# C++
# Raymond Klefstad, Ph.D.
# Inheritance and Polymorphism

---

## Inheritance

- a class has certain attributes
  - methods
  - constructors/destructors
  - member functions
- *base class*
  - AKA superclass or parent class
  - defines member data and member functions
- *derived class*
  - AKA subclass or child class
  - inherits all the members from its base class
  - may add members
  - may override member functions to change behavior
- 

  ---
  EG
  ```
  class Vehicle {...};
  class Car : public Vehicle {...};
  ```
- Vehicle is the *base class*
- Car is the *derived class*
- this kind of type derivation is called *inheritance*

---

## Hierarchy, Abstraction, and Inheritance

- humans use hierarchy and abstraction to manage complexity
  `AnimalTaxonomy, Vehicles, Houses`
- we use a graph to show hierarchy
  - root is most abstract (general)
  - leaves are more concrete (specialized)
- hierarchy reflects the "kind of" relationship
  - a Bus is a kind of Vehicle
  - a Car is a kind of Vehicle
  - a Sedan is a kind of Car
- 

  ---
  inheritance allows reuse of common attributes and operations
  - all Mamals are warmblooded and suckle their young
  - all Vehicles may be driven
  - all Houses have an address and electricity
  - all Shapes have Origin and Color, can be drawn, moved, etc
- identifying classes

- *classical categorization*
    - group objects by common properties (attributes)
    - EG has four wheels, a steering wheel, a break, a gas pedal

---

## Class Relationships

- inheritance
  - shows "kind of" relationship
  - a Car is a kind of Vehicle
  - done in C++ via public inheritance
    ```
    class Car : public Vehicle { ... };
    ```

- containment (using)
  - shows "part of" (or "has a") relationship
  - an Engine is a part of a Car
  - done in C++ via a data member
    ```
    class Car : public Vehicle { Engine e; ... };
    ```

---

## Inheritance in C++

- class header may include a derivation list:
  ```
  class Screen { ... };
  class Window : public Screen { ... };
  ```

- Screen is a public base class of Window
- Window is *derived* from Screen
- Window inherits data and member functions from Screen
- derived class can be a base class
  ```
  class Menu : public Window {...};
  ```

- Menu inherits data and member functions from Window

---

## Example: Vector

- class Vector implements an unchecked, uninitialized array of ints
  ```
  class Vector
  {
    int *buf;
    int sz;
  public:
    Vector (int s)
      : sz (s), buf (new int[s])
    {
    }
    ~Vector()
    {
      delete buf;
    }
  ```

```
  int size()
  {
    return sz;
  }
  int &operator[] (int i)
  {
    return buf[i];
  }
};
int main()
{
  Vector v(10);
  v[6] = v[5] + 4; // oops, no init values
  int i = v[10]; // oops, out of range!
  //...
}
```

---

## Benefit of Inheritance

- inheritance allows you to extend a type hierarchy
- you need not modify the source code for the rest of the system
- EG we need a vector whose bounds are checked when indexing
- derive a new class CheckedVector
- it inherits characteristics of base class Vector
- we can add to or modify characteristics as needed

---

## Example: Range Checked Vector

- EG

```
class CheckedVector
  : public Vector
{
public:
  CheckedVector(int s)
    : Vector(s)
  {
  }
  int &operator [](int i)
  {
    if (i < 0 || i >= size())
      throw range_error();
    else
      return (*(Vector *)this)[i];
  }
  // Vector::size() and ~Vector are inherited from Vector
};
int main()
{
  CheckedVector v(10);
```

```
  int i = v[10]; // Error detected!
}
```

---

## Data Hiding and Derived Classes

- derived class can access public and protected members of base class
- derived class MAY NOT access private members
- protected members should hide representation from derived classes
- this protects derived classes from base class representation change

```
class Vector
{
  int *buf;
  int sz;
protected:
  // allow derived classes direct access
  int &element(int i) { return buf[i]; }
  int in_range(int i) { return i>=0 && i < sz; }
  int &vector_size() { return sz; }
public:
  int &operator [](const int i)
           { if (in_range(i)) return element(i); }
  };
```

---

## Type and Subtype Relationships

- derived class introduce a subtype of base class
- pointer or reference to base may refer to any derived instance

```
Menu m; Window &w = m; Screen *ps = &w;
```

