Class/Object Relationships

CS(217) Object Oriented Programming

Identifiers and Downcasting

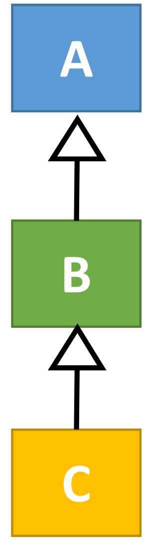
Inheritance (is-a) Identifier **override**

- A derived class can override, inherited virtual functions.
  - but the return type, name and parameters should same.
- If by mistake the programmer change return type, name or parameters the program may generate logical errors.
  - To avoid this issue the identifier **override** is added at end of virtual overridden function header.
  - Compiler will generate an error message, if function is not properly overridden in derived class.
- Programmer can visualize the overridden virtual functions directly by looking at derived class implementation.

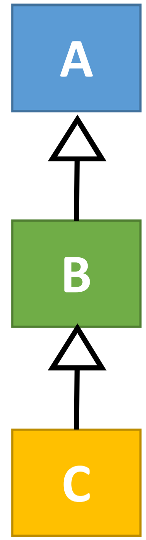
Inheritance (is-a) Identifier **override**

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
// Compile Time Error: change return type
    int print() override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



Inheritance (is-a) Identifier **override**



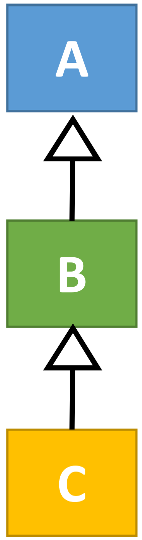

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
// Compile Time Error: Not override
    void print(int x) override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```

Inheritance (is-a) Identifier **override**

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
// Compile Time Error: Not override
    void print() const override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



Inheritance (is-a) Identifier **final**

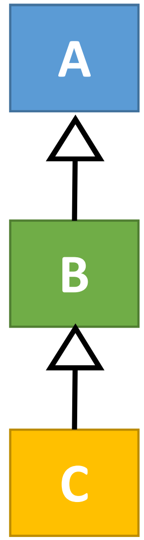
- We can stop a derive class to override an inherited function.
  - Add final keyword at end of the function header.
  - Compiler will generate an error and will not allow to override a final function.

- We can stop inheritance of a class.
  - Define the class as final
  - Compiler will generate an error and will not allow to derive a class from final class.

Inheritance (is-a) Identifier **final** function

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override final{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

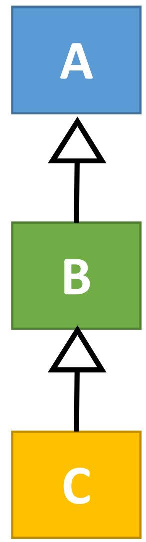
```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
// Compile Time Error: Cannot override print
function inherited from class B as declared
final in class B
    void print(){
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



Inheritance (is-a) Identifier **final** Class

```
class A final{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){
        cout<<a;
    }
    virtual ~A(){}
};
```

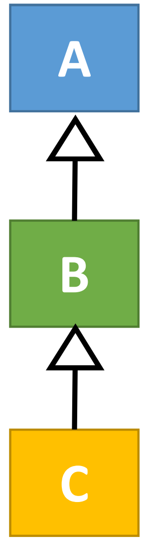
```
// Compile Time Error: Cannot
derive from final class A
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override {
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```



Inheritance (is-a) Identifier **final** Class

```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B final : public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

```
// Compile Time Error: Cannot derive from
final class B

class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
void print(){
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



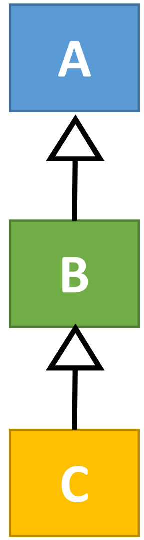
Inheritance (is-a) **Down casting Pointers**

- Down casting converts base class pointer to derived class pointer,
  - if base class is pointing to derived class object.
  - `dynamic_cast` operator is used for down casting pointers
    - Determine object's type at runtime
    - Returns 0 or Null, if not of proper type (cannot be cast)
  - `dynamic_cast` will not work
    - With protected and private inheritance
    - With classes, which not have any virtual functions.
- Down casting is helpful
  - For accessing explicitly derived class data and functions that does not exist in base class.

