**Part C - Class Relationships**  
  
**Inheritance and Inclusion Polymorphism**  
  
Model generalization and specialization using inheritance hierarchies  
Model polymorphic behavior using interfaces and virtual functions

*"Abstract interfaces help you focus on getting an abstraction right without muddling it with implementation or state management details." Sutter, Alexandrescu (2005).*

[Inheritance Basics](https://ict.senecacollege.ca/~oop345/pages/content/inher.html#abs) | [Inclusion Polymorphism](https://ict.senecacollege.ca/~oop345/pages/content/inher.html#pol) | [Liskov Substitution Principle](https://ict.senecacollege.ca/~oop345/pages/content/inher.html" \l "lis) | [Exercises](https://ict.senecacollege.ca/~oop345/pages/content/inher.html#exe)

Relationships between classes can exhibit various degrees of coupling.  Inheritance is the tightest possible relationship between classes.  It is the most highly coupled and is defined in hierarchical terms and supports both abstraction and polymorphism.  A derived class in an inheritance hierarchy includes the entire structure of its base class and only defines those additional features that specialize its base class.  Over the life-cycle of a class hierarchy, programmers add features that specialize the hierarchy further.  This simplifies the development process throughout the life cycle of the hierarchy.  Such reuse of the base class improves substitutability incrementally and constructively.  An abstract base class exposes the structure common to all classes in the hierarchy.  The Liskov Substitution Principle ensures that the behaviors of the derived classes do not violate constraints on the base classes.

This chapter reviews inheritance of single lineage and the role of an abstract base class as the interface to a class hierarchy.  This chapter also reviews inclusion polymorphism, which distinguishes the functionality of different classes in a hierarchy through type-specific virtual functions.  This chapter concludes with a brief discussion of the substitution principle that ensures proper design of inheritance hierarchies.  Inheritance of multiple lineages is covered in a later chapter entitled [Multiple Inheritance](https://ict.senecacollege.ca/~oop345/pages/content/mult_.html).

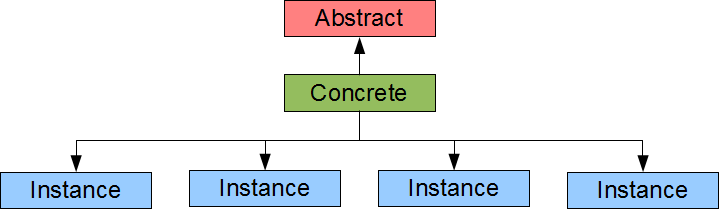
**INHERITANCE BASICS**

Inheritance is a relationship between user-defined types.  These types may be:

* concrete types - their representation is part of their definition and is known
* abstract types - their representation is not part of their definition and is unknown

**Abstract and Concrete Classes**

An abstract class defines an interface to an inheritance hierarchy.  It is an incomplete class in the hierarchy.  We complete its implementation by deriving a new class that adds the missing details.  We cannot create an instance of an abstract class.  We call the complete class a *concrete* class.



As an example of an abstract base class, consider the **Shape** class defined below.  A **Shape** has a volume.

|  |
| --- |
| **#ifndef SHAPE\_H**  **#define SHAPE\_H**  **// A Shape**  **// Shape.h**  **class Shape {**  **public:**  **virtual double volume() const = 0;**  **};**  **#endif** |

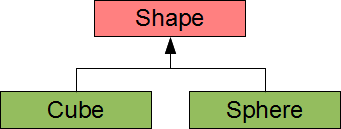
The pre-processor guard ensures that the **Shape** class is only defined once in a translation unit.  Once **SHAPE\_H** has been defined, any **include** directive that attempts to insert the class definition skips the definition.  This implements the one-definition rule of C++.

This abstract class identifies the member function that the hierarchy exposes to its clients and guarantees to deliver to its clients.  The signature of the pure virtual function identifies the exposed member function uniquely.  The assignment to **0** identifies the function as undefined or *pure*.

**Completing an Abstract Class**

A **Shape** object has volume.  To instantiate a **Shape** object, we need to identify this geometric property.  We do so by deriving a concrete class from the abstract **Shape** class.  The derived class contains the geometric properties needed to determine its volume.  Since the derived class includes the missing geometric properties and the definition of the object's volume, we can create instances of the derived class.

Consider an inheritance hierarchy that is built around the **Shape** class and consists of two derived classes named **Cube** and **Sphere**.  Each **Shape** has its own calculation of volume.



