(121) Shape Hierarchy

```
class shapes                    abstract class
{   public:
    virtual void draw() = 0; // pure virtual fn
};
class polygon: public shapes
{   public:
    void draw() { cout << "polygon" << endl; }
};
class closedconic: public shapes // Abstract
{
};
class Triangle: public polygon
{   public:
    void draw()
    { cout << "Triangle" << endl;
    }
};
class Quadrilateral: public polygon
{   public:
    void draw()
    { cout << "Quadrilateral" << endl;
    }
};
class Circle: public closedConic
{   public:
    void draw()
    {   cout << "Circle" << endl;
    }
};
```

```
class Ellipse : public closed conics.
{
    public:
    void draw()
    { cout<<"Ellipse"<<endl;
    }
};
Shapes *arr[] = { new Triangle, new
    Quadrilateral, new Circle, new Ellipse};
for(int i=0; i< sizeof(arr) / sizeof(
    Shapes *); ++i)
    arr[i]->draw();
```

O/p. Triangle
Quadrilateral
Circle.
Ellipse.

* A pure virtual may have a body. It we provide the implementation and its purity due not gua. generally it is used to contain some common code.

```
class Shapes
{   public:
    virtual void draw()=0
    { cout<<"Init"<<endl;
    }
};
```

(203)
```
class polygon : public shapes
{
    void draw()
    {
        shapes :: draw();
        cout << "polygon" << endl;
    }
};
class closedcurve : public shapes
{
};
class Triangle : public polygon
{
        public:
        void draw()
        {
            shapes :: draw();
            cout << "Triangle" << endl;
        }
};
class Quadrilateral : public polygon
{
        public:
        void draw()
        {
            shape :: draw()
            cout << "Quadrilateral" << endl;
        }
};
class circle : public closedcurve
{
        public:
```

```
void draw()
{
    Shapes::draw();
    cout<<"Circle"<<endl;
}
```

Q:

```
class Ellipse: public ClosedCurves
{
    public:
    void draw()
    {
        Shapes::draw();
        cout<<"Ellipse"<<endl;
    }
};
```

O/P:

Init
Triangle
Init
Quadrilateral
Init
Circle
Init
Ellipse.

Exercise

```
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A*) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B*) { }
};
class C: public B { public:
    void g(double) { }
    int h(B*) { }
};
```


```
//Application code
A a;
B b;
C c;
```

```
A *pA
B *pB
```

```
A* pA = &a;
pA = &b;
pA = &c;
```

initialization.

invocation	pA = &a;	pA = &b	pA = &c
pA->f(3)	A::f	B::f	~~B::f~~ B::f
pA->g(3.2)	A::g	A::g	C::g
pA->h(&a)	A::h	A::h	A::h
pA->h(&b)	A::h	A::h	A::h

Initialization

Invocation	PB = &a	PB = &b	PB = &c
PB → f(2)	Error Down conversion (A*) to (B*)	B::f	B::f
PB → g(3.2)		A::g	C::g
PB → h(&a)	"	No conversion from (A*) to (B*)	No conversion from (A*) to (B*)
PB → h(&b)	"	B::h	C::h

* polymorphic Dispatcher Dynamic Binding

* Calling a virtual function using a pointer need dynamic binding

Virtual function pointer table:

Base class

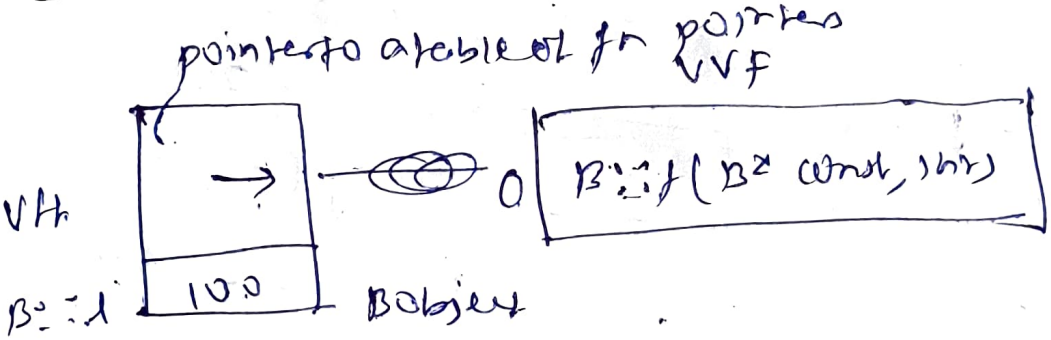
```
class B
{   int i;
    public:
      B(int a): i(a) {}
        void f(int); // B::f(B*const, int)
    virtual void g(int); // B::g(B*const, int)
};
B b(100)
B *p = &b
```