Inheritance (is-a) **Downcasting Pointers**

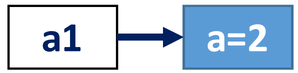
```
class A{
    int a;
public:
    A(int a=0){ this->a=a;}
    virtual void print(){ cout<<a;}
    virtual ~A(){}
};
class B: public A{
    int b;
public:
    B(int a=0, int b=0):A(a)
    { this->b = b;}
    void print() override{
        A::print();
        cout<<b;
    }
    virtual ~B(){}
};
```

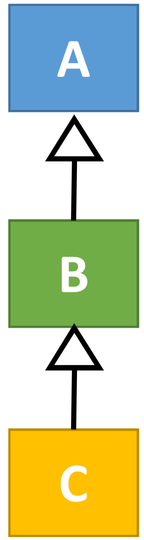
```
class C: public B{
    int c;
public:
    C(int a=0, int b=0, int c=0) :B(a,b)
    { this->c = c;}
    void print() override{
        B::print();
        cout<<c;
    }
    virtual ~C(){}
};
```



Inheritance (is-a) Downcasting Pointers

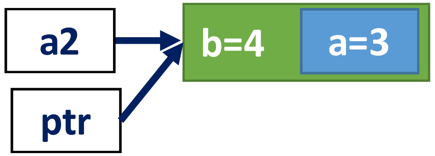
```
void main(){
   A * a1 = new A(2); //A’s pointer to A’s object
   a1->print(); //A’s print called.

   B *ptr = dynamic_cast<B*>(a1);
      if (ptr != NULL)  //return null when failed
       ptr->print();

   // Type Casting failed as A’s pointer is pointing
   to A’s object
   // Through Null check we can avoid run time error

}
```

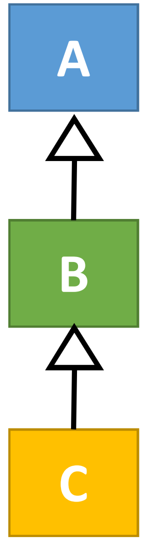




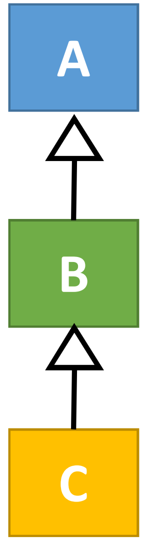
Inheritance (is-a) **Downcasting Pointers**

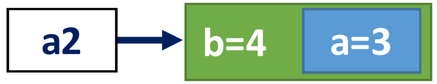
```
void main(){
    A * a2 = new B(3, 4); //A's pointer to B's object
    a2->print(); //B's print called.

    B *ptr = dynamic_cast<B*>(a2);
       if (ptr != NULL)  //return null when failed
        ptr->print();

    // Type Casting is successful because A's pointer
    is pointing to B's object
    // Not create new object just perform down casting
    of same object for derived class pointer

}
```





Inheritance (is-a) **Downcasting Pointers**

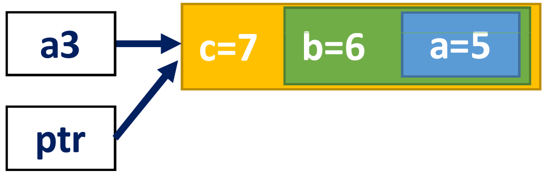


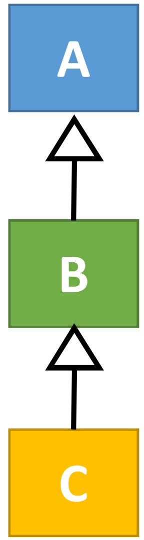



```
void main(){
   A * a2 = new B(3, 4); //A’s pointer to B’s object
   a2->print(); //B’s print called.

   C *ptr = dynamic_cast<C*>(a2);
      if (ptr != NULL)  //return null when failed
       ptr->print();

   // Type Casting failed as A’s pointer is pointing to
   B’s object
   // Through Null check we can avoid run time error


}
```