Each derived class - **Cube** and **Sphere** - defines its **volume()** with its own query:

|  |  |
| --- | --- |
| **// A Cube**  **// Cube.h**  **#include "Shape.h"**  **class Cube : public Shape {**  **double len;**  **public:**  **Cube(double);**  **double volume() const;**  **};** | **// A Sphere**  **// Sphere.h**  **#include "Shape.h"**  **class Sphere : public Shape {**  **double rad;**  **public:**  **Sphere(double);**  **double volume() const;**  **};** |

The implementation files - **Cube.cpp** and **Sphere.cpp** - define the volume calculation:

|  |  |
| --- | --- |
| **// A Cube**  **// Cube.cpp**  **#include "Cube.h"**  **Cube::Cube(double l) : len{l} {}**  **double Cube::volume() const {**  **return len \* len \* len;**  **}** | **// A Sphere**  **// Sphere.cpp**  **#include "Sphere.h"**  **Sphere::Sphere(double r) : rad{r} {}**  **double Sphere::volume() const {**  **return 4.18879 \* rad \* rad \* rad;**  **}** |

**Application**

The following example calculates the volume of any **Shape** in the hierarchy.  The user selects the **Shape** and enters the required dimension.  The results for two separate runs are shown on the right:

|  |  |
| --- | --- |
| **// Shape Hierarchy**  **// Shape.cpp**  **#include <iostream>**  **#include "Cube.h"**  **#include "Sphere.h"**  **void displayVolume(const Shape\* shape) {**  **if (shape)**  **std::cout << shape->volume() << std::endl;**  **else**  **std::cout << "error" << std::endl;**  **}**  **Shape\* select() {**  **Shape\* shape;**  **double x;**  **char c;**  **std::cout << "s (sphere), c (cube) : ";**  **std::cin >> c;**  **if (c == 's') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Sphere(x);**  **} else if (c == 'c') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Cube(x);**  **} else**  **shape = nullptr;**  **return shape;**  **}**  **int main() {**  **Shape\* shape = select();**  **displayVolume(shape);**  **delete shape;**  **}** | **First run :**  **-----------**  **s (sphere), c (cube) : s**  **dimension : 1**  **4.18879**  **Second run :**  **------------**  **s (sphere), c (cube) : c**  **dimension : 2**  **8**  **Third run :**  **-----------**  **s (sphere), c (cube) : d**  **error** |

Adding another derived class to the **Shape** hierarchy will only require an upgrade to the **select()** function.  **displayVolume()** and **main()** work with the interface directly and are sufficiently general not to require any upgrade.

**Good Design Practice**

Including the header files for the **Cube** class and the **Sphere** class in **Shape.cpp** would lead to a multiple definition of the **Shape** class if the pre-processor guard was omitted.

It is common design practice to wrap each header file in its own pre-processor guard directives.

**INCLUSION POLYMORPHISM**

**Polymorphic Objects**

A polymorphic object is an object that can have different types throughout its lifetime.  Consider the function named **displayVolume()** in the **Shape.cpp** example above.

|  |
| --- |
| **void displayVolume(const Shape\* shape) {**  **if (shape)**  **std::cout << shape->volume() << std::endl;**  **else**  **std::cout << "error" << std::endl;**  **}** |

The identifier **shape** receives the address of a polymorphic object.  In order to determine which function definition of the **Shape** hierarchy to call in calculating the volume of the object, the identifier needs to know the dynamic type of the object to which the identifier is attached.  (**Shape** is the static type.)

C++ supplies the dynamic type at run-time.

Problems that arise with dynamic typing include:

* determining the dynamic type in copying a polymorphic object to another polymorphic object
* specializing an operation for a dynamic type
* excluding a specific type from most derived selection

**Copying Operations**

Copying a polymorphic object at different stages of execution requires knowledge of its dynamic type.  In order to allocate memory for the copy, the run-time needs to know the dynamic type at the moment of copying.  For example, the following **copy** function needs to know which constructor to call - **Cube** or **Sphere**:

|  |
| --- |
| **// Copy a Shape Object**  **// copy\_shape.cpp**  **#include "Shape.h"**  **Shape\* copy(Shape\* original) {**  **Shape\* copy = new ???? (original);**  **return copy;**  **}** |

To determine the dynamic type at run-time we can define a cloning member function for each concrete class in the hierarchy.

