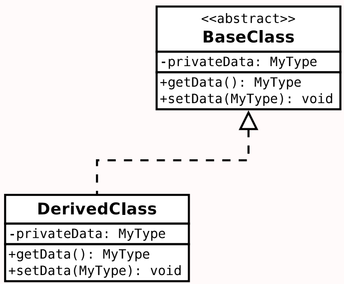
Inheritance & Polymorphism in C++

January 31, 2016

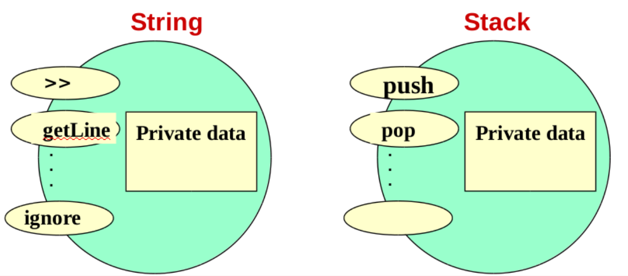
Brian A. Malloy



Slide 1 of 20
Go Back
Full Screen
Quit

1. C++ Class

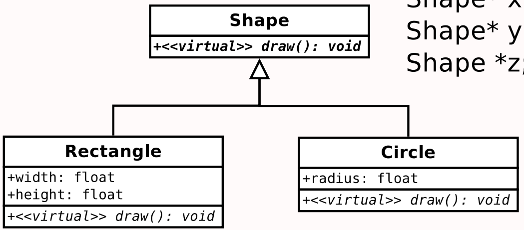
- Unit of Encapsulation
- Defines an Abstraction
- Specifies a user-defined type
- A type defines an interface



2. Inheritance

- Unit of reuse
- A derived class defines a subtype that inherits the interface of the base type
- Inheritance models is-a

2.1. Examples of Types and Subtypes




```
Shape* x = new Rectangle;
Shape* y = new Circle;
Shape *z;
```

1. `z` can be many types
2. `x` and `y` are different objects but same type
3. Rectangle is a subtype of Shape
4. Rectangle inherits the interface of Shape

2.2. Dynamic Binding

- When a request is sent to an object, the operation that's performed depends on:
  1. The request
  2. The receiving object
- Different objects that support the same requests may have different implementations of the request. e.g., x and y in previous example are different objects that both support draw, but have different implementations of draw.
- The run-time association of a request to an implementation is `dynamic binding`
  Shape *s;
  s $\rightarrow$ draw();

3. Base Class Functions

- A base class can have 3 types of functions:
  - non-virtual
  - virtual
  - pure virtual
- A non-virtual function should not be overridden
- A virtual function may be overridden
- A pure virtual function must be overridden

3.1. Example of 3 functions

```
class Shape {
public:
  std::string getName() const;
  virtual void getArea() { return 0; }
  virtual void draw() = 0;
};
```

- getName() is a <u>non-virtual</u> function
- getArea() is a <u>virtual</u> function
- draw() is a <u>pure virtual</u> function
- A class with a *pure virtual* function is abstract and cannot be instantiated.
- Classes derived from an abstract base class must implement the pure virtual functions in the base.

4. Polymorphism

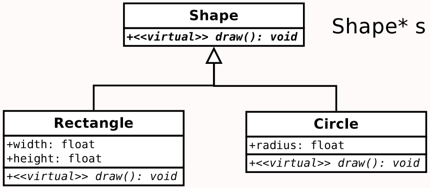
- Dynamic binding means that a request is bound to an implementation at run-time
- Dynamic binding permits substitution of objects with identical interfaces at run-time
- This substitutability is aka `polymorphism`.
- Polymorphism simplifies the definition of clients, decouples objects from each other, and lets them vary their relationship at run-time.

4.1. Polymorphism in C++

- The form of a C++ declaration is:
  type *variable*;

  ```
  int x;
  Shape * s;
  ```

- s can point to any object in the hierarchy because it's type is pointer to base class.
- Ex: it can point to a Rectangle:



Shape* s = new Rectangle;

4.2. Polymorphism in C++

- Consider: `Shape* s = new Rectangle;`
- The type of s is pointer to Shape,
- but the type of the object that it can point to is: Shape, Rectangle, or Circle.
- In other words, a pointer of type base can point to any object in the inheritance hierarchy.
- This is polymorphism and it's power comes from virtual functions.

