

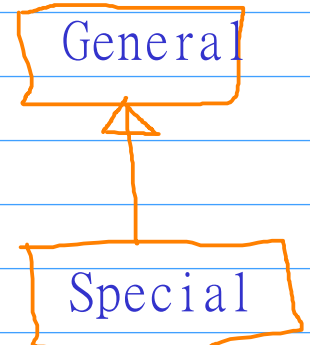
Inheritance

is - a

has - a, part - of
using

Examples:-

- * Vehicle and Car
- * Car -- Mini, Sedan, SUV
- * Person and Student
- * Person and Employee
- * Laptop and types of Laptops
- * Polygon and Rectangle, Triangle
- * Shape and Circle, Polygon
- * Image and PNG, JPEG images
- * Sensors and Proximity, Temperature, IMU/Motion
- * MotionSensor and Accelorometer, Gyroscope, Magnetometer
- * Employee and Engineer, Manager, Faculty, Trainee
- * Account and Saving, Current, Credit account
- * Customer and Prepaid, Postpaid



Composition : part-of

Aggregation : has-a

Association : using

Vehicle and Engine (part-of)

Vehicle and Wheel (part-of)

Library and Book (part-of)

Employee and Address (has-a)

Hospital and Doctor (part-of)

University and Student (part-of)

Student and Sports

Student and Library

Checkpoint:-

- * What is Inheritance
- * What is not Inheritance / Bad Inheritance
(When there is a scope of composition/aggregation,
When two objects just differ by data attribute)

learncpp : 23.1, 23.2, 23.3, 23.4

@startuml

class Person

{

}

class Student

{

}

Student --|> Person

@enduml

Types of Inheritance

- Single
- Multilevel
- Multiple

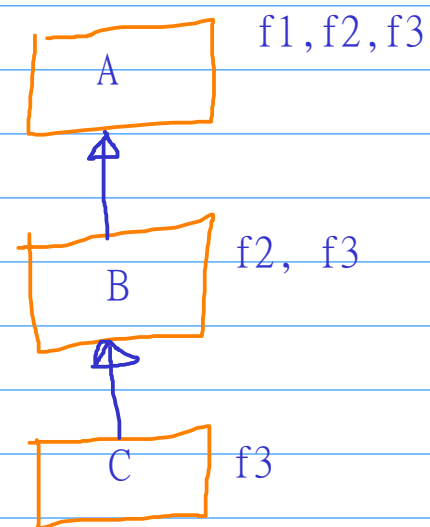
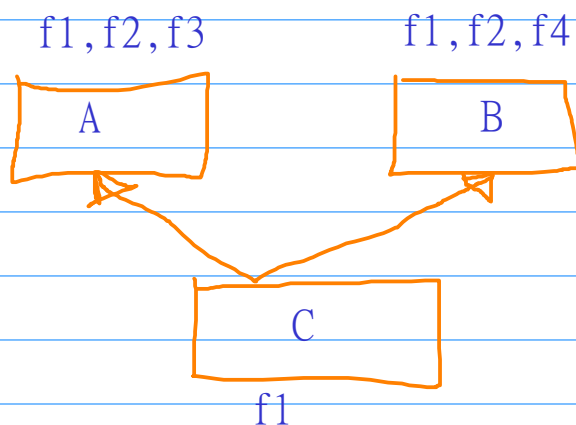
- Hierarchical
- Hybrid

Person

- Name
- UID/SSN
- Gender
- DOB
- + isMajor
- + isEligible
- + computeAge

Student

- rollno
- score
- + grade()
- + isPass()



Calling f2 from C -- ambiguous

Herbivores, Carnivore

Animal

Mammal

Carnivore

Lion

Student inherits from Person, Sports

Student inherits from Person, Library

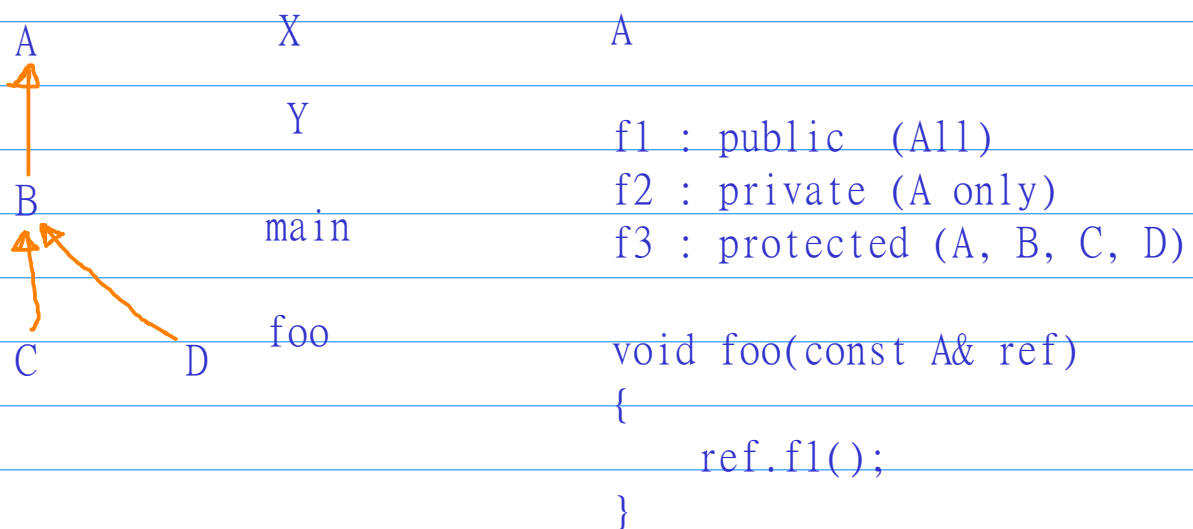
=> Sports/Library instead of base class, make it
as feature (is-a => has-a/part-of)