Let us upgrade the **Shape** interface to expose a **clone()** member function from the hierarchy.  This member function returns the address of a copy of the object created using its current dynamic type. :

|  |
| --- |
| **#ifndef SHAPE\_H**  **#define SHAPE\_H**  **// Polymorphic Objects - Cloning**  **// Shape.h**  **class Shape {**  **public:**  **virtual double volume() const = 0;**  **virtual Shape\* clone() const = 0;**  **};**  **#endif** |

The definition of each concrete class declares the **clone()** member function:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Cloning**  **// Cube.h**  **#include "Shape.h"**  **class Cube : public Shape {**  **double len;**  **public:**  **Cube(double);**  **double volume() const;**  **Shape\* clone() const;**  **};** | **// Polymorphic Objects - Cloning**  **// Sphere.h**  **#include "Shape.h"**  **class Sphere : public Shape {**  **double rad;**  **public:**  **Sphere(double);**  **double volume() const;**  **Shape\* clone() const;**  **};** |

The **Cube** and **Sphere** implementation files define the different versions of the **clone()** member function:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Cloning**  **// Cube.cpp**  **#include "Cube.h"**  **Cube::Cube(double l) : len(l) {}**  **Shape\* Cube::clone() const {**  **return new Cube(\*this);**  **}**  **double Cube::volume() const {**  **return len \* len \* len;**  **}** | **// Polymorphic Objects - Cloning**  **// Sphere.cpp**  **#include "Sphere.h"**  **Sphere::Sphere(double r) : rad(r) {}**  **Shape\* Sphere::clone() const {**  **return new Sphere(\*this);**  **}**  **double Sphere::volume() const {**  **return 4.18879 \* rad \* rad \* rad;**  **}** |

Each member function creates a copy of the object by calling the copy constructor for the relevant dynamic type.

The following program uses the **clone()** member function to copy the current dynamic type and return the address of the copy.  The results of two runs are shown on the right:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Cloning**  **// cloning.cpp**  **#include <iostream>**  **#include "Cube.h"**  **#include "Sphere.h"**  **void displayVolume(const Shape\* shape) {**  **if (shape)**  **std::cout << shape->volume() << std::endl;**  **else**  **std::cout << "error" << std::endl;**  **}**  **Shape\* select() {**  **Shape\* shape;**  **double x;**  **char c;**  **std::cout << "s (sphere), c (cube) : ";**  **std::cin >> c;**  **if (c == 's') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Sphere(x);**  **} else if (c == 'c') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Cube(x);**  **} else**  **shape = nullptr;**  **return shape;**  **}**  **int main() {**  **Shape\* shape = select();**  **Shape\* clone = shape->clone();**  **displayVolume(shape);**  **displayVolume(clone);**  **delete clone;**  **delete shape;**  **}** | **First run :**  **-----------**  **s (sphere), c (cube) : s**  **dimension : 1**  **4.18879**  **Second run :**  **------------**  **s (sphere), c (cube) : c**  **dimension : 2**  **8** |

**Specializing an Operation for a Dynamic Type (Optional)**

An operation with a polymorphic operand requires knowledge of its dynamic type.  In order to determine which concrete type to invoke the run-time needs to know the object's dynamic type.  A *dynamic cast* returns this type.

Let us upgrade the **Shape** interface to expose a member function for comparing objects.  This function receives the address of an object and return true if the object has the same properties as the current object:

|  |
| --- |
| **#ifndef SHAPE\_H**  **#define SHAPE\_H**  **// Polymorphic Objects - Dynamic Cast**  **// Shape.h**  **class Shape {**  **public:**  **virtual double volume() const = 0;**  **virtual void display() const = 0;**  **virtual bool operator==(const Shape&) const = 0;**  **};**  **#endif** |

The member function will have different definitions in different derived classes and will require access to the instance variables.

