C++

PROGRAMMING LANGUAGE

L11-POLYMORPHISM

Mohammad Shaker
mohammadshaker.com
@ZGTRShaker
2010, 11, 12, 13, 14

Polymorphism

Polymorphism

- Polymorphism
  - “Multi-form”
  - Enables us to program “in general”
  - Write program that process object of classes that are part of the same class hierarchy as if they are all objects of the hierarchy's base class
  - Makes programs extensible
    - Add new classes (new classes to the hierarchy) with no modification

Polymorphism

```
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point
#endif
```

```
#include "point.h"   // Point class definition

Point::Point( int xValue, int yValue ): x( xValue ), y( yValue ){}

void Point::setX( int xValue ){   x = xValue; }
int Point::getX() const{   return x;}
void Point::setY( int yValue ){   y = yValue;}
int Point::getY() const{   return y;}

void Point::print() const
{   cout << '[' << getX() << ", " << getY() << ']';}
```

Polymorphism

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h”
class Circle: public Point {
public:
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;        // return diameter
   double getCircumference() const;   // return circumference
   double getArea() const;            // return area

   void print() const;          // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

```
#include "circle.h"    // Circle class definition

Circle::Circle( int xValue, int yValue, double radiusValue )
: Point( xValue, yValue )  // call base-class constructor
{   setRadius( radiusValue );}

void Circle::setRadius( double radiusValue )
{   radius = ( radiusValue < 0.0? 0.0: radiusValue );}

double Circle::getRadius() const{   return radius;}

double Circle::getDiameter() const{   return 2 * getRadius();}

double Circle::getCircumference() const{   return 3.14159 * getDiameter();}

double Circle::getArea() const
{   return 3.14159 * getRadius() * getRadius();}

void Circle::print() const
{  cout << "center = ";
   Point::print();
   cout << "; radius = " << getRadius();
}
```

Polymorphism

```
void main()
{
   Point point( 30, 50 );    Point *pointPtr = 0;     // base-class pointer
   Circle circle( 120, 89, 2.7 );   Circle *circlePtr = 0;  // derived-class pointer

   // set floating-point numeric formatting
   cout << fixed << setprecision( 2 );
   // output objects point and circle
   cout << "Print point and circle objects:” << "\nPoint: ";
   point.print();   // invokes Point's print
   cout << "\nCircle: ";
   circle.print();  // invokes Circle's print

   pointPtr = &point;
   cout << "\n\nCalling print with base-class pointer to “ << "\nbase-class object invokes base-class print “<< "function:\n";
   pointPtr->print();  // invokes Point's print

   circlePtr = &circle;
   cout << "\n\nCalling print with derived-class pointer to “  << "\nderived-class object invokes derived-class “ << "print function:\n";
   circlePtr->print();  // invokes Circle's print

   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling print with base-class pointer to “  << "derived-class object\ninvokes base-class print “ <<
           "function on that derived-class object\n";
   pointPtr->print();  // invokes Point's print
   cout << endl;
}
```

```
Print point and circle objects:
Point: [30, 50]
Circle: center = [120, 89]; radius = 2.7

Calling print with base-class pointer to
base-class object invokes base-class print function:
[30, 50]

Calling print with derived-class pointer to
derived-class object invokes derived-class print function:
center = [120, 89]; radius = 2.7

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object
[120, 89]
Press any key to continue
```

Polymorphism

```
void main()
{
   Point point( 30, 50 );   Point *pointPtr = 0;     // base-class pointer
   Circle circle( 120, 89, 2.7 );   Circle *circlePtr = 0;  // derived-class pointer

   // set floating-point numeric formatting
   cout << fixed << setprecision( 2 );
   // output objects point and circle
   cout << "Print point and circle objects:” << "\nPoint: ";
   point.print();   // invokes Point's print
   cout << "\nCircle: ";
   circle.print();  // invokes Circle's print

   pointPtr = &point;
   cout << "\n\nCalling print with base-class pointer to “ << "\nbase-class object invokes base-class print “<< "function:\n";
   pointPtr->print();  // invokes Point's print

   circlePtr = &circle;
   cout << "\n\nCalling print with derived-class pointer to “  << "\nderived-class object invokes derived-class “ << "print function:\n";
   circlePtr->print();  // invokes Circle's print

   // aim base-class pointer at derived-class object and print
   pointPtr = &circle;
   cout << "\n\nCalling print with base-class pointer to “  << "derived-class object\ninvokes base-class print “ <<
           "function on that derived-class object\n";
   pointPtr->print();  // invokes Point's print
   circlePtr = &point;
}
```

Compiler error, `circlePtr = &point;` a point is not a circle!