Derived class

```
class D: public B {
    int j;
public:
    D(int a, int b): B(a), j(b) { }
    void f(int); // D::f(D* const, int)
    void g(int); // D::g(D* const, int)
};

D d(200, 500);
B *p = &d
```

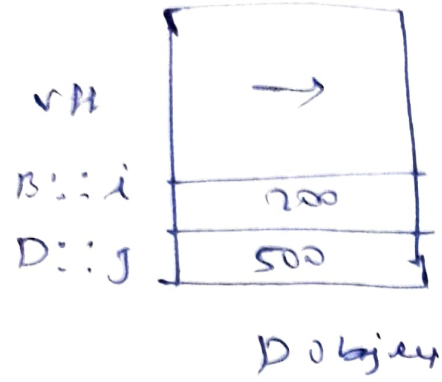
~~source exp~~ For Base class



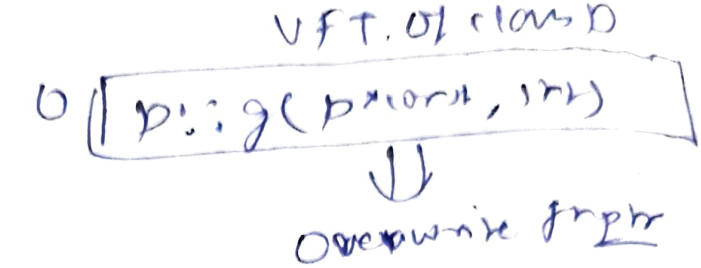
source exp	Compiled exp
b.f(15);	B::f(&b, 15);
p->f(25);	B::f(p, 25)
b.g(35);	B::g(&b, 35)
p->g(45);	p->VPtr[0](p, 45)

D d (200, 500)

B *p = &d;



D Object



Source exp	Compiled exp
d.f(15);	D::f(&d, 15);
p->f(25);	D::f(p, 25)
d.g(35);	D::g(&d, 35)
p->g(45);	p->VH[0](p, 45);

* Compiler knowing that the fn is virtual and it is called using pointer it does not generate the hardcode / static time code But it puts the call as it through VFT.

* If a class is polymorphic or type is polymorphic i.e at it has atleast one virtual fn then for that class there will be a virtual fn table which will have all the

(20) Virtual fn listed one after the other in the table in the order in which they have been declared as the class is specified. When the compiler creates the derived class it redefine the fn, if it does then it changes the corresponding entries in its own VFT virtual fn table and replaces it with overriden

Note:

(i) When a class declares a virtual fn. a hidden member variable is added to the class which points to an array of pointers to (virtual) fn called the virtual fn table (VFT)

(ii) VFT pointers are used at run time to invoke the appropriate fn implementations because at compile time it may not yet be known if the base fn is to be called or derived one implemented by class that inherits from base class

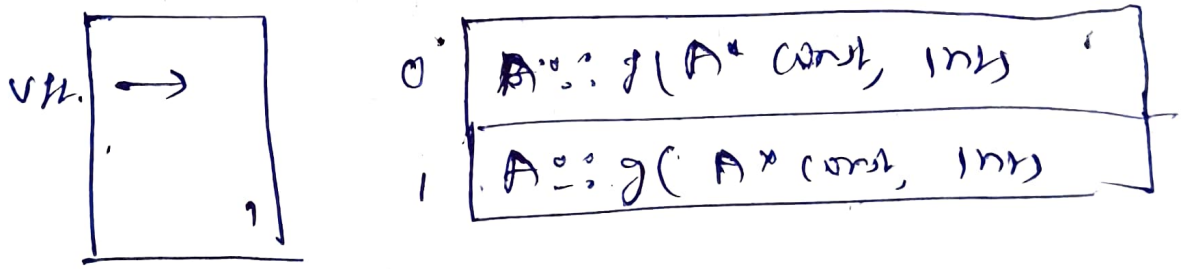
(iii) VFT is class specific - all instances of the class have the same VFT.

