Inheritance

Encapsulation    Inheritance    Polymorphism

Object Oriented Programming Pillars
Here the child class adds an additional ivar, var3 and overrides the parent class's method1().
A subclass may redefine its behavior

A subclass may opt to redefine (override) the superclass's behavior.

This is needed when super's behavior is too general and subclass needs more specificity.

Here the child class adds an additional ivar, `ivar3` and overrides the parent class's `method1()`.
A Superclass should offer only common features

Design Blindside

• If a superclass includes behavior that is not truly common to all subclasses, then some of those subclasses will have to override.

• Extensive overriding leads to code duplication efforts and more maintenance headaches.

Here the child class adds an additional ivar, var3 and overrides the parent class's method1().
Sphere - Ball Demo

Sphere

- r:double

  + diameter():double {query}
  + circumference():double {query}
  + area():double {query}
  + volume():double {query}
  + toString():string {query}

Ball

- name:string

  + toString():string {query}
class Sphere {
private:
    double r;

public:
    Sphere() : r(1.0) { }
    Sphere(double r) : r(r) { }
    void setRadius(double r) { this->r = r; }
    double getRadius() const { return r; }
    double diameter() const { return r * 2.0; }
    double circumference() const { return 2.0 * PI * r; }
    double area() const { return 4.0 * PI * r * r; }
    double volume() const { return 4.0 / 3.0 * PI * r * r * r; }
    string toString() const { return "class: Sphere"; }
};

Inherited behavior
class Ball : public Sphere {
private:
    string name;

public:
    Ball() : Sphere(1.0), name("na") { }
    Ball(string name, double r): Sphere(r), name(name) { }

    void setName(string name) { this->name = name; }
    double getName() const { return name; }
    string toString() const { return "class: Ball" + ", parent " + Sphere::toString(); }
};
The process of creating a subclass object (Ball), involves the creation of the superclass object (Sphere) first. The subclass's memory footprint shows that the new object is made up of all ivars in the super, plus all ivars in the sub.

Similarly, all super's public methods are accessible to the sub, its descendants and of course clients. A client of the sub will have access to all methods, from sub to sup to its sup and up the line of inheritance.
Calling a method

Sphere sphere(10.0);
Ball bball("Basket Ball", 5.0);

cout << sphere.toString();
  // class: Sphere

cout << bball.toString();
  // class: Ball, parent class: Sphere

In order for the compiler to decide if a method call is even valid, it must check with the declared type's class.

If the declared type offers the method being called, then no error will ensue.

If the declared class does not offer the method being called, then an error will be given.

In object creation: The super's constructor must be called first, either explicitly or implicitly. This is required in order for the super class object to be initialized properly. This call is made from the sub's constructor(s). We will use the idea of a designated constructor to carry out this task.

In object termination: When a sub object goes out of scope, its destructor is called first to release all the memory allocated for its needs, and then the super's destructor is called to release its resources. The call is made by the run-time, so please do not call the destructore explicitly.
Three access modes are available

- **Private**
  - Not accessible by clients
  - Accessible by class and friends

- **Protected**
  - Not accessible by clients
  - Accessible by class and friends
  - Accessible by derived classes and friends

- **Public**
  - Accessible by all
There are three types of Class relationships

- **Is-a**: Implemented using *public* inheritance
  - may provide a child object whenever a parent or ancestor object is needed

- **As-a**: Implemented using *private* inheritance
  - one class makes exclusive use of another

- **Has-a**: Implemented using *composition*
  - class contains objects of another class as member
There are three types of Inheritance

Think of the type of inheritance as a privacy filter that controls how methods in the super are viewed in the sub. This view of course relates only to clients of the sub class, not the sub class itself. The sub’s access to methods in the super is only affected by the accessibility keyword used to define the methods in the super, not the filter.

**public inheritance (open filter)**

Public inheritance does not alter the original access in any way. So, all inherited members that are public in the super class remain public in the sub. Effectively this has not noticeable changes or effects on clients of the sub class.
Public Inheritance

The object can only access + members of its class.
The public inheritance does not alter the accessibility of A's members for B's descendants.