4.3. Variable vs Object

- If a variable points to the base class, it can legally point to any object in the inheritance hierarchy.
- This is polymorphism: the variable can take *many forms*.

```
Shape* s = new Rectangle;
```

type of the variable is Shape*

type of the object s points to is Rectangle

4.4. Code Example of Polymorphism

```
class Shape {
public:
  Shape(const std::string& n) : name(n) {}
  const std::string& getName() const { return name; }
  virtual float getArea() const = 0;
private:
  std::string name;
};

class Circle : public Shape {
public:
  Circle(const std::string& n, float r) : Shape(n), radius(r) {}
  virtual float getArea() const { return 3.14*radius*radius; }
private:
  float radius;
};

class Rectangle : public Shape {
public:
  Rectangle(const std::string& n, float w, float h) :
    Shape(n), width(w), height(h) {}
  virtual float getArea() const { return width*height; }
private:
  float width, height;
};
```

```
void printArea(const Shape* s) {
  std::cout << "Area of " << s->getName() << " is " << s->getArea();
  std::cout << std::endl;
}

int main() {
  printArea(new Circle("circle", 5.0));
  printArea(new Rectangle("rectangle", 5.0, 4.0));
}
```

- printArea polymorphically accepts either Circle or Rectangle
- We cannot instantiate Shape because draw is *pure virtual*, which means that Shape is *abstract*.
- Since getArea is *virtual*, calls to this function are dynamically bound to the type of the object, not to the type of the variable.

5. Program to an Interface, not an Implementation

- When inheritance is used carefully, all classes derived from an abstract class will share its interface
- This implies that a subclass merely adds or overrides operations (adding is okay in Smalltalk, but problematic in C++)
- All subclasses can then respond to the requests in the interface of this abstract class
- Two benefits of this:
  1. Clients know only the abstract class
  2. Clients remain unaware of specific types

- "Program to an interface" really means program to a super type.
- This reduces implementation dependencies so dramatically between subsystems that it leads to:
  First principle of reusable OO design:
  Program to an interface,
  not an implementation
- Don't declare variables to be instances of particular concrete classes
- This is a common theme of design patterns

6. Inheritance v Composition

- The two most common techniques for reuse:
  1. `Class Inheritance`: define the implementation of one class in terms of another.
  2. `Object Composition`: new functionality by assembling or composing objects.
- `White box reuse`: reuse by subclassing
- `Black box reuse`: reuse by composition

6.1. Tradeoffs: Inheritance over Composition

- Advantages:
  - Defined statically (compile time)
  - Straightforward
  - Supported by programming language
  - Easier to modify the implementation: override some of the operations.
- Disadvantages
  - Can't change implementation dynamically
  - Parent class defines at least part of its subclasses physical representation.
  - Because inheritance exposes a subclass

to details of its parent's implementation, it's often said that "Inheritance breaks encapsulation" (Snyder, OOPSLA 1986)

- Because the implementation of a subclass becomes bound to the implementation of the parent, a change to the parent requires a change to the subclass.
- If any aspect of the inherited implementation is inappropriate for new problem domains, the parent must be rewritten or replaced.
- One solution is to only inherit from abstract base classes.

6.2. Tradeoffs: Composition over Inheritance

- Advantages:
  - Defined dynamically (at run-time)
  - Requires objects to respect each others interfaces
  - Because objects are accessed solely thought their interface, composition does not break encapsulation.
  - Any object can be replaced at run-time by another (as long as it has the same type – strategy pattern)
  - Object composition keeps each class encapsulated and focused on one task.

- Class hierarchies remain small, less likely to grow into unmanageable monsters.

- Disadvantages:
  - A design based on object composition will have more objects and the system behavior will depend on their interrelationships instead of being defined in one class.

- Second principle of reusable object oriented design:
  Favor object composition over class inheritance.