Assignment:-

- * Person, Student
- * Person, Employee
- * With UML Diagrams, C++ code
- * Order of executing constructors, destructors in
 - Single, multilevel, multiple
- * Protected access (self-read)

Further:-

- Overriding (Specialization)
- Upcasting and Downcasting
- Call binding - static/dynamic
- Virtual Functions (Dynamic/Late Binding)
- Runtime Polymorphism
- Pure virtual function
- Abstract class



Overloading

same name, diff params, same scope

A -- f1, f2, f3

Overriding

same name, same params (including qualifiers)
defined in both base & derived

B -- f2, f3

C -- f3

Class Base

```

int sum(int,int);
int sum(float,float);
    
```

```

c1.f3(); //1
c1.f2(); //2
c1.f1(); //3
    
```

-- overloading
-- overloading
-- overriding
-- nothing

Class Derived

```

int sum(int,int,int); //3
int sum(float,float); //4
    
```

```
Box *pb = &b1;
Box &rb = b1;
```

Identify functions for which overriding applicable

- * Shape and Circle, Polygon
- * Polygon and Rectangle, Triangle -- area, circumference
- * Account and Saving, Current, Credit account
 - debit/withdraw, credit/deposit
- * Customer and Prepaid (balance), Postpaid (outstanding)
 - credit (billPay/recharge), makeCall
- * Employee and Engineer, Manager, Faculty, Trainee
 - payroll, appraisal
- * Vehicle -- Bus, Car (start, stop, break, accel, decel)
- * Car -- Mini, Sedan, SUV

- * Person and Student
- * Person and Employee

- * Image and PNG, JPEG images
 - crop, scale, skew, rotate

- * Sensors and Proximity, Temperature, IMU/Motion
- * MotionSensor and Accelerometer, Gyroscope, Magnetometer

Upcasting & Downcasting

	Objects	Pointers	Reference
Shape	s1	ps	sref
Circle	c1	pc	cref
Triangle	t1	pt	tref
Rectangle	rect	prect	rectref
Shape& rs = s1; Shape *ps = &s1;		Shape& rs = c1; Shape* ps = c1;	Upcasting OK, Allowed
Circle& rc = c1; Circle* pc = &c1;		Circle& rc = s1; Circle *pc = &s1;	Down casting Not allowed Error

A is base class

B is derived from A

A a1;

B b1;

```
A &r1 = a1; //no casting
B &r2 = b1; //no casting
A &r3 = b1; //up, OK
B &r4 = a1; //down, error
A &r5 = r2;
```

A *p1, *p2, *pa;

B *p3, *p4, *pa;

p1 = &a1; //OK, no casting

p3 = &b1; //OK, no casting

p2 = &b1; //Upcasting, allowed

p4 = &a1; //Downcasting, not allowed

```
void Compute(A *pa);
void Calculate(A& ra);
Compute(&a1);
Compute(&b1);
Calculate(a1);
Calculate(b1);
```

A *pa = new B(); //Allowed , up

B *pb = new A(); //Not allowed, down

```
void Demo(B *pa);
void Test(B& rb);
```

```
demo(&a1); //not allowed
test(a1); //not allowed
demo(&b1); //ok
test(b1); //ok
```

Function binding rules in C++

- In case of normal functions (non-virtual), binding depends on type of pointer/reference...but not on type of actual objects being pointed/referenced
- In case of virtual functions, binding depends on type of actual object (being pointed/referenced) but not on type of pointer/reference

Non Virtual Functions -- Static/Early binding

Virtual Functions -- dynamic/late binding -->
runtime polymorphism

class A	A a1;	
{	B b1;	
public:		
virtual void f1();	A* p1 = &a1;	A& r1 = a1;
virtual void f2();	p1->f1();	r1.f1();
void test();	p1->f2();	r1.f2();
};	p1->test();	r1.test();
class B : public A		
{	A *p2 = &b1;	A& r2 = b2;
public:	p2->f1();	r2.f1();
void f1();	p2->f2();	r2.f2();
void f3();	p2->test();	r2.test();
void test();		
};	p2->f3();	//error

Shape -- area, circumference	} as virtual
Circle -- area, circumference	
Triangle -- area, circumference	
Rectangle -- area, circumference	

```
Circle c1(7);
Triangle t1(3,4,5);
Rectangle rect(10,20);
```

Shape *p1, *p2, *p3;	Shape *ptr;
p1 = &c1;	if(cond) // ch==1
p2 = &t1;	ptr = &c1;
p3 = ▭	else
	ptr = &t1;

p1->area();	ptr->area();
p2->area();	ptr->circumference();
p3->area();	--

Dynamic binding/Late binding
Runtime Polymorphism