Polymorphism

```
// Fig. 10.6: fig10_06.cpp
// Aiming a derived-class pointer at a base-class object.
#include "point.h"    // Point class definition
#include "circle.h"   // Circle class definition

int main()
{
   Point point( 30, 50 );
   Circle *circlePtr = 0;

   // aim derived-class pointer at base-class object
   circlePtr = &point;  // Error: a Point is not a Circle

   return 0;

} // end main
```

Compiler error, `circlePtr = &point;` a point is not a circle!

Polymorphism

```
#include <iostream>
using namespace::std;

#include"xpoint.h"
#include"xcircle.h"


int main()
{
   Point *pointPtr = 0;
   Circle circle( 120, 89, 2.7 );

   // aim base-class pointer at derived-class object
   pointPtr = &circle;

   // invoke base-class member functions on derived-class
   // object through base-class pointer
   int x = pointPtr->getX();
   int y = pointPtr->getY();
   pointPtr->setX( 10 );
   pointPtr->setY( 10 );
   pointPtr->print();

} // end main
```

```
[10, 10]
Press any key to continue
```

Polymorphism

```
int main()
{
   Point *pointPtr = 0;
   Circle circle( 120, 89, 2.7 );

   // aim base-class pointer at derived-class object
   pointPtr = &circle;

   // invoke base-class member functions on derived-class
   // object through base-class pointer
   int x = pointPtr->getX();
   int y = pointPtr->getY();
   pointPtr->setX( 10 );
   pointPtr->setY( 10 );
   pointPtr->print();

   // attempt to invoke derived-class-only member functions
   // on derived-class object through base-class pointer
   double radius = pointPtr->getRadius();
   pointPtr->setRadius( 33.33 );
   double diameter = pointPtr->getDiameter();
   double circumference = pointPtr->getCircumference();
   double area = pointPtr->getArea();

   return 0;

} // end main
```

```
Compiler error
```

Polymorphism - Summary

- Base-Pointer can aim at derived class
  - But can only call base-class functions
  - Calling derived class functions is a compiler error
    - Coz functions are not defined at base class

RULE

THE TYPE OF POINTER/REFERENCE
DETERMINES FUNCTION TO CALL

`virtual` functions

`virtual` functions

- The type of object, not pointer, determines functions to call
- Why this is so good? suppose the following:
  - draw function in base class
    - Shape
  - draw functions in all derived class
    - Circle, triangle, rectangle

- Now, to draw any shape:
  - Have base class shape pointer, call member function draw
  - Treat all these shapes generically as object of the base class shape
  - Program determines proper draw function at run time dynamically based on the type of the object to which the base class shape pointer points to at any given time.
  - Declare draw virtual in the base class
  - Override draw in each base class (must have the same signature)
  - Declare draw virtual in all derived class
  - But that's not even necessary.

RULE WHEN USING `virtual`

THE TYPE OF THE OBJECT, NOT THE POINTER,
DETERMINES FUNCTION TO CALL

The Point Class Was Looking Like this:

```
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point
#endif
```

```
#include "point.h"   // Point class definition

Point::Point( int xValue, int yValue ): x( xValue ), y( yValue ){}

void Point::setX( int xValue ){   x = xValue; }
int Point::getX() const{   return x;}
void Point::setY( int yValue ){   y = yValue;}
int Point::getY() const{   return y;}

void Point::print() const
{   cout << '[' << getX() << ", " << getY() << ']';}
```

Now, we use `virtual` functions

```
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   virtual void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point
#endif
```

```
#include "point.h"   // Point class definition

Point::Point( int xValue, int yValue ): x( xValue ), y( yValue ){}

void Point::setX( int xValue ){   x = xValue; }
int Point::getX() const{   return x;}
void Point::setY( int yValue ){   y = yValue;}
int Point::getY() const{   return y;}

void Point::print() const
{   cout << '[' << getX() << ", " << getY() << ']';}
```

Just here

The Same in Circle Class, Which Was Looking Like this:

