**Inheritance and Inclusion Polymorphism**

The classes in an object-oriented program stand in a variety of relationships to one another.  These relationships exhibit various degrees of coupling.  Inheritance is the tightest possible relationship.  The classes are highly coupled and part of a hierarchy that supports both abstraction and polymorphism.  A derived class in an inheritance hierarchy includes the entire structure of its parent base class and defines only those additional features that specialize the base class.  Over the life-cycle of a class hierarchy, programmers add more features that specialize the hierarchy further.  This reuse of existing code simplifies the development process over the life cycle of the hierarchy.  Such reuse improves substitutability incrementally and constructively.  An abstract base class exposes to the program the structure common to all the 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 single inheritance and the role of an abstract base class as the interface to an inheritance hierarchy.  It 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, which advocates proper design of inheritance hierarchies.  Inheritance from several bases classes is covered in a later chapter entitled [Multiple Inheritance](https://mwatler.github.io/sep200/Week12/MultipleInheritance.docx).

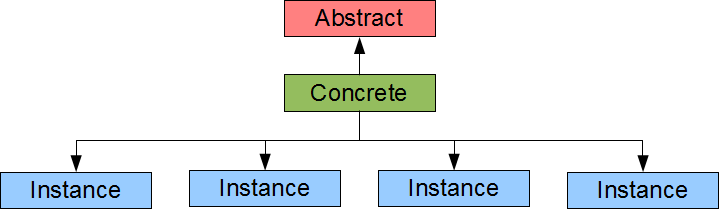
Inheritance Basics

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

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

Abstract and Concrete Classes

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



As an example of an abstract base class, consider the Shape class defined below.  All that we know of a Shape is that it has a volume.

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

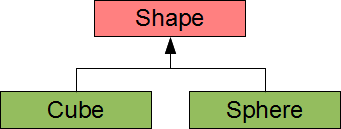
The pre-processor guard (#ifndef ... #endif) ensures that the definition of the Shape class occurs only once in a translation unit.  Once SHAPE\_H has been defined, any include directive that specifies insertion of the class definition skips the definition.  This implements the one-definition rule of C++.

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

Completing an Abstract Class

To instantiate a Shape object, we need to define a function that returns its volume.  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.  If the derived class includes the missing geometric properties needed to define the object's volume, we can create instances of that class.

Consider an inheritance hierarchy built around the Shape class that consists of two derived classes named Cube and Sphere.  Each Shape has its own specific calculation of volume.  Each classes constructor takes in the required geometric property.



Each derived class - Cube and Sphere - defines its own volume() 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 type of Shape and enters its required dimension.  The results for three 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.  The displayVolume() and main() functions 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.

Inclusion Polymorphism

Polymorphic Objects

A polymorphic object is an object that can have different dynamic 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 parameter shape receives the address of a polymorphic object.  In order to determine which of the Shape hierarchy's function definition to call in calculating the volume of the object, the function needs to know the dynamic type of the object to which the parameter points.  (Shape is the static type.)

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

Questions that arise with dynamic typing include:

* how to determine the dynamic type in copying a polymorphic object to another polymorphic object
* how to specialize an operation for a dynamic type
* how to exclude a specific type from the default (most derived) selection of a function

Copying Operations

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

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

To determine the dynamic type at run-time we introduce a cloning member function to the hierarchy.

Let us upgrade the Shape interface to expose a clone() query.  This member function returns the address of a copy of the object created using its 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() query:

|  |  |
| --- | --- |
| // 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 class specific 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 query creates a copy of the object by calling the copy constructor for its own class' dynamic type.

The following program uses the clone() member function to copy the 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

In expressions that involve operands of polymorphic type, the operator functions require knowledge of the received object's dynamic type.  A *dynamic cast* returns this type.

Let us upgrade the Shape interface to expose a query 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 comparison will have a different definition in each concrete class and will require access to the instance variables of that class.

We declare the query in our 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 each query in the Cube and Sphere implementations and use the dynamic\_cast template to cast the type of the address received in the parameter to an address of 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 respective complete object.  Without the dynamic cast, the compiler would report 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  } |

Moreover, receiving the address of a Cube or Sphere object directly would theoretically enable access to the object's instance variable but would generate an error that the object is incomplete because the bool operator==(const Shape&) const is undefined:

|  |
| --- |
| 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

The C++ language includes two elements that hold run-time information in addition to dynamic\_cast for identifying a polymorphic object's dynamic type.  These elements can be used where type specific coding beyond the virtual mechanism is required.

In the following example, the show() function calls the display() function on all concrete classes except for objects of type C, which are considered private.  The typeid() operator allows the type of an object to be determined at run-time.  It receives a type and returns a std::type\_info object that represents the type received and 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 |

We determine the type by comparing typeid(\*a) to typeid(cref) where cref is the reference type.  The name() query on the type\_info object referred to by 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

Some confusion can arise in distinguishing type from subtypes in inheritance hierarchies.  The Liskov Substitution Principle provides guidance that avoids design ambiguity.  The 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

The initial version focuses on the square alone:

|  |  |
| --- | --- |
| // 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

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

|  |  |
| --- | --- |
| // 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 |

An Incorrect Design

In mathematics, a square is a special case of a rectangle: it has equal sides.  This might lead us to propose a design in which the square is derived from a rectangle (a special case of it).  Consider this alternative design:

The Initial Design

The initial design focuses on a rectangle alone:

|  |  |
| --- | --- |
| // 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 now 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.