(iv) VFT contains the RTTI (Run time type information) of objects

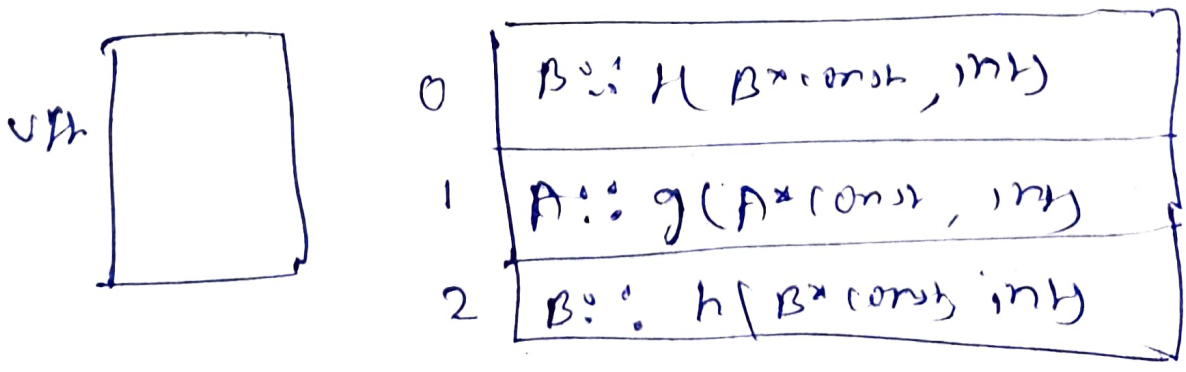
Exp. 2

```
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A*) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B*) { }
};
class C: public B { public:
    void g(double) { }
    int h(B*) { }
};
```

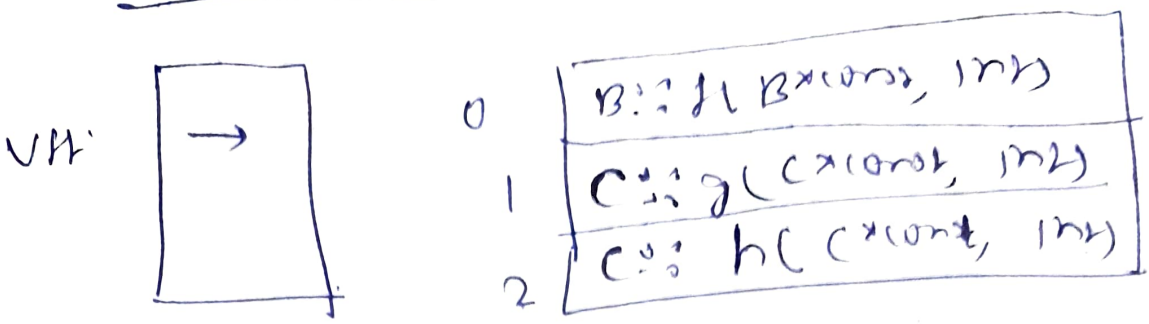
a Object layout



b object layout



C. Object layout



A a; B a; C c;

A *pA; B *pB;

source exp	compiled exp
pA->f(2)	pA->VH[0](pA, 2);
pA->g(3.2)	pA->VH[1](pA, 3.2);
pA->h(&a)	A::h(pA, &a);
pA->h(&b)	A::h(pA, &b);
pB->f(2)	pB->VH[0](pB, 2);
pB->g(3.2)	pB->VH[1](pB, 3.2);
pB->h(&a);	pB->VH[2](pB, &a);
pB->h(&b)	pB->VH[2](pB, &b);

Type casting

Why casting

Casts are used to convert the type of an object, expression, function arguments or return value to that of another type.