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h”
class Circle: public Point {
public:
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;        // return diameter
   double getCircumference() const;   // return circumference
   double getArea() const;            // return area

   void print() const;          // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

```
#include "circle.h"    // Circle class definition

Circle::Circle( int xValue, int yValue, double radiusValue )
: Point( xValue, yValue )  // call base-class constructor
{   setRadius( radiusValue );}

void Circle::setRadius( double radiusValue )
{   radius = ( radiusValue < 0.0? 0.0: radiusValue );}

double Circle::getRadius() const{   return radius;}

double Circle::getDiameter() const{   return 2 * getRadius();}

double Circle::getCircumference() const{   return 3.14159 * getDiameter();}

double Circle::getArea() const
{   return 3.14159 * getRadius() * getRadius();}

void Circle::print() const
{  cout << "center = ";
   Point::print();
   cout << "; radius = " << getRadius();
}
```

Now, we use `virtual` functions

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h”
class Circle: public Point {
public:
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;        // return diameter
   double getCircumference() const;   // return circumference
   double getArea() const;            // return area

   virtual void print() const;          // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

```
#include "circle.h"    // Circle class definition

Circle::Circle( int xValue, int yValue, double radiusValue )
: Point( xValue, yValue )  // call base-class constructor
{   setRadius( radiusValue );}

void Circle::setRadius( double radiusValue )
{   radius = ( radiusValue < 0.0? 0.0: radiusValue );}

double Circle::getRadius() const{   return radius;}

double Circle::getDiameter() const{   return 2 * getRadius();}

double Circle::getCircumference() const{   return 3.14159 * getDiameter();}

double Circle::getArea() const
                                Radius() * getRadius();}

                                st
{   cout << "center = ";
   Point::print();
   cout << "; radius = " << getRadius();
}
```

Just here

virtual functions

```
int main()
{
   Point point( 30, 50 );   Point *pointPtr = 0;
   Circle circle( 120, 89, 2.7 );   Circle *circlePtr = 0;
   // output objects point and circle using static binding
   cout << "Invoking print function on point and circle "
   << "\nobjects with static binding "        << "\n\nPoint: ";
   point.print();          // static binding
   cout << "\nCircle: ";
   circle.print();         // static binding

   cout << "\n\nInvoking print function on point and circle "
   << "\nobjects with dynamic binding";

   pointPtr = &point;
   cout << "\n\nCalling virtual function print with base-class" << "\npointer to base-cl
function:\n";
   pointPtr->print();

   circlePtr = &circle;
   cout << "\n\nCalling virtual function print with "    << "\nderived-class pointer to derived-class object "    << "\ninvokes derived-
class print function:\n";
   circlePtr->print();

   pointPtr = &circle;
   cout << "\n\nCalling virtual function print with base-class"
        << "\npointer to derived-class object "
        << "\ninvokes derived-class print function:\n";
   pointPtr->print();  // polymorphism: invokes circle's print}
```

```
Invoking print function on point and circle
objects with static binding

Point: [30, 50]
Circle: Center = [120, 89]; Radius = 2.70

Invoking print function on point and circle
objects with dynamic binding

Calling virtual function print with base-class
pointer to base-class object
invokes base-class print function:
[30, 50]

Calling virtual function print with
derived-class pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70

Calling virtual function print with base-class
pointer to derived-class object
invokes derived-class print function:
Center = [120, 89]; Radius = 2.70
Press any key to continue
```

```
[30, 50]
center = [120, 89]; radius = 2.7
[30, 50]
center = [120, 89]; radius = 2.7
center = [120, 89]; radius = 2.7

Press any key to continue
```

REMEMBER, REMEMBER THE RULE WHEN USING `virtual`

THE TYPE OF THE OBJECT, NOT THE POINTER, DETERMINES FUNCTION TO CALL

`virtual` in base class

not `virtual` at child class

What happens?

virtual in base class, not virtual at child class

```
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   virtual void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
class Circle: public Point {
public:
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;        // return diameter
   double getCircumference() const;   // return circumference
   double getArea() const;            // return area

            void print() const;          // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Just here

Nothing here

virtual functions

```
int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;

   point.print();       cout << endl;      // static binding
   circle.print();     cout << endl;       // static binding

   pointPtr = &point;
   pointPtr->print();   cout << endl;

   circlePtr = &circle;
   circlePtr->print();   cout << endl;

   pointPtr = &circle;
   pointPtr->print();    cout << endl;
   cout << endl;

   return 0;

} // end main
```

```
[30, 50]
center = [120, 89]; radius = 2.7
[30, 50]
center = [120, 89]; radius = 2.7
center = [120, 89]; radius = 2.7

