8. Program Set 6

1. Encapsulation and public interface – program increments a counter

#include <iostream>

class Counter {

    int value;                 // hidden (data hiding)

public:

    Counter() : value(0)

    {}

    void inc()

    {

        ++value;

    }    // public interface

    int get() const

    {

        return value;

    }

};

int main()

{

    Counter c;

    c.inc();

    std::cout << c.get() << std::endl;

    return 0;

}

2. has-a vs is-a, has-a here means an instance of Engine class is declared in Car class, meaning semantically that Car contains Engine (correct). is-a here would have meant that Car class inherits Engine class, semantically meaning Car is another form of Engine (incorrect). If is-a was used here, then the program would have worked, but is wrong in meaning.

#include <iostream>

#include <string>

class Engine {

    int hp;

public:

    Engine(int h):hp(h)

    {}

    int getHP() const

    {

        return hp;

    }

};

class Car {                 // Car HAS an Engine (has-a)

    Engine engine;

    std::string model;

public:

    Car(std::string m,int hp):engine(hp),model(m)

    {}

    void spec() const

    {

        std::cout << model << " " << engine.getHP() << "HP" << std::endl;

    }

};

int main()

{

    Car c("Coupe", 180);

    c.spec();

    return 0;

}

3. Inheritance in C++ classes

#include <iostream>

#include <string>

class Vehicle {

protected:

    std::string license; int year;

public:

    Vehicle(const std::string& L,int Y):license(L),year(Y)

    {}

    std::string getDesc() const

    {

        return license + " from " + (year<0?"?":(std::string("")+char('0'+(year/1000%10)))) ;

    } // tiny stub

    const std::string& getLicense() const

    {

        return license;

    }

    int getYear() const

    {

        return year;

    }

};

class Car : public Vehicle {     // Car IS A Vehicle

    std::string style;

public:

    Car(const std::string& L,int Y,const std::string& S): Vehicle(L,Y),style(S)

    {}

    const std::string& getStyle() const

    {

        return style;

    }

};

int main()

{

    Car c("MIT-007", 2, "sedan");

    std::cout << c.getLicense() << std::endl;

    return 0;

}

4. Overriding a class method

#include <iostream>

#include <string>

class Vehicle {

protected:

    std::string license;

    int year;

public:

    Vehicle(const std::string& L,int Y):license(L),year(Y)

    {}

    std::string getDesc() const

    {

        return license + " (" + "Vehicle" + ")";

    }

};

class Car : public Vehicle {

    std::string style;

public:

    Car(const std::string& L,int Y,const std::string& S):Vehicle(L,Y),style(S)

    {}

    std::string getDesc() const

    {

        return style + ": " + license;

    } // overrides

};

int main()

{

    Car c("MIT-123", 2003, "hatch");

    std::cout << c.getDesc() << std::endl;

    return 0;

}

5. protected class members are accessible if inherited as public in derived class, but protected members are not accessible anywhere else.

#include <iostream>

#include <string>

class Vehicle {

protected:

    std::string license;

    int year; // visible to derived, not to users

public:

    Vehicle(const std::string& L,int Y):license(L),year(Y)

    {}

};

class Car : public Vehicle {

public:

    Car(const std::string& L,int Y):Vehicle(L,Y)

    {}

    void reregister(const std::string& L)

    {

        license = L;

    } // allowed: protected

};

int main()

{

    Car c("ABC", 1999);

    c.reregister("XYZ");

    std::cout<<c.license; // ERROR

    return 0;

}

6. Usage of virtual in base class so that overloaded methods in derived classes are executed instead of the same method names in base class. A function is defined in the derived class as ‘std::string getDesc() const’. The const means that the getDesc() function cannot modify the object.

#include <iostream>

#include <string>

class Vehicle {

public:

    virtual ~Vehicle()

    {}                  // virtual destructor

    virtual std::string getDesc() const

    {

        return "Vehicle";

    }

};

class Car : public Vehicle {

public:

    std::string getDesc() const

    {

        return "Car";

    }

};

int main(){

    Car c;

    Vehicle\* vp = &c;           // base ptr to derived obj

    std::cout << vp->getDesc() << std::endl;

// prints "Car" because virtual

    return 0;

}

7. The effect of not using virtual in base class to dynamically dispatch overloaded functions in derived class.

#include <iostream>

#include <string>

class Vehicle {

public:

    std::string getDesc() const

    {

        return "Vehicle";

    } // NOT virtual

};

class Car : public Vehicle {

public:

    std::string getDesc() const

    {

        return "Car";

    }

};

int main(){

    Car c;

    Vehicle\* vp = &c;

    std::cout << vp->getDesc() << std::endl; // "Vehicle"

    return 0;

}

8. Virtual function behaviour works correctly with references

#include <iostream>

#include <string>

class Vehicle {

public:

virtual ~Vehicle()

{}

virtual std::string getDesc() const

{

return "Vehicle";

}

};

class Car : public Vehicle {

public:

std::string getDesc() const

{

return "Car";

}

};

void print(const Vehicle& v)

{

std::cout << v.getDesc() << std::endl; // virtual works

}

int main(){

Car c;

print(c);