We declare the member function in the definitions of the **Cube** and **Sphere** classes:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Dynamic Cast**  **// Cube.h**  **#include "Shape.h"**  **class Cube : public Shape {**  **double len;**  **public:**  **Cube(double len);**  **void display() const;**  **double volume() const;**  **bool operator==(const Shape&)**  **const;**  **};** | **// Polymorphic Objects - Dynamic Cast**  **// Sphere.h**  **#include "Shape.h"**  **class Sphere : public Shape {**  **double rad;**  **public:**  **Sphere(double);**  **void display() const;**  **double volume() const;**  **bool operator==(const Shape&)**  **const;**  **};** |

We define the member function in the **Cube** and **Sphere** implementation and use the **dynamic\_cast** template to cast the address received from a **Shape\*** type to the object's dynamic type:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Dynamic Cast**  **// Cube.cpp**  **#include <iostream>**  **#include "Cube.h"**  **Cube::Cube(double l) : len(l) { }**  **void Cube::display() const {**  **std::cout << "length = " <<**  **len << std::endl;**  **}**  **double Cube::volume() const {**  **return len \* len \* len;**  **}**  **bool Cube::operator==(const Shape& s)**  **const {**  **const Cube\* c =**  **dynamic\_cast<const Cube\*>(&s);**  **return c ? len == c->len : false;**  **}** | **// Polymorphic Objects - Dynamic Cast**  **// Sphere.cpp**  **#include <iostream>**  **#include "Sphere.h"**  **Sphere::Sphere(double r) : rad(r) { }**  **void Sphere::display() const {**  **std::cout << "radius = " <<**  **rad << std::endl;**  **}**  **double Sphere::volume() const {**  **return 4.18879 \* rad \* rad \* rad;**  **}**  **bool Sphere::operator==(const Shape& s)**  **const {**  **const Sphere\* c =**  **dynamic\_cast<const Sphere\*>(&s);**  **return c ? rad == c->rad : false;**  **}** |

Casting from the abstract **Shape\*** type to a concrete **Cube&** or **Sphere&** type enables access to a complete object.  Without the dynamic cast, the compiler reports an error that the instance variable is not a member of the **Shape** class:

|  |
| --- |
| **bool Cube::operator==(const Shape& s) const {**  **return len == s.len; // ERROR because len is not a member of Shape**  **}** |

Receiving the address of a **Cube** or **Sphere** object directly admit the definition but generates an error that the object is incomplete since **bool operator==(const Shape&) const** has yet to be defined:

|  |
| --- |
| **bool Cube::operator==(const Cube& s) const {**  **return len == s.len; // ERROR because Cube is not concrete**  **}** |

In the following program the user can compare two **Shape** objects.  The results for three runs are shown on the right:

|  |  |
| --- | --- |
| **// Polymorphic Objects - Dynamic Cast**  **// dynamic\_casting.cpp**  **#include <iostream>**  **#include "Cube.h"**  **#include "Sphere.h"**  **Shape\* select() {**  **Shape\* shape;**  **double x;**  **char c;**  **std::cout << "s (sphere), c (cube) : ";**  **std::cin >> c;**  **if (c == 's') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Sphere(x);**  **} else if (c == 'c') {**  **std::cout << "dimension : ";**  **std::cin >> x;**  **shape = new Cube(x);**  **} else**  **shape = nullptr;**  **return shape;**  **}**  **int main() {**  **Shape\* s1 = select();**  **Shape\* s2 = select();**  **s1->display();**  **s2->display();**  **if (\*s1==\*s2)**  **std::cout << "Same" << std::endl;**  **else**  **std::cout << "Different" << std::endl;**  **delete s1;**  **delete s2;**  **}** | **First Run**  **---------**  **s (sphere), c (cube) : c**  **dimension : 21**  **s (sphere), c (cube) : c**  **dimension : 21**  **length = 21**  **length = 21**  **Same**  **Second Run**  **----------**  **s (sphere), c (cube) : s**  **dimension : 20**  **s (sphere), c (cube) : s**  **dimension : 21**  **radius = 20**  **radius = 21**  **Different**  **Third Run**  **---------**  **s (sphere), c (cube) : s**  **dimension : 20**  **s (sphere), c (cube) : c**  **dimension : 20**  **radius = 20**  **length = 20**  **Different** |

**Dynamic Type Identification (Optional)**

Polymorphic objects can be interrogated for thier type, where type specific coding is necessary beyond the virtual mechanism provided by the language.  C++ supports run-time type identification (RTTI) for identifying a polymorphic object's dynamic type.