Press any key to continue
```

The same behavior as before, that means virtual funtions are directly virtual in child classes once declared virtual in base class

`virtual` declaration

`virtual` funtions are directly

`virtual` in child classes

once declared

`virtual` in base class

Virtual Destructors

Virtual Destructors

- If an object with non-virtual destructor is destroyed explicitly, by applying the delete operator to a base class pointer to the object
  - The behavior is un-expected
- How to fix this?
  - Declare base-class destructor virtual
  - Making all derived class virtual as well (that's not necessary as we have seen)
  - Now, when deleting, the appropriate destructor will be called based on the type on object the base class pointer points to.
- Note
  - Constructors can't be virtual.

Virtual Destructors

```
#include <iostream>
using namespace::std;

class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };

};


class Son: public Dad
{
public:
            Son(){cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;

void main()
{
            Son S;

}
```

```
I'm constructor of dad
I'm constructor of son
I'm destructor of son
I'm destructor of dad
Press any key to continue
```

Virtual Destructors

```
#include <iostream>
using namespace::std;

class Dad
{
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };


};


class Son: public Dad
{
            Son(){cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;

void main()
{
            Son S;

}
```

Compile error. Declaring constructor and desrtuctors private in base class

`dynamic_cast` keyword

dynamic_cast keyword

```
#include <iostream>
using namespace::std;

class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son(){cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D;
            Son S;
            cout << "_____\n";
            Son *SonPtr =&S;
            Dad *DadPtr =&D;
            Dad *Person = dynamic_cast <Dad *> (DadPtr);
            if (Person)
            // means: if Person points to an object of type Dad
            {                       cout << "It works!\n";          }
            else
            {                       cout <<"Not working!\n";        }
}
```

```
I'm constructor of dad
I'm constructor of dad
I'm constructor of son

_____
It works!
I'm destructor of son
I'm destructor of dad
I'm destructor of dad
Press any key to continue
```

dynamic_cast keyword

```
#include <iostream>
using namespace::std;

class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son(){cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D;
            Son S;
            cout << "_____________________\n";
            Son *SonPtr =&S;
            Dad *DadPtr =&D;
            Dad *Person = dynamic_cast <Son *> (DadPtr);
            if (Person)
            {                       cout << "It works!\n";          }
            else
            {                       cout << "Not working!\n";       }
}
```

```
Compiler error
```

`typeid` Keyword

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad;
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
}
```

```
I'm constructor of dad
I'm constructor of dad
class Dad
class Dad *
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad;
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
            delete D1;
}
```

```
Compiler error Delete D1
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad;
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
            delete D2;
}
```

```
I'm constructor of dad
I'm constructor of dad
class Dad
class Dad *
I'm destructor of dad
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad();
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
            delete D2;
}
```

```
I'm constructor of dad
I'm constructor of dad
class Dad
class Dad *
I'm destructor of dad
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
            delete D2;
}
```

```
I'm constructor of dad
class Dad
class Dad *
Press any key to continue
```

Then runtime error Delete with no “new”

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            cout << typeid(D1).name() << endl;
            cout << typeid(D2).name() << endl;
}
```

```
I'm constructor of dad
class Dad
class Dad *
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Son S1;
            Son *S2 = new Dad();
            cout << typeid(S1).name() << endl;
            cout << typeid(S2).name() << endl;
}
```

```
Compiler error
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            cout << typeid(D1).name() << endl;
            cout << typeid(*D2).name() << endl;
}
```

```
I'm constructor of dad
class Dad
class Dad
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            cout << typeid(Dad).name() << endl;
}
```

```
I'm constructor of dad
class Dad
I'm destructor of dad
Press any key to continue
```

typeid Keyword

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;

void main()
{
Dad D1;
Dad *D2;
typeid(Dad)=typeid(D2)?
            cout <<"Yeah\n": cout << "No\n";
}
```

```
Compiler error
```

Quiz, Small One

Quiz #1

```
#ifndef POINT_H
#define POINT_H

class Point {

public:
   Point( int = 0, int = 0 ); // default constructor

   void setX( int );  // set x in coordinate pair
   int getX() const;  // return x from coordinate pair

   void setY( int );  // set y in coordinate pair
   int getY() const;  // return y from coordinate pair

   virtual void print() const;  // output Point object

private:
   int x;  // x part of coordinate pair
   int y;  // y part of coordinate pair

}; // end class Point

#endif
```