(i) Implicit conversion

The standard C++ conversion and user defined conversion

Explicit conversion

Type is needed for an expression that cannot be obtained through an implicit conversion or for one standard conversion creates an ambiguous situation

To perform a type cast the compiler may do the following

(a) allocates temporary storage

(b) initializes temporary with value being cast.

```
double f(int i, int j) {return
    (double) i / j;}
```

while casting the resultant value may have different representation so may be stored in temporary location

(213) compiler generate

```
double f( int i, int j)
{
    double temp_i = i;
    double temp_j = j;   // Conversion
    return temp_i / temp_j;
}
```

Casting in C and C++

Casting in C

* implicit cast
* explicit "
* loses type information in several cases
* lacks clarity of semantics

Casting in C++

* performs fresh inference of types without change of value.
* perform fresh inference of types with change of value

Both can be done

(i) using implicit computation

(ii) using explicit (user defined) computation

* preserves type info in all contexts
* provides clear semantics through cast operators.

(i) const_cast
(ii) static_cast
(iii) reinterpret_cast
(iv) dynamic_cast

* cast operator can be grepe in source
* C style cast must be avoided in C++

Cast operators

A cast operator take an exp. of source type (implicit from the exp) and convert it to an exp. of target type (explicit in the operator) following the semantics of the operator

```
i: T1   ⟶   T2
sourcetype   target type
```

```
Cast<T2>(i)
    ↑
  target
```

* Use of cast operator increases robustness by generating errors in static or dynamic time

(115) const_cast Operator

* const_cast Operator converts between types with different cv (const-volatile) qualification.

* only const_cast maybe used to cast away (remove) const-ness or volatility

* usually does not perform computation or change value

Ex.

```
class A {int i;
    public: A(int a): i(a) {}
    int get() const { return i; }
    void set(int j) { i = j; }
};
void print(char *str) { cout << str; }

main()
    const char *c = "sample text";
    print(c)  // compiler error
    print( const_cast<char*>(c))
                ↑
        It will remove the const
        ness from c

const A a(1);
a.get()
a.set(5) // error
```

const_cast<A&>(a).set(5)

creating a ref to a which is non const

is const object

exp has this pointer which does not point to a const object

const_cast<A&>(a).set(5) //error

x we can not strip the constness of whole object bcoz it mean a completely diff object

exp.2 const_cast versus C-style cast

```
int main()
{
    const char *c = "sample text";
    print( const_cast<char*>(c));
    print((char*)(c));

    const A a(1);
    const_cast<A&>(a).set(5);
    ((A&)a).set(5)
```

(17) const_cast<A>(a).set(5) // error

((A)a).set(5) ✓ ok in ~~type~~ C++

Ex-3

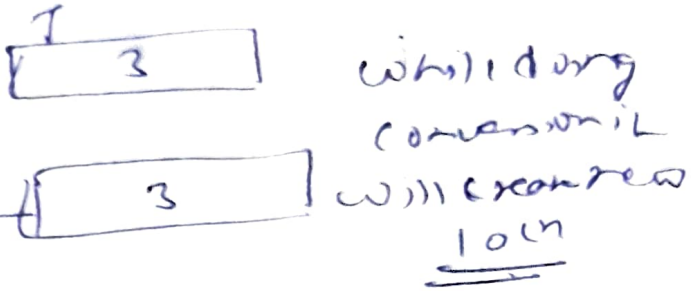
```
struct type { type() : i(3) { }

    void m1(int v) const {
        this->i=v // Not allowed
        const_cast<type*>(this)->i=v // allowed
    }
    int i;
};
int i=3
const int& cref_i = i
const_cast<int&>(cref_i)=4 // ok
cout << "i = " << i << endl;
type t
for to m1(4):
    cout << "type::i = " << f.i << endl;
```

Undefined behaviour value of j and pj may differ

```
const int j = 3
int *pj = const_cast<int*>(&j);
*pj=4
cout << j << " " << *pj << endl;
```

box2



while doing conversion it will create new loc

```
void (type::*mfp)(int) const = &type::m1;
const_cast<void (type::*)(int)>(mfp);
```

Note: constness cannot be removed from function pointer

O/p: i = 4
type::i = 4
3, 4

Static_cast operators

→ static_cast perform all conversion which are allowed implicitly (not only those with pointers to classes) and also the opposite of these. It can

(a) convert from void* to any pointer type

(b) convert integer, floating point values and enum types to enum types

(19) * static_cast can perform conversion between pointers to related classes.

(a) Not only upcast, but it can also do downcast

(b) No checkery are perform during runtime to guarantee that the object being converted is infact a full object of the desire type.