In the following example, the **show()** function calls the **display()** function on all concrete classes except for objects of type **C**.  The **typeid()** operator receives either an object, a pointer to an object, or an expression and returns a reference to a **type\_info** object that holds type information.  **type\_info** is defined in the header file **<typeinfo>**:

|  |  |
| --- | --- |
| **// Polymorphic Objects - RTTI**  **// rtti.cpp**  **#include <typeinfo> // for typeid**  **#include <iostream>**  **class A {**  **int x;**  **public:**  **A(int a) : x(a) {}**  **virtual void display() const {**  **std::cout << x << std::endl;**  **}**  **};**  **class B : public A {**  **int y;**  **public:**  **B(int a, int b) : A(a), y(b) {}**  **void display() const {**  **A::display();**  **std::cout << y << std::endl; }**  **};**  **class C : public B {**  **int z;**  **public:**  **C(int a = 4, int b = 6, int c = 7) :**  **B(a, b), z(c) {}**  **void display() const {**  **B::display();**  **std::cout << z << std::endl; }**  **};**  **// show calls display() on all types except C**  **//**  **void show(const A\* a) {**  **C cref;**  **if (typeid(\*a) != typeid(cref)) {**  **a->display();**  **} else**  **std::cout << typeid(cref).name() <<**  **" objects are private" << std::endl;**  **}**  **int main() {**  **A\* a[3];**  **a[0] = new A(3);**  **a[1] = new B(2, 5);**  **a[2] = new C(4, 6, 7);**  **for(int i = 0; i < 3; i++)**  **show(a[i]);**  **for(int i = 0; i < 3; i++)**  **delete a[i];**  **}** | **3**  **2**  **5**  **class C objects are private** |

Note how we determine the type by comparing **typeid(\*a)** with **typeid(cref)**.  The **name()** query on **typeid()** returns the address of a C-style null-terminated string that holds some description of the type name.  The description itself is implementation dependent.

**LISKOV SUBSTITUTION PRINCIPLE**

The Liskov Substitution Principle states that "a function that uses pointers or references to base classes must be able to use objects of derived classes without knowing it".  In other words, we should model our classes on their behaviors not their properties and model our data based on properties and not on behaviors.

**The Correct Design**

The classical example involves a square and a rectangle.  Each is a kind of the other, but only one design ensures substitutability.  That is, a rectangle is less abstract than a square: a rectangle should be derived from a square, since the rectangle adds a new measurement.  This design is illustrated in the following 2 programs.

**The Initial Design** version is:

|  |  |
| --- | --- |
| **// Liskov Substitution Principle**  **// liskov\_square.cpp**  **#include <iostream>**  **class Square {**  **double width;**  **public:**  **void setWidth(double w) { width = w; }**  **double getWidth() const { return width; }**  **};**  **void set(Square& s, double d) {**  **s.setWidth(d);**  **}**  **int main() {**  **Square s;**  **s.setWidth(20.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **set(s, 15.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **}** | **Square : 20**  **Square : 15** |

**The Upgraded Design**

|  |  |
| --- | --- |
| **// Liskov Substitution Principle - Rectangle**  **// liskov\_rectangle.cpp**  **#include <iostream>**  **class Square {**  **double width;**  **public:**  **void setWidth(double w) { width = w; }**  **double getWidth() const { return width; }**  **};**  **class Rectangle : public Square {**  **double height;**  **public:**  **void setHeight(double h) { height = h; }**  **double getHeight() const { return height; }**  **};**  **void set(Square& s, double d) {**  **s.setWidth(d);**  **}**  **int main() {**  **Square s;**  **s.setWidth(20.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **Rectangle r;**  **r.setWidth(10.0);**  **r.setHeight(30.0);**  **std::cout << "Rectangle : " << r.getWidth() << ", "**  **<< r.getHeight() << std::endl;**  **set(s, 15.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **set(r, 25.0);**  **std::cout << "Rectangle : " << r.getWidth() << ", "**  **<< r.getHeight() << std::endl;**  **}** | **Square : 20**  **Rectangle : 10, 30**  **Square : 15**  **Rectangle : 25, 30** |

The upgraded version adds a new dimension for the rectangle without changing any of the original code:

**An Incorrect Design**

Consider the alternative design, based on the fact that a square is a rectangle with certain properties: equal sides.

