Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 16

- C++ Class Inheritance
- Assignments
- ctor, dtor, cctor, assignment op. and Inheritance
- Virtual Functions

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
  - Code adaption (make use of base-class impl.)
- Single inheritance (inherit from one base-class)
- Multiple inheritance
  (more than one base-class, not in Java/C#!)

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 <---+--- Rectangle
          +--- Square
          +--- Ellipse
          +--- Circle
```



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

class Rectangle : public Shape {
public:
  Rectangle(int xl_, int xr_, int yt_, int yb_) :
    xl(xl_), xr(xr_), yt(yt_), yb(yb_) { }

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

private:
  int xl, xr, yt, yb; // describes Rectangle (left,right,top,bottom)
                      // also inherits color
};

class Circle : public Shape {
public:
  Circle(int x_, int y_, int r_) : x(x_), y(y_), r(r_) { }
  float area() const { return r * r * PI; } // overrides Shape::area()

private:
  int x, y, r; // describes a Circle, also inherits color
};
```

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


```
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
};
```

```
class Y : public X // Y "is an" X
{
  void foo() {
    a = 0; fa(); // OK
    b = 0; fb(); // OK
    c = 0; fc(); // NOT OK!
  }
};
```

```
int main() {
  X x;
  Y y;
  x.a = 0; // OK
  y.a = 0; // OK
  x.b = 0; // NOT OK
  x.c = 0; // NOT OK
}
```


Assignments Across Class Hierarchy

`class Y : public X {...}`

- Y inherits data and function members from X
- Public inheritance: "is-a" relationship
- public and protected X members visible in Y

`X a; Y b;`

- Assignments: a = b; or b = a; meaningful?
- How to implement Y assignment operator and copy constructor?

`X *pa; Y *pb;`

- Assignments: pa = pb; or pb = pa; meaningful?

Object Assignment

```
class Y : public X {...};
X a; Y b;

a = b;  // OK - but slicing!
```

- assignment operator is called with reference to b
- X-parts of b are copied to a, Y parts are not

`b = a; // not OK`

- Y can contain more data than X
- How to fill the rest?

Y assignment op. and copy constructor can make use of X operators

Pointer Assignment

```
class Y : public X {...};
X *pa; Y *pb;
```

- `pa = &b;` or `pa = pb;` `// OK`
  - pa now points to b, or *pb respectively
  - information about Y is not available when accessing *pa
- `pb = &a;` or `pb = pa;` `// not OK`
  - *pb is object of type Y
  - again, where would the additional data come from?

Reusing Base-Class Operators

```
struct X {
  int x;
  X() { x = 0; }
};

struct Y : public X {
  int y;
  Y() { y = 0; }

  Y(const Y &a) : X(a) {      // X copy constructor, copy X-part
    y = a.y;                  // copy Y-part
  }

  Y &operator=(const Y &a) {
    X::operator=(a);          // X assignment operator, copy X-part
    y = a.y;                  // copy Y-part
    return *this;
  }
};

X a, *pa;
Y b;
a = b;      // a.x = b.x; b.y not copied (object "slicing")
pa = &b;    // OK, *pa is object of type X. Y-parts invisible
```


Inheritance and Constructors

```
class X {
public:
  X(int a_=0) { ... }
};

class Y : public X {
public:
  Y() { /* X() is called here */ ... }
  Y(int b_) : X(b_) { ... } // explicit X(int) call
};
```

- Base-class constructors, copy constructors, and assignment operators are not inherited!
- Derived class constructor calls the base-class constructor first to initialize base-class members
- If ommitted, the default derived class constructor is the base-class constructor

Inheritance and Destructors

```
struct X {  // struct = class ... public:
  int *p;
  X()  { p = new int[100]; }
  ~X() { delete [] p; }
};

struct Y : public X {
  int *q;
  Y() { /*X() called here*/ q = new int[200]; }
  ~Y(){ delete [] q; /* ~X() called here*/ }
};
```

- Are called in reverse order of constructor calls
- Derived class destructor `~Y()` calls base-class destructor `~X()` at the end
- `~Y()` only deals with resources allocated in Y! `~X()` takes care of the rest

Using Inheritance: Graphics Example

- Class Graphics contains a list of pointers to objects to be drawn: Circles, Rectangles, ...
- 1st solution: Objects contain an id to identify type

```
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 = new Circle; p->draw();`
  Would be nice if this calls `Circle::draw` !
- Polymorphism: same function name, different action
- Requires that objects "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
                                     // virtual function
private:
  int x,y,r;
}
```

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

```
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 $\rightsquigarrow$ 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);` replaced by `(*p->VFTP[C_DRAW])(p, screen);`