- this allows polymorphic programming

```
void driveAll( Vehicle * a[], int n)
{
  for ( int i = 0; i < n; i++ )
    a[i]->start(); // start depends on kind of Vehicle
}
```

- new subtypes can be added to a system without changing the rest of the system
- example uses
  - adding a new stack or queue representation to your holder library
  - adding an AVL tree to your table library
  - adding a new car to your vehicle hierarchy
  - adding a new type of menu to your GUI widget set

---

## Subtype Example

- EG

```
extern void dump_image (Screen &s);
Screen s;
Window w;
```

```
Menu m;
Bit_Vector bv;
dump_image(w); // OK: Window is a kind of Screen
dump_image(m); // OK: Menu is a kind of Screen
dump_image(s); // OK: argument types match exactly
dump_image(bv); // Error: Bit_Vector not a kind of Screen!
```

---

## Dynamic vs. Static Binding

- consider the following:
```
CheckedVector cv(20);
Vector &vp = cv;
do_something_with(vp[0]);
```

- which version of operator [] is called?
  - *static binding*
    - operator is chosen at compile time based on declared type of vp
    - calls Vector::operator[]
  - *dynamic binding*
    - decision is deferred until run-time when actual type of object is known
    - calls CheckedVector::operator []

---

## Dynamic Binding

- dynamic binding is used only for virtual member functions
```
struct Base {
  virtual int virtual_fn();
  int non_virtual_fn();
};
```

- when over-riding a virtual function in a derived class, virtual is optional
```
struct Derived : public Base {
  int virtual_fn(); // still virtual
  int non_virtual_fn();
  // ...
};
```

- preferred style is to include virtual when overriding

---

## Use of Dynamic or Static Binding

- static binding:
  - useful when dealing with homogeneous set of similar objects
  - inheritance here allows reuse of portions of base class
- dynamic binding:
  - useful when dealing with a heterogeneous mix of objects
  - they share common attributes and/or operations
  - implementation of attribute may vary with each object

    - inheritance here allows mixing of various similar objects
- static binding examples
    - Vector, CheckedVector, InitVector, InitCheckedVector
- dynamic binding examples:
    - screen, window, menu of widget toolkit
    - holders like stack, queue, deque, bag
    - tables like array, hash_table, search_list, binary_search_tree
    - symbols, operators, AST, or intermediate codes in a compiler

---

## Example Use of Dynamic Binding

- a shape hierarchy in a GUI (graphical user interface)
- shapes, like Circle, Square, Rectangle, and Triangle, are derived from a base class, Shape
- class Shape defines common member functions:
    - Point where(); // return coordinates of a Shape
    - void move(Point to); // move a Shape to new coordinates
    - void rotate(int degrees); // rotate the Shape by a specified degree
    - void draw(); // draw the Shape on the screen
- in C, we would use a union or discriminated record to represent Shape
- a tag or discriminant indicates kind of shape in a Shape
- each Shape operation must switch on kind of shape
- EG

```
void rotate_shape(Shape *sp, int degrees)
{
  switch (sp->type_tag) {
    case CIRCLE:
      return;
    case SQUARE:
      /* rotate a square */
    /* ... */
}
```

---

## C++ Solution

- in C++, dynamic binding replaces switching on specific kind of object

```
class Shape {
public:
  virtual void rotate(int degrees);
};
class Circle : public Shape {
public:
  virtual void rotate(int degree) { /* nothing - noop */ }
};
class Rectangle : public Shape {
public:
  virtual void rotate(int degree);
};
```

- any Shape can now be rotated independent of specific method of rotation
- can be done with pointer or reference

```
void rotate_shape(Shape *sp, int degrees)
{
  sp->rotate(degrees);
}
OR
void rotate_shape(Shape &sp, int degrees)
{
  sp.rotate(degrees);
}
```

---

## Extensibility

- virtual functions allow you to define **polymorphic** operations
- can add to type hierarchy without modifying polymorphic operations

```
class Square : public Rectangle {
public:
  virtual void rotate(int degree)
  {
    if (degree % 90 != 0)
      Rectangle::rotate(degree);
  }
};
```

- we can still rotate any Shape object by saying

```
void rotate_shape(Shape *sp, int degrees)
{
  sp->rotate(degrees);
}
```