**The Initial Design**

Consider the initial program as one for a rectangle.

|  |  |
| --- | --- |
| **// Liskov Substitution Principle - Rectangle**  **// liskov\_rectangle\_.cpp**  **#include <iostream>**  **class Rectangle {**  **double width;**  **double height;**  **public:**  **void setWidth(double w) { width = w; }**  **void setHeight(double h) { height = h; }**  **double getWidth() const { return width; }**  **double getHeight() const { return height; }**  **};**  **void set(Rectangle& r, double d) {**  **r.setWidth(d);**  **}**  **int main() {**  **Rectangle r;**  **r.setWidth(10.0);**  **r.setHeight(30.0);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **set(r, 25);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **}** | **Rectangle : 10, 30**  **Rectangle : 25, 30** |

**The Upgraded Design**

If we add a square, derive it from a rectangle and ensure that the invariant of the square is maintained (equal sides), we need to add **virtual** to the original design (that is, change the original design):

|  |  |
| --- | --- |
| **// Liskov Substitution Principle - Square from Rectangle**  **// liskov\_square\_.cpp**  **#include <iostream>**  **class Rectangle {**  **double width;**  **double height;**  **public:**  **virtual void setWidth(double w) { width = w; }**  **virtual void setHeight(double h) { height = h; }**  **double getWidth() const { return width; }**  **double getHeight() const { return height; }**  **};**  **class Square : public Rectangle {**  **public:**  **void setWidth(double s) {**  **Rectangle::setWidth(s);**  **Rectangle::setHeight(s);**  **}**  **void setHeight(double s) {**  **Rectangle::setWidth(s);**  **Rectangle::setHeight(s);**  **}**  **};**  **void set(Rectangle& r, double d) {**  **r.setWidth(d);**  **}**  **int main() {**  **Square s;**  **s.set(20.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **Rectangle r;**  **r.setWidth(10.0);**  **r.setHeight(30.0);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **set(s, 15.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **set(r, 25.0);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **}** | **Square : 20**  **Rectangle : 10, 30**  **Square : 15**  **Rectangle : 25, 30** |

**The Design is Still Flawed**

Consider the following initial program for a rectangle.

|  |  |
| --- | --- |
| **// Liskov Substitution Principle - Rectangle**  **// liskov\_rectangle\_\_.cpp**  **#include <iostream>**  **class Rectangle {**  **double width;**  **double height;**  **public:**  **virtual void setWidth(double w) { width = w; }**  **virtual void setHeight(double h) { height = h; }**  **double getWidth() const { return width; }**  **double getHeight() const { return height; }**  **};**  **void set(Rectangle& r) {**  **r.setWidth(3);**  **r.setHeight(4);**  **if(r.getWidth() \* r.getHeight() != 12)**  **std::cerr << "\* volume isn\'t 12 \*\n";**  **}**  **int main() {**  **Rectangle r;**  **r.setWidth(10.0);**  **r.setHeight(30.0);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **set(r);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **}** | **Rectangle : 10, 30**  **Rectangle : 3, 4** |

Deriving the square from the rectangle makes this obvious.  Note the error message from **set**:

|  |  |
| --- | --- |
| **// Liskov Substitution Principle - Square from Rectangle**  **// liskov\_square\_\_.cpp**  **#include <iostream>**  **class Rectangle {**  **double width;**  **double height;**  **public:**  **virtual void setWidth(double w) { width = w; }**  **virtual void setHeight(double h) { height = h; }**  **double getWidth() const { return width; }**  **double getHeight() const { return height; }**  **};**  **class Square : public Rectangle {**  **public:**  **void setWidth(double s) {**  **Rectangle::setWidth(s);**  **Rectangle::setHeight(s);**  **}**  **void setHeight(double s) {**  **Rectangle::setWidth(s);**  **Rectangle::setHeight(s);**  **}**  **};**  **void set(Rectangle& r) {**  **r.setWidth(3);**  **r.setHeight(4);**  **if(r.getWidth() \* r.getHeight() != 12)**  **std::cerr << "\* volume isn\'t 12 \*\n";**  **}**  **int main() {**  **Square s;**  **s.setWidth(20.0);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **Rectangle r;**  **r.setWidth(10.0);**  **r.setHeight(30.0);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **set(s);**  **std::cout << "Square : " << s.getWidth() << std::endl;**  **set(r);**  **std::cout << "Rectangle : " << r.getWidth()**  **<< ", " << r.getHeight() << std::endl;**  **}** | **Square : 20**  **Rectangle : 10, 30**  **volume is not 12**  **Square : 4**  **Rectangle : 3, 4** |

When creating a derived class causes a change to the base class, this is a sign that the design is probably faulty.

**EXERCISES**

* Complete the practice problem in the Handout on [Abstract Base Classes](https://ict.senecacollege.ca/~oop345/pages/handouts/h20.html).
* Read Robert Martin on [The Liskov Substitution Principle](http://web.archive.org/web/20151128004108/http:/www.objectmentor.com/resources/articles/lsp.pdf)