C++
Inheritance and Polymorphism

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 then itself 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 class hierarchy
- You need not modify the source code for the rest of the system
- Example: 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

The best code is code you don't have to write at all
(inheritance facilitates this)

Example: Range Checked Vector

```
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]; // invoke Vector::operator[]
  }
  // 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
- These features *protect 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
}
```

Type and Subtype Relationships (cont'd)

- 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

Types vs Classes

- **Types** exist outside of OOP and correspond to mathematical sets
  - subtype = subset
  - e.g., naturals are a subtype (subset) of integers
- **Classes** are germane to OOP
  - subclass = particular kind of subset
  - more specialized behavior restricts the range of objects populating the set
  - e.g.,

```
class Vehicle : public Object
class Car : public Vehicle
```

```
    Car    <:    Vehicle    <:    Object
```

(more specialized) (less specialized)

Subtype Example

```
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

```
class Base {
  protected:
   virtual int virtual_fn();
   int non_virtual_fn();
};
```

- when over-riding a virtual function in a derived class, virtual is optional

```
class Derived : public Base {
  protected:
   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 to represent `Shape`
  - a tag indicates kind of shape in a `Shape`
  - each `Shape` operation must switch on kind of shape
  - error-prone (tag---operation link not enforced by the compiler)
  - e.g.,

```
void rotate_shape(Shape *sp, int degrees)
{
  switch (sp->type_tag) {
    case CIRCLE:
      do_rectangle_rotation(); // compiler says A-OK
    case SQUARE:
      do_circle_rotation();    // compiler says A-OK
}
```

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) { }// no-op
};
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'd)

- 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 (e.g., is in a library)

Where is inheritance useful?

Inheritance can be used for different purposes:

- to allow dynamic binding
  - e.g., `Circle` is a subclass of `Shape`
- to allow extension/modification of an existing class
  - e.g., `CheckedVector` that inherits from `Vector`
- to allow reuse of an implementation
  - `Stack` that inherits from `Vector`

`public, protected, private` Inheritance

- Implementation could be a misuse of public inheritance
  - When no subtype/kind-of relationship to base class
  - Operations on base class may not apply to derived
  - e.g., `class Stack : public Vector`
    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
- cannot instantiate `Shape`
  - 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'd)

- 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
- Example:

```
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

```
class Base1 { int foo(); /* ... */ };
class Base2 { int foo(); /* ... */ };
class Derived : Base1, Base2 { /* ... */ };
int main()
{
  Derived d;
  d.foo(); //g++: error: request for member 'foo' is ambiguous
}
```

- 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.,

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

Type and Subtype Conversion

- A derived class can add new members not defined in base class, e.g.

```
class Base {
  protected: int i;
             virtual int foo() { return i; }};
class Derived : public Base {
  protected: int j;
             int foo() { return j; }};
void f() {
  Base b;
  Derived d;
```

- Upcasting: always OK
  - `Derived` contains a `Base` and operations are well defined

```
Base *bp = &d; // OK, a Derived can be a Base
bp->i = 10; bp->foo();
```

- Downcasting: programmer must ensure `dp` operations don't access undefined members
  - `Base` does not contain a `Derived` and operations aren't defined

```
dp = static_cast<Derived *>(&b); dp->j = 20; // compiles, but undefined behavior
// static_cast = "compiler, trust me"
Derived *dp = &b; // g++: error: invalid conversion from 'Base*' to 'Derived*'

dp = dynamic_cast<Derived *>(&b); // g++ warning:dynamic_cast of 'Base b' to 'class
  Derived*' can never succeed
```

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, `Shape()`, must be called

Creating And Destroying Derived Classes

- derived class must call base constructor
  - otherwise, the default constructor (no args) 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 (rare)