

Programming Languages: OO Paradigm, Polymorphism and Class Members

Onur Tolga Şehitoğlu

Computer Engineering, METU

1 May 2009

Outline

- 1 Polymorphism
- 2 Abstract Classes

Polymorphism

- Inheritance → inclusion polymorphism
- Binding is still **static**, at compile time
- Pointers of derived classes are converted to superclass types

```
class A { int x;  
public: void get() { cout << 'A::get()';}  
};  
class B : public A { int y;  
public: void get() { cout << 'B::get()';}  
}  
...  
A a, *p;  
B b;  
p=&a; p->get();  
p=&b; p->get();
```

Late Binding

- Delaying binding possible

```
class A { int x;  
public: virtual void get() { cout << 'A::get()';}  
};  
class B : public A { int y;  
public: void get() { cout << 'B::get()';}  
}  
...  
A a, *p;  
B b;  
p=&a; p->get();  
p=&b; p->get();
```

- binding of **virtual** member functions done at run time.

Abstract Classes

- `void f() = 0 ;` makes the function an **abstract member**
- A class with at least one abstract member is an **abstract class**.
- Abstract classes cannot be instantiated
- A derived class remains abstract unless all abstract members are implemented somewhere in derivation chain.
- Java **interfaces**: abstract classes with only abstract member functions and constants.

- binding of `move()` is static but the `draw()`'s inside are still late.

```
class Shape { int x,y;
public: virtual void draw() = 0;
        void move(int a, b) {
                setbgcolor(); draw();
                x=a; y=b; setfgcolor(); draw();
        }
};

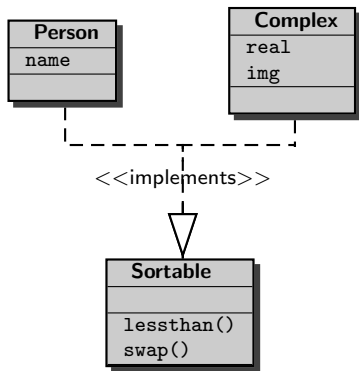
class Circle : public Shape { int r;
public: void draw() { /* draw circle here */ }
}

class Rectangle : public Shape { int w,h;
public: void draw() { /* draw rectangle here */ }
}

...
Circle a(...); Rectangle b(...);
a.move(2,4); b.move(3,4);
```

Interfaces

- Java does not have multiple inheritance but a class can **implement** multiple interfaces
- Functions working on interfaces provide polymorphism for the classes implementing them
- `Person` and `Complex` implements the interface `Sortable` so that `sort(...)` can work uniformly on both

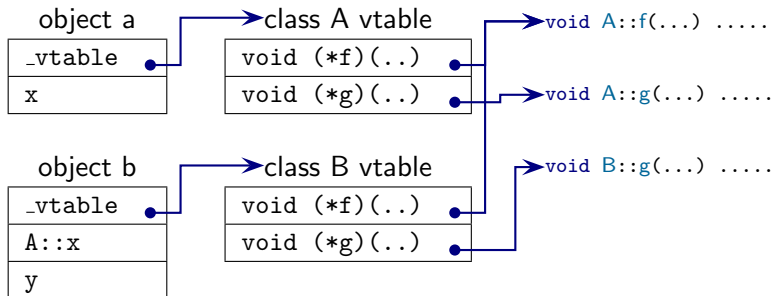


```
sort(Sortable a[], int n);
```

Implementation of virtual members

- For each class, a table for virtual member functions are kept globally (array of function pointers)
- Each **object** contains a pointer to its virtual function table
- Size of an object is : (size of member variables + pointer to virtual mem

```
class A { int x;  
public: virtual void f(...) {...}  
        virtual void g(...) {...}  
} a;  
class B : public A { int y;  
public: virtual void g(...) {...}  
} b;
```

one table per class

one pointer

per object

Assuming `p` points to an object of A or B, `p->g(..)`; call is mapped by the compiler as:

```
*((p->_vtable)[1])(...);
```

(assume 0 is the offset of `f`, 1 is the offset of `g`)

Generic Abstraction

- Abstraction over a declaration
- Polymorphism can be defined in terms of generic abstractions
- C++ templates
- Java generic classes

Templates (C++)

- Template metaprogramming approach:
All template definitions are expanded as they are **instantiated**
- Macro-like operation. Parameters can be an type or value.
- each **distinct** usage like `vector<Person> a` creates a new instance of the template class `vector`.
- All declaration body is expanded as an overloaded version.
- Functions can be declared with templates too. Each distinct typed call is a new instance, a new overload
- Very efficient but compiled code gets larger as different instances used
- Parametric polymorphism provided at compile time. Source code required.

Generics (Java)

- Restricts parameters to be classes. Primitive types and values does not work.
- Only one copy of the class and class functions exists.
- Type checking and verification done at compile time. Polymorphic code compiled in the binary.
- In Java: All object values are references, all member functions are virtual by default.
- Member functions of the parameter class are bound at run-time providing parametric polymorphism.

Class Members

- Members shared by objects of the same class. Only one copy per class.
- Assume you need a counter for each created object

```
int counter=0;

class A { int x;
public: A(int a) { x=a; counter++;}
       ~A()   { counter--;}
       int getcount() { return counter;}
};
```

- What is wrong with this code?

- **static** keywords make a member a class member

```
class A { int x;  
        static int counter;  
public: A(int a) { x=a; counter++;}  
        ~A() { counter--;}  
        int getcount() { return counter;}  
};  
int A::counter=0; // this is required to define the storage  
                 // it is scope of A
```

- Now the counter is safe. Arbitrary values cannot be assigned.
- Why do you need an object to call `getcount()`?

- Member functions can be class members too.

```
class A { int x;  
        static int counter;  
public: A(int a) { x=a; counter++;}  
        ~A()   { counter--;}  
        static int getcount() { return counter;}  
};  
int A::counter=0;
```

- Class members can be accessed with scope operator:
`A::getcount();`
- No object required. What if `getcount()` tries to access an object? You don't have one!
- Class member functions can only access other class members.
- Objects can access class members.