Object Oriented Programming Pillars

- Encapsulation
- Inheritance
- Polymorphism
A Subclass redefines its behavior

Polymorphism is a two-sided contract.
Superclass identifies behavior that should be redefined. These are the virtual methods.

Subclass chooses to override virtual methods in order to offer specific behavior.

<table>
<thead>
<tr>
<th>Superclass</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ivar1</td>
<td></td>
</tr>
<tr>
<td>ivar2</td>
<td></td>
</tr>
<tr>
<td>method1()</td>
<td></td>
</tr>
<tr>
<td>virtual method2()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subclass</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ivar3</td>
<td></td>
</tr>
<tr>
<td>virtual method2()</td>
<td></td>
</tr>
</tbody>
</table>

Here the child class adds an additional ivar, var3 and overrides the parent class's method1().
Calling a method (late)

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

cout << sphere->toString();
// class: Sphere

method is selected at compile time
This is early binding

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

Declared = actual: Sphere 
Actual = actual: Ball

sphere = bball;

method selected at run time
This is late binding

cout << sphere->toString();
// class: Ball, parent class: Sphere

If Ball does not override toString(), then polymorphism cannot occur.

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 destructor explicitly.
Here we have a pointer to a super class (Sphere), which we assign an object of type Sphere (sphere). Thus the declared type and the actual type are of the same class.

A call to a method through the pointer will resolve to the method in the actual type, in this case the same declared type (Sphere).

Then we assign to the pointer a sub instance (ball). Now the declared type is Sphere and the actual type is Ball (ball). However, when we make the call, the compiler still uses the super's method and not the sub's like expected. This is due to the non-transparency of polymorphism in C++.

Polymorphic behavior in C++ must be enforced by the programmer, unlike Java let's say where it is enforced by the language itself. So, we as C++ programmer's must take some extra steps in order for our methods to become polymorphic. See the next slide.
Use these virtual guidelines

- In Superclass
  - Ctors, accessors should not be virtual
  - All else should be virtual

- In Subclass
  - Use virtual when overriding virtual methods
  - Overriding a virtual method enables polymorphism
  - Overriding a non-virtual method does not enable polymorphism
Preventing instantiation with Abstract Classes

- A class is abstract if it includes pure virtual methods (at least one)

- A pure virtual method does not include an implementation, only a prototype.

- Abstract methods must be implemented by a subclass
This class included two pure virtual methods (= 0). This sets up the class to be an abstract class and thus no instances can be created. It serves the purpose of providing required functionality to sub classes.
#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. */
void setColor(const int r, const int g, const int b);

string toString() const;

#endif /* defined(__cppProject__Color__) */