Class B has access to +,# members of A.
Class C has access to +,# members of A.

The public inheritance does not alter the accessibility of A's members for B's descendants.
There are three types of Inheritance

**protected inheritance (translucent filter)**

protected inheritance effectively removes public access to super methods by sub clients. It is a restrictive filter that allows sub methods access to those methods while at the same time preventing clients of sub from accessing those same methods.

It is used primarily to give inherited classes access without exposing that same functionality to their clients.
Protected Inheritance

Class B has access to + and members of A.

Class C has only # access to members of A.

The object can only access + members of its class.

The protected inheritance alters the accessibility of A's + members for B's descendants.
There are three types of Inheritance

- **Public**
- **Protected**
- **Private**

**Inheritance Filter**

- Method accessibility in superclass:
  - Public
  - Protected
  - Private

- Method accessibility in subclass:
  - Private
  - Private
  - Private

Direct subclass not affected by filter

**Private inheritance (closed filter)**

Private inheritance is the most restrictive and basically gives no access to super methods to any of sub's clients, not any of sub's children.

Effectively this is a privileged filter for the exclusive use of the direct sub class. Used in as-a relationships, where you might want to implement a class (Stack let's say) as another class (List let's say), without advertising that fact to anyone else. As far as clients and children, they only have access to Stack methods.
Private Inheritance

The object can only access + members of its class.

Class B has access to +, # members of A.

Class C has no access to members of A.

The private inheritance removes the accessibility of A's +, # members for B's descendants.
#ifndef __cppProject__Color_
#define __cppProject__Color__

#include <string>
using namespace std;

class Color {
private:
    int r;
    int g;
    int b;

public:
    Color(const int r = 0, const int g = 0, const int b = 0);
    int getR() const;
    int getG() const;
    int getB() const;
    void setR(const int r);
    void setG(const int g);
    void setB(const int b);
/* We provide a setColor to make it more convenient for the user. Now they can call this method to set all three values. The getColor() is not needed since we have the = operator for that. */

void setColor(const int r, const int g, const int b);

    string toString() const;
};

#ifdef __cppProject__Color__
#endif /* defined(__cppProject__Color__) */
#include "Color.h"
#include <sstream>

using namespace std;

Color::Color(const int r, const int g, const int b) : r(r), g(g), b(b)
{} // Color()

int Color::getR() const { return r; } // end getR()
int Color::getG() const { return g; } // end getG()
int Color::getB() const { return b; } // end getB()

void Color::setR(const int r) { this->r = (r < 0 || r > 255)? 0 : r; }
void Color::setG(const int g) { this->g = (g < 0 || g > 255)? 0 : g; }
void Color::setB(const int b) { this->b = (b < 0 || b > 255)? 0 : b; }

void Color::setColor(const int r, const int g, const int b) {
    setR(r);
    setG(g);
    setB(b);
} // end setColor()
```cpp
string Color::toString() const {
    stringstream oss;
    oss << "rgb: (" << getR() << ", " << getG() << ", " << getB() << ")";

    return oss.str();
} // end toString()
```
#ifndef __cppProject__Rectangle__
#define __cppProject__Rectangle__

#include <string>
#include "Color.h"

using namespace std;

class Rectangle {
private:
  int width;
  int length;
  Color fillColor;

public:
  /* Notice the use of Color() - the call to the default Color ctor - as 
  * the default initial value for the fillColor. This creates a default 
  * Color object that is then assigned to the fillColor field. 
  */
  Rectangle(const int width = 0, const int length = 0, 
            const Color& fillColor = Color());

  int getWidth() const;
  int getLength() const;
  Color getColor() const;
```cpp
void setWidth(const int width);
void setLength(const int length);
void setDimensions(const int width, const int length);
void setColor(const Color& fillColor);

int area() const;
int perimeter() const;

string toString() const;

};

#ifndef __cppProject__Rectangle__
#endif /* defined(__cppProject__Rectangle__) */
```
#include <sstream>
#include "Rectangle.h"

using namespace std;

