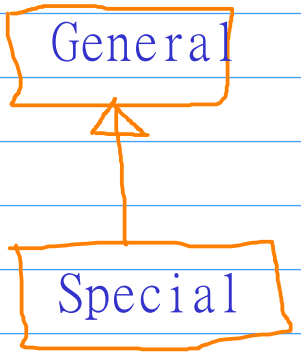
Inheritance

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

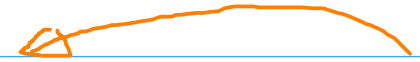
Checkpoint:-
* What is Inheritance
* What is not Inheritance / Bad Inheritance
  (When there is a scope of composition/aggregation,
   When two objects just differ by data attribute)

learncpp : 23.1, 23.2, 23.3, 23.4

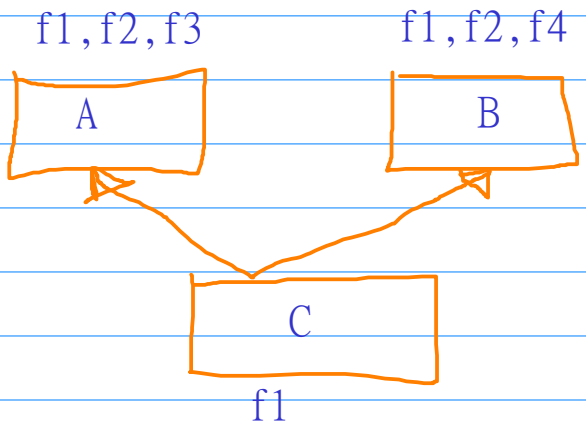
```
@startuml
class Person
{

}
class Student
{

}
Student -up-> Person
@enduml
```

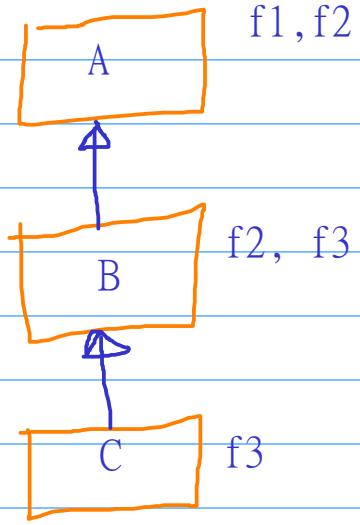
Types of Inheritance
- Single
- Multilevel
- Multiple

- Hierarchial
- Hybrid

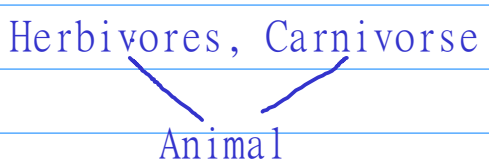


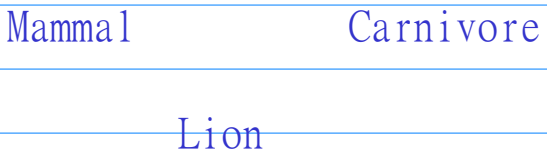
Person	Student
- Name	- rollno
- UID/SSN	- score
- Gender	
- DOB	
+ isMajor	+ grade()
+ isEligible	+ isPass()
+ computeAge	





Calling f2 from C -- ambiguous





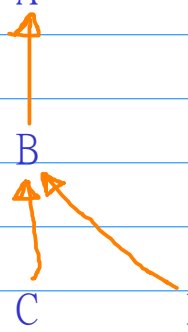
Student inherits from Person, Sports
Student inherits from Person, Library
==> Sports/Library instead of base class, make it as feature (is-a ==> has-a/part-of)

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



X

Y

main

foo

```
A

f1 : public  (All)
f2 : private (A only)
f3 : protected (A, B, C, D)

void foo(const A& ref)
{
    ref.f1();
}
```

Overloading
same name, diff params, same scope

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

```
A -- f1, f2, f3

B -- f2, f3

C -- f3

c1.f3();
c1.f2();
```

```
Class Base
    int sum(int,int);          //1
    int sum(float,float);      //2

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

1 & 2 -- overloading
3 & 4 -- overloading
2 & 4 -- overriding
1 & 3 -- nothing

```
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 Accelorometer, 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& rs = c1;
Shape *ps = &s1;         Shape* ps = c1;
```

Upcasting OK, Allowed