---

## Extensibility (cont.)

- in C, we must modify every function dealing with Shape
- we must add a new case for new object to each switch

```
void rotate_shape(Shape *sp, int degree)
{
  switch (sp->type_tag) {
    case CIRCLE:
      return;
    case SQUARE:
      if (degree % 90 == 0)
        do_rectangle_rotation();
  /* ... */
}
```

- C approach prevents adding Square if the code of rotate_shape() can't be modified

## Subtyping vs. Reuse

- inheritance can be used for different purposes:
  - to allow dynamic binding
    - EG Circle is a subclass of Shape
  - to allow extension/modification of an existing class
    - EG CheckedVector that inherits from Vector
  - to allow reuse of an implementation
    - Stack that inherits from Vector

## public and protected Inheritance

- implementation is a misuse of public inheritance
  - no subtype/kind-of relationship to base class
  - operations on base class may not apply to derived
  - EG array subscripting into a stack??
- private inheritance
  - base class public and protected members become private in derived
  - only members/friends can convert to base class reference
- protected inheritance
  - base class public and protected members become protected in derived
  - members/friends and derived classes can convert to base reference

## Abstract Base Classes

- a *base class* is a (sub)root of an inheritance hierarchy
- may contain function stubs called *pure virtual functions*
- a class with pure virtual functions is called an *abstract base class*

```
class Shape {
public:
  Shape(int x = 0, int y = 0);
  virtual void move(Point to);
  virtual void draw() = 0;
  virtual void rotate(int degrees) = 0;
};
```

- draw() and rotate() can't be written yet, but move() can
- can declare only references or pointers to abstract class

## Virtual Function Example

- // Abstract Base Class and Derived Classes

```
class Shape {
private:
  Point shape_center;
  Color shape_color;
public:
  Point where() const { return shape_center; }
```

```
  void move(const Point &to) { shape_center = to; draw();
}
  virtual void draw() const = 0;
  virtual void rotate(int degrees) = 0;
};
class Circle : public Shape {
private:
  int radius;
public:
  void draw(); // Code to draw a circle
  void rotate(int degrees) { /* do nothing */ }
};
class Rectangle : public Shape {
private:
  int width;
  int length;
public:
  void draw(); // Code to draw a Rectangle
  void rotate(int degrees); // Code to rotate a Rectangle
};
```

---

## Polymorphism

- polymorphism
  - when the specific operation invoked by a call depends on the type of an object
- static polymorphism - via overloading
- dynamic polymorphism - via virtual functions
  - useful for dealing with sets of objects having similar interface, but different implementations
  - Vehicles, Shapes, GUI Widgets

---

## Polymorphic Function Example

- example function that rotates all size shapes by angle degrees:

```
void rotate_all(Shape *vec[], int size, int angle)
{
  for (int i=0; i < size; i++)
    vec[i]->rotate(angle);
}
```

- vec[i]->rotate() is a virtual function call resolved at run-time
- which *rotate* depends on actual type of shape in vec[i]

---

## Virtual Function Example (cont.)

- example use of function rotate_all()

```
Shape *shapes[] = {new Circle, new Square, new Rectangle};
int size = sizeof shapes / sizeof *shapes;
rotate_all(shapes, size, 90);
```

- specific types of shapes are unknown until run-time
- however, they are all derived from common base class Shape

---

## Virtual Base Classes

- a base class may appear only once in a derivation list
- however, a base class may appear multiple times within a derivation hierarchy
- this presents two problems with multiple inheritance:
    - it may introduce member function and data object ambiguity
    - it may also cause unnecessary duplication of storage
- 'virtual base classes' are included only once even if repeated

---

## Virtual Multiple Inheritance

- a class can be simultaneously derived from two or more base classes
- EG

```
class CheckedVector : public virtual Vector {
  /* ... */
};
class InitVector : public virtual Vector {
  /* ... */
};
class InitCheckedVector : public CheckedVector, public
InitVector {
  /* ... */
};
```

- the virtual keyword prevents two copies of Vector in InitCheckedVector
- virtual base classes have certain restrictions:
    - they must possess constructors that take no arguments
- understanding and using virtual base classes can be difficult

---

## Multiple Inheritance Ambiguity

- member names can conflict in multiple inheritance

```
struct Base1 { int foo(); /* ... */ };
struct Base2 { int foo(); /* ... */ };
struct Derived : Base1, Base2 { /* ... */ };
int main()
{
  Derived d;
  d.foo(); //Error, ambiguous call to foo ()
}
```

