << endl;
237     }
238     cout << endl;
239 }
240
241 int main()
242 {
243     Grid G1, G2;
244     G1.print();
245
246     for (int i = 1; i <= Cycles; i++)
247     {
248         cout << "Cycle " << i << endl;
249         if (i % 2)
250         {
251             G2.update(G1);
252             G2.print();
253             G1.release();
254         }
255         else
256         {
257             G1.update(G2);
258             G1.print();
259             G2.release();
```

```
260         }  
261     }  
262 }
```


***** Output *****

WHROW0RR
 ROWWWHWH
 HRH0HORW
 ORWORHHR
 HRW0HHR
 HHRWW0WH
 W0HORWWH
 HWWROHRR

Cycle 1
 0H0WWW00
 0R0WWH0H
 H0HHHR00
 R00H0000
 H00RH000
 H00WWHWH
 WHHRRWWH
 0WWORH00

Cycle 2
 0HWWWWH0
 H0HWWHH0
 HRH000H0
 0HHORH00
 HHW0HHHW
 0HR00HW0
 W0HRRWW0
 WWWR0HHW

Cycle 3
 H0WWWW0H
 HHHWW00H
 H00HHH0H
 H00H00HH
 000R000W
 H00RR0WW
 WHH00WWW
 WW00RHOW

Cycle 4
 0HWWWWH0
 000WWHH0
 0HH000H0
 0HHHHH00
 HHR0RWHW
 0HR00WWW
 W0HRRWWW
 WHH00HWW

Cycle 5
 00WWWW0H

HHHWW00H
 H00HHH0H
 H000H0HH
 000ROW0W
 H00RRWWW
 WHH0RWWW
 WWHRRHWW

Cycle 6
 HHWWWWH0
 000WWHH0
 0HH000H0
 0H0H0H00
 HHR0RWHW
 0HR00WWW
 W0HROWWW
 WWH00HWW

Cycle 7
 00WWWW0H
 HHHWW00H
 H00HHH0H
 H0HHHHHH
 000ROW0W
 H00RRWWW
 WHH0WWW
 WWHHW0WW

Cycle 8
 HHWWWWH0
 000WWHH0
 0HH000H0
 0HH00000
 HHR0RWHW
 0HR0RWWW
 W00RWWW
 WW00WWW

Cycle 9
 00WWWW0H
 HHHWW00H
 H00HWH0H
 H00RWHHH
 000RW0W
 HH0RRWWW
 WWROWWWW
 WWWWWW

Cycle 10
 HHWWWWH0
 000WWWWH0

OHHHWOHO
OHR00H00
HHR00WHW
OHR0RWWW
WWR0WWW0
WWWWW0