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "point.h"
class Circle: public Point {
public:
   Circle( int = 0, int = 0, double = 0.0 );

   void setRadius( double );    // set radius
   double getRadius() const;    // return radius

   double getDiameter() const;        // return diameter
   double getCircumference() const;   // return circumference
   double getArea() const;            // return area

   virtual void print() const;          // output Circle object

private:
   double radius;  // Circle's radius

}; // end class Circle

#endif
```

Quiz #1

```
int main()
{
   Point point( 30, 50 );
   Point *pointPtr = 0;

   Circle circle( 120, 89, 2.7 );
   Circle *circlePtr = 0;
   {
            static Point P(20,40);
   }
   point.print();        cout << endl;
   circle.print();      cout << endl;

   pointPtr = &circle;
   pointPtr->print();    cout << endl;

   circlePtr = &circle;
   circlePtr->print();    cout << endl;

   pointPtr->print();     cout << endl;
   cout << endl;

   return 0;
} // end main
```

```
WaaaWeee Constructor of Point!
WaaaWeee Constructor of Point!
WaaaWeee Constructor of Cricle!
WaaaWeee Constructor of Point!
[30, 50]
center = [120, 89]; radius = 2.7
center = [120, 89]; radius = 2.7
center = [120, 89]; radius = 2.7
center = [120, 89]; radius = 2.7

WaaaWeee Destructor of Cricle!
WaaaWeee Destructor of Point!
WaaaWeee Destructor of Point!
WaaaWeee Destructor of Point!
Press any key to continue
```

Quiz #2

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            typeid(Dad)==typeid(D1)?
cout<<"Yeah\n":cout<<"No\n";
}
```

```
I'm constructor of dad
Yeah
I'm destructor of dad
Press any key to continue
```

Quiz #3

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            typeid(Dad)==typeid(D2)? cout
<<"Yeah\n": cout << "No\n";
}
```

```
I'm constructor of dad
No
I'm destructor of dad
Press any key to continue
```

Quiz #4

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            Son *S1 = new Son();
            D2 = &(S1);
            cout << typeid(D2).name() << endl;
}
```

```
Compiler error D2 = &(S1);
```

Quiz #5

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            Son *S1 = new Son;
            D2 = &(*S1);
            cout << typeid(D2).name() << endl;
}
```

```
I'm constructor of dad
I'm constructor of dad
I'm constructor of son
class Dad *
I'm destructor of dad
Press any key to continue
```

Quiz #6

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2;
            Son *S1;
            D2 = &(*S1);
            cout << typeid(D2).name() << endl;
}
```

```
Runtime error
```

Quiz #7

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad ();
            Son *S1 = new Son;
            D2 = &(*S1);
            cout << typeid(D2).name() << endl;
            delete Dad;
}
```

```
Compiler error delete Dad;
```

Quiz #8

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Dad *D2 = new Dad ();
            Son *S1 = new Son;
            D2 = &(*S1);
            cout << typeid(D2).name() << endl;
            delete S1;
}
```

```
I'm constructor of dad
I'm constructor of dad
I'm constructor of dad
I'm constructor of son
class Dad *
I'm destructor of son
I'm destructor of dad
I'm destructor of dad
Press any key to continue
```

Quiz #9

```
#include <iostream>
using namespace::std;



class Dad
{
public:
            Dad(){cout << "I'm constructor of dad " << endl; };
            ~Dad(){cout << "I'm destructor of dad " << endl; };
};


class Son: public Dad
{
public:
            Son() {cout << "I'm constructor of son " << endl; }
            ~Son() {cout << "I'm destructor of son " << endl; }
};
```

```
#include <iostream>
#include "Testing.h"
using namespace::std;


void main()
{
            Dad D1;
            Son *S1 = new Son;
            Dad *D2 = &(*(&(*S1)));
            cout << typeid(D2).name() << endl;
}
```

```
I'm constructor of dad
I'm constructor of dad
I'm constructor of son
class Dad *
I'm destructor of dad
Press any key to continue
```

Keep in touch and let's connect


http://www.mohammadshaker.com


mohammadshakergtr@gmail.com


http://mohammadshakergtr.wordpress.com/


https://de.linkedin.com/pub/mohammad-shaker/30/122/128/


https://twitter.com/ZGTRShaker @ZGTRShaker


http://www.slideshare.net/ZGTRZGTR


https://www.goodreads.com/user/show/11193121-mohammad-shaker


https://plus.google.com/u/0/+MohammadShaker/


https://www.youtube.com/channel/UCvJUfadMoEaZNWdagdMyCRA