- two ways to fix this problem:
    - qualify the call with the name of the class and the scope qualifier, e.g.,
      `d.base1::foo();`
    - add a new member function foo to class Derived, e.g.,

```
struct Derived : Base1, Base2 {
  int foo() {
    base1::foo();
    base2::foo();
  }
};
```

---

## Type and Subtype Conversion

- a derived class can add new members not be defined in base class
- consider the following derivation hierarchy,

```
struct Base {
  int i;
  virtual int foo() { return i; }
};
struct Derived : public Base {
  int j;
  int foo() { return j; }
};
void f() {
  Base b;
  Derived d;
  Base *bp = &d; // OK, a Derived is a Base
  Derived *dp = &b;// Error, a Base may not be a Derived
}
```

- Derived contains a Base and operations are well defined

```
bp->i = 10;
bp->foo();
```

- Base does not contain a Derived and operations aren't defined

```
dp->j = 20;
```

- C++ permits conversion from Base to Derived

```
dp = static_cast<Derived *>(&b);
```

- programmer must insure dp operations don't access undefined members

```
dp = dynamic_cast<Derived *>(&b); // throws bad_cast
exception
```

---

## Extended Example (Static Binding)

- Geometric Shape Hierarchy

```
class Shape
{
protected:
  Point origin;
  Color color;
public:
  Shape( Point newOrigin, Color newColor )
```

```
      : origin( newOrigin ), color( newColor )
    {
    }
    void moveTo(Point to)
    {
      origin = to;
    }
  };
```

- derived class Circle

```
class Circle
  : public Shape
{
private:
  const double PI = 3.14159;
protected:
  double radius;
public:
  Circle( double newRadius, Point newOrigin, Color
newColor )
    : Shape( newOrigin, newColor ), radius( newRadius )
  {
  }
  // inherits moveTo from Shape
  double circumference()
  {
    return 2.0 * PI * radius;
  }
};
```

- derived class can access public and protected members
- cannot access private members
- note: base class constructor must be called

---

## Creating And Destroying Derived Classes

- derived class must call base constructor
- otherwise, no arg constructor is called
- order of construction: base classes then new data members
- **virtual destructor is required** for hierarchy with virtual functions
- virtual destructor must be introduced at root class
- in case any derived class requires a destructor
- may be eliminated if no destruction will ever be needed

---

## Extended Example (Dynamic Binding)

- Pictures containing 2-D Geometrical Shapes

```
class Shape
{
```

```
public:
  virtual double area() = 0;
  virtual ~Shape() {} // note required virtual destructor
};
```

- ---

  Triangle

  ```
  class Triangle
    : public Shape
  {
  protected:
    double base;
    double height;
  public:
    Triangle( double newBase, double newHeight )
      : base( newBase ), height( newHeight )
    {
    }
    virtual double area()
    {
      return base * height / 2.0;
    }
  };
  ```

- ---

  Square

  ```
  class Square
    : public Shape
  {
  protected:
    double side;
  public:
    Square( double newSide )
      : side( newSide )
    {
    }
    virtual double area()
    {
      return side * side;
    }
  };
  ```

- ---

  Linked List of Shapes

  ```
  typedef class ShapeLinkedListPair * ShapeLinkedList;
  class ShapeLinkedListPair
  {
  public:
    Shape * info;
    ShapeLinkedList next;
  ```

```
  ShapeLinkedListPair( Shape * newInfo, ShapeLinkedList
newNext )
    : info(newInfo), next( newNext )
  {
  }
};
```

- ---

Picture (a List of Shapes)

```
class Picture
{
  ShapeLinkedList head;
public:
  Picture()
    : head( 0 )
  {
  }
  void enter( Shape * a )
  {
    head = new ShapeLinkedListPair( a, head );
  }
  double totalArea()
  {
    double total = 0.0;
    for (ShapeLinkedList p = head; p != 0; p = p->next)
      total += p->info->area();
    return total;
  }
};
```

- ---

main program

```
int main()
{
  Picture p;
  p.enter( new Triangle(10,10) );
  p.enter( new Square(10) );
  cout << "Total area = " << p.totalArea() << endl;
}
```