Lecture 16

- Vector class
- Class inheritance
- Virtual functions

Shallow vs. Deep Copy

- If object only contains simple types or pointers that are shared among objects, bit-wise (=shallow) copy is O.K. - no need to define the copy constructor and assignment operator
- Otherwise, use deep-copy: recursively clone data members
- Make sure there are **no resource leaks** and **no self-assignments**!

```
class X { public:
   X &operator= (const X &x) {
      if (this == &x) return *this; // self-assignment!
      ... // release current resources and copy x
      return *this
   }
};
```


Vector class that requires definitions

```
#include <iostream>
using namespace std;

class V {
public:

    V(int n_) { alloc(n_); } // creates vector of n_ elements
    V(const V& x) { copy(x); }
    V &operator=(const V &x) { free(); copy(x); return *this; } // BUGGY!
    ~V() { free(); }

    int size() const { return n; } // return #elements in vector

private:
    int n;    // number of elements
    int *p;   // vector has its own array, thus bit-copy does not work!

    void alloc(int n_) { n = n_; p = new int[n]; } // allocates array

    void free() { delete [] p; }                   // releases array

    void copy(const V &x) {                        // copies array
       alloc(x.size()); for (int i=0; i < n; ++i) p[i] = x.p[i];
    }
};
```


Class Inheritance

- Object Oriented Programming Paradigm
- Derive new class from existing base class(es)
  - inherits data and function members from base class(es)
  - **code/data reuse**
    (functions and data are inherited from base classes)
  - **code adaption** (make use of base class impl.)
- Single inheritance (inherit from one base class)
- Multiple inheritance (more than one, rare)

Inheritance Example

- Sub-class/derived class **specializes** super-class/base class
- Usually models **"Is A"** relationship
- E.g. "a **Rectangle** is a **Shape**",
  "a **Square** is a **Shape**",
  "an **Ellipse** is a **Shape**",
  "a **Circle** is a **Shape**"
- Type hierarchy:
  - **Shape** <--+---- **Ellipse**
    +---- **Circle**
    +---- **Rectangle**
    +—— **Square**

Example: Shapes

```
class Shape {           // base class
public:
   int color;           // all shapes have a color
   float area() const { return 0; } // and area, too
};

class Rectangle : public Shape {
private:
   int xl, xr, yt, yb; // describes a Rectangle
                       // also inherits color
public:
   Rectangle(int xl_, int xr_, int yt_, int yb_):
    xl(xl_), xr(xr_), yt(yt_), yb(yb_) {}

   // overrides Shape::area()
   float area() const { return (xr-xl)*(yb-yt); }
};

class Circle : public Shape {
private:
   int x, y, r;        // describes a Circle
                       // also inherits color
public:
  Circle(int x_, int y_, int r_) :
    x(x_), y(y_), r(r_) {}

  float area() const { return r * r * PI; }
};
```


Inheritance Types

- Derived class inherits all data and function members from base class(es)
- Access permissions depend on qualifiers
- class Y : **public** X { ... }
  - **Y "is an" X**
  - Sub-class Y can **access public** and **protected** members of X, **cannot** access private members of X
- class Y : **protected** X { ... }
  - **Y "is implemented in terms of" X**
  - public members of X become protected in Y

Example

```
class X {

public:
   int a;            // visible to all: users of X,
   void fa();        // X itself, and derived classes
protected:
   int b;            // visible to derived classes & X,
   void fb();        // but not to users of class X!
private:
   int c;            // only visible to member functions
   void fc();        // of X
};                                      |  int main() {
                                        |     X x;
class Y : public X { // Y "is an" X      |     Y y;
                                        |     x.a = 0; // OK
   void foo() {                         |     y.a = 0; // OK
      a = 0; fa();   // OK              |     x.b = 0; // NOT OK
      b = 0; fb();   // OK              |     x.c = 0; // NOT OK
      c = 0; fc();   // NOT ALLOWED!    |  }
   }                                    |
};                                      |
```

Graphics Example

- Class Graphics contains a list of pointers to objects to be drawn: Circles, Rectangles, ...
- First solution: Objects contain their type-id

```
class Shape {
public:
  int type_id;
  int color;
};

enum { CIRCLE, RECTANGLE, TRIANGLE, ... };

class Circle : public Shape {
  int x,y,r;
public:
  Circle() { x=y=r=0; type_id = CIRCLE; }
  void draw(Screen *s) const { ... }
  ...
}
```



```
class Graphics {
public:
  void draw() {       // draw all objects
    for (int i=0; i < n_objs; ++i) {
      Shape *p = objs[i];
      switch(p->type_id) {
      case CIRCLE:
        static_cast<Circle*>(p)->draw(screen);
        break;
      case RECTANGLE:
        static_cast<Rectangle*>(p)->draw(screen);
        break;  ...
    }
  }
  Shape **objs;     // array of pointers to Shapes
  int n_objs;       // number of objects
  Screen *screen;
};
```

Problems: slow + need to change code when adding new shapes, hard to maintain

Virtual Functions

- For a base class pointer, execute member functions in the current object context [**Shape *p;... p->draw()**]
- **Polymorphism**: same function name, different action
- Objects must know their type!
- Solution: **Virtual Functions**

Graphics 2

```
class Shape {    // abstract base class
public:
  int color;
  virtual void draw(Screen *s) const = 0; //abstract
};   // = 0: derived classes must implement function

class Circle : public Shape {
public:
  Circle() { x = y = r = 0; }
  void draw(Screen *s) const { ... } // implements
private:                             // virtual function
  int x,y,r;
}
```

- Second solution: virtual function draw
- Keyword **virtual** indicates that the function in sub-classes is accessible via base-class pointers

Solution 2

```
class Graphics {
public:
   void draw() {     // draw all objects
      for (int i=0; i < n_objs; ++i) {
         objs[i]->draw(screen);
      }
   }
   Shape **objs;     // array of pointers to Shapes
   int n_objs;       // number of objects
   Screen *screen;
};
```

- No type_id, no switch. Faster and easy to maintain
- Type of *objs[i] known at runtime => the correct draw function can be called. **HOW?**

Virtual Function Implementation

- New data-member is added to class variables
  - pointer to virtual function table (VFTP)
- One virtual function table is created for each class
- The virtual function table contains addresses of virtual functions
- Two stage access: Shape *p; **p->draw(screen);** calls **(*p->VFTP[C_DRAW])(screen);**