Inheritance (is-a) **Downcasting Pointers**

```
void main(){
    A * a3 = new C(5, 6, 7); //A's pointer to C's object
    a3->print(); //C's print called.

    C *ptr = dynamic_cast<C*>(a3);
        if (ptr != NULL)  //return null when failed
         ptr->print();

    // Type Casting is successful because A's pointer is
    pointing to C's object
    // Not create new object just perform down casting of
    same object for derived class pointer

}
```



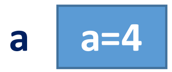


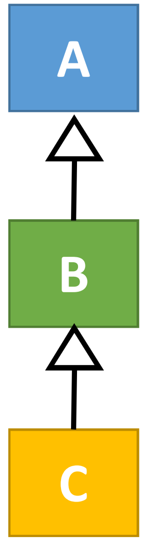
Inheritance (is-a) **Down casting References**

- Down casting converts base class reference to derived class object,
  - if base class is pointing to derived class object.
  - **`dynamic_cast`** operator is used for down casting References
    - Determine object's type at runtime
    - No way to check, if not of proper type (cannot be cast)
    - Exception is generated by system for bad cast error.
  - **`dynamic_cast`** will not work
    - With protected and private inheritance
    - With classes, which not have any virtual functions.
- Down casting is helpful
  - For accessing explicitly derived class data and functions that does not exist in base class.

Inheritance (is-a) **Down casting References**

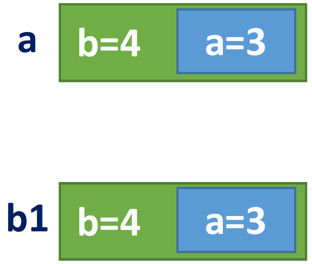
```
void main(){
    A & a = A(4);
    a.print(); //A's print called.

    try{
        B b1 = dynamic_cast<B &> (a);
        b1.print();
    }
        catch (bad_cast e){ //throws bad cast error.
        cout << e.what()<<endl;
    }
    // Type Casting failed as A's reference is to A's object
    // Bad Cast Error is generated by system

}
```

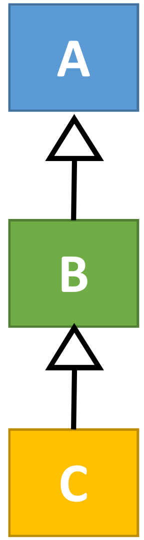




Inheritance (is-a) **Down casting References**

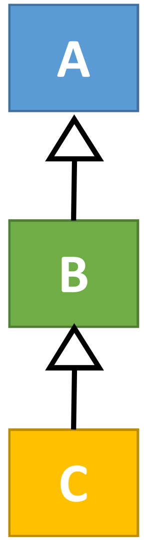
```
void main(){
    A & a = B(3, 4);
    a.print(); //B’s print called.

    try{
        B b1 = dynamic_cast<B &> (a);
        b1.print();
    }
        catch (bad_cast e){ //throws bad cast error.
        cout << e.what()<<endl;
    }
    // Type Casting is successful because A’s reference to
    B’s object
    // Create new object by calling copy constructor
}
```





Inheritance (is-a) **Down casting References**

```
void main(){
    A & a = B(3, 4);
    a.print(); //B's print called.

    try{
        C c1 = dynamic_cast<C &> (a);
        c1.print();
    }
        catch (bad_cast e){ //throws bad cast error.
        cout << e.what()<<endl;
    }
    // Type Casting failed as A's reference to B's object
    // Bad Cast Error is generated by system
}
```



Inheritance (is-a) Down casting References

```
void main(){
    A & a = C(5,6,7);
    a.print(); //B's print called.

    try{
        C c1 = dynamic_cast<C &> (a);
        c1.print();
    }
        catch (bad_cast e){ //throws bad cast error.
        cout << e.what()<<endl;
    }
    // Type Casting is successful because A's reference to
    C's object
    // Create new object by calling copy constructor
}
```