```
Circle& rc = c1;         Circle& rc = s1;
Circle* pc = &c1;        Circle *pc = &s1;
```

Down casting Not allowed Error

```
A is base class
B is derived from A

A a1;
B b1;

A *p1, *p2, *pa;
B *p3, *p4, *pa;

p1 = &a1; //OK, no casting
p3 = &b1; //OK, no castng
p2 = &b1; //Upcasting, allowed
p4 = &a1; //Downcating, not allowed

A *pa = new B(); //Allowed , up
B *pb = new A(); //Not allowed, down
```

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

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

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
{
   public:
   virtual void f1();
   virtual void f2();
   void test();
};
class B : public A
{
   public:
   void f1();
   void f3();
   void test();
};
```

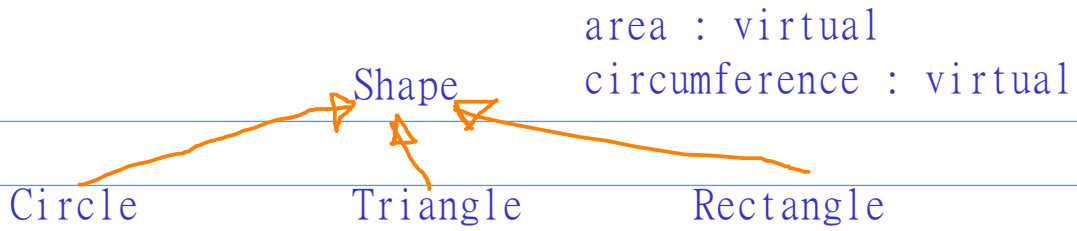
```
A a1;
B b1;

A* p1 = &a1;     A& r1 = a1;
p1->f1();        r1.f1();
p1->f2();        r1.f2();
p1->test();      r1.test();

A *p2 = &b1;     A& r2 = b2;
p2->f1();        r2.f1();
p2->f2();        r2.f2();
p2->test();      r2.test();

p2->f3();        //error
```

Shape -- area, circumference as virtual
Circle -- area, circumference
Trianle -- area, circumference
Rectangle -- area, circumference

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

Shape *p1, *p2, *p3;
p1 = &c1;
p2 = &t1;
p3 = &rect;

p1->area();
p2->area();
p3->area();
```

```
Shape *ptr;
if(cond) // ch==1
    ptr = &c1;
else
    ptr = &t1;

ptr->area();
ptr->circumference();
```

--
Dynamic binding/Late binding
Runtime Polymorphism




```
void compute(Shape* ptr)
{
    //ptr->area()
    //ptr->circumference()
}

compute(&c1);
compute(&t1);
compute(&rect);

c1.area()
c1.circumference()
```

upcasting

```
void calculate(Shape& ref)
{
        //ref.area()
        //ref.circumference()
}
calculate(c1)
calculate(t1)
calculate(rect)
```

Summary
- Inheritance, Types of Inheritance
- Constructor syntax (multilevel, multiple)
- Overriding
- Upcasting
- Binding in case of normal functions
- Binding in case of virtual functions
- Polymorphism using virtual functions
- Pure Virtual Functions, Abstract classes
- Need for abstract classes, limitations

protected

General Guideline:-
* Specialization : Override, make it as virtual
* No Specialization : Keep it Base only (re-use)

Re-use vs Specialization

Pure virtual functions

```
class IShape
{
    public:
    virtual double area()=0;
    virtual double circumference()=0;
};
```

```
class Shape
{
    public:
    virtual double area();
    virtual double circumference();
};
```

IShape considered as abstract keyword (one or more pure virt)
Objects can't be instantiated from abstract class

Derived classes will implement pure virt functions, who can create objects..and considered non-abstract/concrete class

```
void compute(IShape* ptr)
{
    //ptr->area()
    //ptr->circumference()
}
```

Clases like Polygon, inheriting abstract class like IShape but not implmenting pure virt functions
--> also considered as abstract class
--> can't create object

Practice
- IShape, Polygon, Circle, Rectangle, Triangle
- Banking Scenario
  : IAccount
  : SavingsAccount, CreditCardAccount (credit, debit)
- Mobile Billing Scenario
  : Customer, getBalance
  : PrepaidCustomer, PostpaidCustomer (credit, makeCall

Further Topics:-
* Virtual Tables (Internal Mchanism)
* Virtual Destructor
* RTTI : typeid, dynamic_cast, static_cast
* Multiple Inheritance, Diamond Inheritance