Vehicle &p = c;

std::cout << "Vehicle reference in main: " << p.getDesc() << std::endl;

return 0;

}

9. Abstract classes

#include <iostream>

#include <string>

class Vehicle { // Abstract class

public:

int m;

virtual std::string getDesc() const = 0; // pure virtual

};

class Car : public Vehicle {

public:

Car (int m\_)

{

m = m\_;

}

std::string getDesc() const

{

return "Car";

}

int getM () const

{

return m;

}

};

int main(){

/\* Vehicle v; // ERROR: abstract \*/

Car c(3);

std::cout<<c.getDesc()<< " " << c.getM() << std::endl;

return 0;

}

10. Programming by difference – meaning reuse the code in the base class using scope resolution operator as shown below. In the below example, the scope resolution operator is used as Vehicle::getDesc().

#include <iostream>

#include <string>

class Vehicle {

protected:

    std::string license;

    int year;

public:

    Vehicle(const std::string& L,int Y):license(L),year(Y)

    {}

    virtual std::string getDesc() const

    {

        return license;

    }

};

class Car : public Vehicle {

    std::string style;

public:

    Car(const std::string& L,int Y,const std::string& S):Vehicle(L,Y),style(S)

    {}

    std::string getDesc() const

    {

        return style + ": " + Vehicle::getDesc(); // add to base

    }

};

int main(){

    Car c("MIT-999", 2010, "sedan");

    std::cout<<c.getDesc()<< std::endl;

    return 0;

}

11. Public and protected inheritance

#include <iostream>

class Base {

public:

    void f()

    {}

};

class Pub : public Base

{};       // f() stays public

class Pro : protected Base

{};    // f() becomes at most protected

int main(){

    Pub a;

    a.f(); // Allowed

    Pro b;

    b.f(); // ERROR: now protected \*/

    return 0;

}

12. virtual destructors

#include <iostream>

class Base {

public:

    // try toggling virtual on/off here

    virtual ~Base()

    {

        std::cout << "~Base\n";

    }

    virtual void f()

    {}

};

class Derived : public Base {

    int\* big\_;

public:

    Derived() : big\_(new int[1000])

    {}

    ~Derived()

    {

        std::cout << "~Derived\n";

delete[] big\_;

    }

};

int main() {

    Base\* p = new Derived;

    delete p;  // needs virtual ~Base() first to call ~Derived()

}

13. pass by value slicing

#include <iostream>

class Vehicle {

public:

    virtual ~Vehicle()

    {}

    virtual void id() const {

        std::cout<<"V\n";

    }

};

class Car : public Vehicle {

public:

    void id() const {

        std::cout<<"C\n";

    }

};

void show(Vehicle v){

    v.id();      // pass by value: slices Car part off!

}

void showRef(Vehicle& v){

    v.id();      // pass by reference: uses Car class members

}

int main(){

    Car c;

    show(c); // prints "V"

    showRef(c); // prints "C"

    return 0;

}

14. Overriding a base class function and at the same time calling the exact base class function

#include <iostream>

class A {

public:

    virtual ~A()

    {}

    virtual void f() const

    {

        std::cout<<"A" << std::endl;

    }

};

class B : public A {

public:

    void f() const {

         std::cout<<"B:"; A::f();

    }

};

int main(){

    B b;

    A\* p = &b;

    p->f(); // prints "B:A"

    return 0;

}

15. One base class pointer pointing to multiple derived classes

#include <iostream>

class Shape {

public:

    virtual ~Shape()

    {}

    virtual double area() const = 0;

};

class Rect : public Shape {

    double w, h;

public:

    Rect(double W,double H) : w(W), h(H)

    {}

    double area() const {

        return w \* h;

    }

};

class Tri  : public Shape {

    double b, h;

public:

    Tri(double B,double H) : b(B), h(H)

    {}

    double area() const {

        return 0.5 \* b \* h;

    }

};

int main(){

    Rect r(3, 4);

    Tri t(3, 4);

    Shape\* s[2] = { &r, &t};

    std::cout << s[0]->area() << " "<< s[1]->area() << std::endl;

    return 0;

}

16. Multiple inheritance

#include <iostream>

class InsuredItem {

public:

    virtual ~InsuredItem()

    {}

    virtual void policy() const {

        std::cout<<"Policy" << std::endl;

    }

};

class Vehicle {

public:

    virtual ~Vehicle()

    {}

    virtual void info()   const {

        std::cout<<"Vehicle" << std::endl;

    }

};

class Car : public Vehicle, public InsuredItem {

public:

    void info() const {

        std::cout<<"Car" << std::endl;

    }

};

int main(){

    Car c;

    c.info();

    c.policy();

    Vehicle &v = c;

    v.info();

    return 0;

}

17. Multiple inheritance with ambiguous name resolution

#include <iostream>

class A{

public:

    void f() const {

        std::cout<<"A" << std::endl;

    }

};

class B{

public:

    void f() const {

        std::cout<<"B" << std::endl;

    }

};

class C: public A, public B {

public:

    void callA() const {

        A::f();

    }

};

int main(){

    C c;

    c.callA();

    c.f(); // ERROR: ambiguous; need A::f or B::f

    return 0;

}