# Runtime Type

class Point {

public:

int x;

int y;

void sayHi() {

cout << "Point says HI" << endl;

}

};

class Point3D : public Point {

public:

int z;

void sayHi() {

cout << "Point3D ... hello" << endl;

}

};

void makePointsTalk(Point\* p) {

p->sayHi();

}

void main() {

Point p;

Point3D q;

makePointsTalk(&p);

makePointsTalk(&q);

system("pause");

}

What does the “makePointesTalk” function print on the screen?

Remember the methods are not stored in the object’s footprint. The code has no way of knowing what kind of object the “p” really is in memory.

We can add our own type information to the class so we can know exactly what it is:

class Point {

public:

int type;

Point() {type=0;}

class Point3D : public Point {

public:

int z;

Point3D() {type=1;}

void makePointsTalk(Point\* p) {

if(p->type==0) {

p->sayHi();

} else {

((Point3D\*)p)->sayHi();

}

}

This is syntactically unpleasant, and this code has to know about every type of object in the universe. If you add another kind of Point you have to come here and add to the “if”.

Notice we really don’t care what kind of object it is here. All we care about is that it has a “sayHi” method. In fact, it is none of our business what kind of object it is! We want all Point3D points to behave as they should and all of Point objects to behave as they should. And new object will behave as they should.

All we need to know is that they all have a “sayHi” method because the base class guarantees it.

This object I have a pointer to has methods. I know what the method names are, but I don’t know exactly what it will do when I call it. That’s its business. My business is getting it to behave.

You can do this yourself. C programmers do it all the time with functors (function pointers).

**8100:**

**9000: Function Pointers for Point**

void Point::sayHi() {

cout << "Point says HI" << endl;

}

8300 (Point::getX)

8100 (Point::sayHi)

void Point3D::sayHi() {

cout << "Point3D ..." << endl;

}

**8200:**

**A000: Function Pointers for Point3D**

8300 (Point::getX)

8200 (Point3D::sayHi)

void Point::getX() {

return x;

}

**8300:**

2 (z)

78 (y)

24 (x)

A000 (vtab for Point3D)

**Point3D in memory at 7500**

24 (x)

9000 (vtab for Point)

78 (y)

**Point in memory at 7400**

500 (z)

A000 (vtab for Point3D)

**Point3D in memory at 7800**

101 (y)

99 (x)

Compiler inserts this code to call “sayHi”:

* Get the vtab pointer from the object’s memory (always first)
* Get the address of the function from entry[1] (sayHi)
* Call the function through its pointer and pass arguments

The C++ compiler will manage the tedious syntax for you if you tell it to. You simply identify all the methods that you want to be called through this indirect mechanism with “virtual”.

If have one or more virtual functions in your class then the compiler will generate the table and will add a pointer to it in every object you create.

class Point {

public:

int x;

int y;

virtual void sayHi() {

cout << "Point says HI" << endl;

}

};

In the derived class you should add “virtual” to the method, but it is optional. The compiler knows if it is virtual in the baseclass then it is virtual in ALL classes up the derived chain. No choice.

Now run the program. Points behave like Points and Point3Ds behave like Point3Ds and the code that uses the pointer never knows exactly what type it is using … doesn’t care. It just knows about the methods it can call. This is all about the methods … the interface.

A word of caution. Remember that the compiler generates code to have destructors call the baseclass. You want to make sure the derived class destructor is called first.

IF YOU HAVE ANY VIRTUAL METHODS IN YOUR CLASS, MAKE A VIRTUAL DESTRUCTOR.

virtual ~Point();

Then if you delete one with a base-class pointer it calls the full destructor chain.

virtual double getMagnitude() {

double a = x; // getX()

double b = y; // getY()

return sqrt(a\*a + b\*b);

}

class RandomPoint {

public:

virtual int getX() {

return rand();

}

virtual int getY() {

return rand();

}

};

Do you need this kind of indirection? It costs. Will you ever use it? Crystal ball on future or good massive design up front.

“Pure virtual” means there is no code. The value “0” gets placed in the table (actually the address of an error function). These all-pure-virtual classes are called “interfaces”. You can (and usually do) inherit from several. This is called “implementing” the interface.

class Speaker { // An "interface"

public:

virtual ~Speaker() {}

virtual void sayHi() = 0; // Must be overriden

virtual void sayBye() = 0;

};

class Feed {

public:

virtual ~Feed() {}

virtual void eat() = 0;

};

class Cat : public Speaker, public Feed {

public:

};