/* Use the member initializer (:) to initialize the three members of the class. */
/* Use the name of the field, especially when using composition, such as with the * fillColor. */
Rectangle::Rectangle(const int width, const int length, const Color& fillColor)
    : width(width), length(length), fillColor(fillColor) {} // end Rectangle()

int Rectangle::getWidth() const { return width; } // end getWidth()
int Rectangle::getLength() const { return length; } // end getLength()
Color Rectangle::getColor() const { return fillColor; }

void Rectangle::setWidth(const int width) { this->width = width; }  
void Rectangle::setLength(const int length) { this->length = length; }

void Rectangle::setDimensions(const int width, const int length) {  
    setWidth(width);  
    setLength(length);  
} // end setDimensions()
```cpp
void Rectangle::setColor(const Color &fillColor) { this->fillColor = fillColor; }

int Rectangle::area() const { return getWidth() * getLength(); }
int Rectangle::perimeter() const { return 2 * getWidth() + 2 * getLength(); }

string Rectangle::toString() const {
    stringstream oss;
    oss << "w=" << getWidth() << ",l=" << getLength() << "," << fillColor.toString();
    return oss.str();
} // end toString()
```
#ifndef __cppProject__Box__
#define __cppProject__Box__

#include <string>
#include "Rectangle.h"

using namespace std;

class Box : public Rectangle {
private:
    int height;

public:
    Box(const int height = 0, const int width = 0, const int length = 0);
    int getHeight() const;
    void setHeight(const int height);
    void setDimensions(const int height, const int width, const int length);
    int area() const;
    int volume() const;
    string toString() const;
};
#endif /* defined(__cppProject__Box__) */
#include "Box.h"
#include <sstream>

using namespace std;

Box::Box(const int height, const int width, const int length)
    : Rectangle(width, length), height(height)
{} // end Box()

int Box::getHeight() const { return height; }
void Box::setHeight(const int height) { this->height = height; }

void Box::setDimensions(const int height, const int width, const int length) {
    setHeight(height);
    /* Here we use the scope resolution operator (::) to make it clear that the
     * method belongs to the parent class, since we also have a setDimensions()
     * method in the child class. It makes it clearer, but is not required, since
     * we are overloading the method, not overriding it.
     */
    Rectangle::setDimensions(width, length);
} // end setDimensions()
/* This method is overridden to offer specific functionality that applies to * Box objects. */

int Box::area() const {
    return 2 * getWidth() * getLength() +
            2 * getWidth() * getHeight() +
            2 * getLength() * getHeight();
} // end area()

int Box::volume() const {
    return getWidth() * getLength() * getHeight();
} // end volume()

string Box::toString() const {
    ostringstream oss;
    oss << "h=" << getHeight() << "," << "r=" << Rectangle::toString();
    return oss.str();
} // end toString()
```cpp
#include <iostream>
#include "Box.h"

using namespace std;

void displayBox(const Box & box);

int main() {
    Box b1;
    Box b2(10, 20, 30);
    cout << "b1: " << b1.toString() << endl
         << "b2: " << b2.toString() << endl
         << "b1: ";
    displayBox(b1);
    cout << endl;
    /* Notice how we are creating a nameless object to use as the
       * parameter to the displayBox() function. Nice technique, that
       * is frequently used.
       */
    cout << "anonymous: ";
    displayBox( Box(20, 20, 20) );
    cout << endl;
```
main.cpp

```cpp
b1 = Box(40, 40, 40);

/* Since the Color property is associated with the Rectangle class, and */
/* since we did not update the Box class to take advantage of the Color */
/* we must then use the setColor() to add color to boxes. */
b1.setColor( Color(30, 100, 150) );
cout << "b1: ";
displayBox(b1);
cout << endl;
return 0;
} // end main()

void displayBox(const Box& box) {
    cout << box.toString();
} // end displayBox()
```
Output

b1: h=0, w=0, l=0, rgb:(0,0,0)
b2: h=10, w=20, l=30, rgb:(0,0,0)
b1: h=0, w=0, l=0, rgb:(0,0,0)
anonymous: h=20, w=20, l=20, rgb:(0,0,0)
b1: h=40, w=40, l=40, rgb:(30,100,150)