* Additionally, static_cast can also perform the following

* explicitly call a single argument constructor or a conversion operator. i.e cast through user defined conv.

* convert to rvalue references

* convert enum class values into integer or floating point values.

* convert any type to void, evaluating and discarding the value.

Ex.1 static_cast operator built in types

```
int main()
{   int i = 2;
    double d = 3.7;
    double *pd = &d;
    i = d // warning (implicit)
    i = static_cast<int>(d);
```

we are telling that we are losing information.

```
d = i
d = static_cast<double>(i)
d = (double) i

i = pd // implicit error

j = static_cast<int>(pd) // static cast error

i = (int) pd ; // ~~style~~ C-style OK.
                Risky we should use
reinterpret_cast.
```

static_cast operator class hierarchy

```
class A { };
class B : public A { };

A a
B b
A* p = &b
B *q;

p = &b // implicit OK.

p = static_cast<A*>(&b) // static_cast OK

p = (A*)(&b) // C style cast OK
```

Downcast

(21) `q = &a;` // implicit error

`q = static_cast<B*>(&a);` // static cast OK.

Risky we should use dynamic cast.

`q = (B*)(&a);` // OK - C-style cast

static_cast operator pitfalls

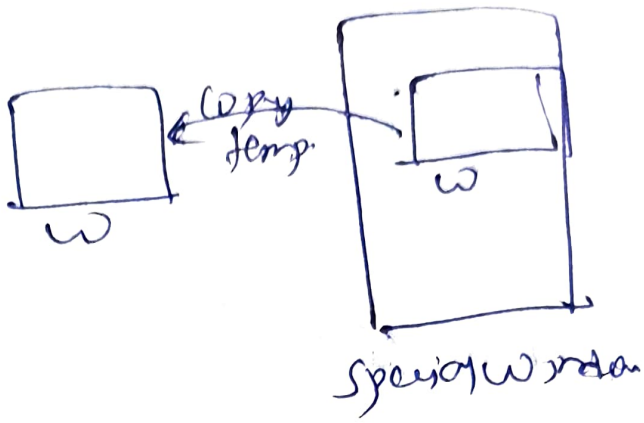
```
class Window { public: virtual void
        onResize(); ... };
class specialWindow: public Window.
{ public:
    virtual void onResize() {
        static_cast<Window>(*this).
                onResize();
        // base class cast *this to
        window and base calls
        its Resize this doesn't
        work
        - // do specialWindow specific
        stuff.
    }
};
```

↳ This copies the whole Object.

It leads to having a new window Object from current special window Object

'window w(x/y/w/h)

this slices the object, creates a temporary window from base part of special window and calls the method



soln

class specialWindow : public Window {

public:

virtual void onResize() {

Window::onResize();

// do window specific work.

}

};

Here this pointer of special window gets implicit converted to window

(223) static_cast operator unrelated classes

```
class B;
class A {
    public:
};
class B{};
int main()
{   A a;
    B b;
    int i=5;
```

we have an object of type B and we need to use it to create an object of type A.

(B) ⇒ A

```
    a=b    //error
    a= static_cast<A>(b) //error
    a=(A) b; //error
    //int ⇒ A
    a=i //error
    a= static_cast<A>(i) // error
    a= (A) i //error
```

This can be done using user defined conversion in the form of constructor

* provide a constructor of target type which takes a source type. for

class B;

class A{ public:

A(int i=0)
{ cout<<"A::A(i)"; }

A(const B &)
{ cout<<"A::B(&)"; }

};

class B{};

A a
B b
int i=5
a = b // uses A::B(&)
a = static_cast<A>(b) uses A::B(&)
a = (A) b // uses A::B(&)

b = b1 (type of A)
A = b1

- It takes a B object and create b1 of object of type A and assign it to a copy free copy assignment operator

a = i // uses A::A(i)
a = static_cast<A>(i) // uses A::A(i)
a = (A) i // uses A::A(i)

(225) User defined conversion provide an appropriate type of ~~Conversion~~ Constructor for in target type for source type

Ex. 2

```
class B;
class A{ int i; public:
};
class B{ public:
};
A a, B a; int i=5;

//B ⇒ A
A a=b; //error
a= static_cast<A>(b); //error
a= (A)b //error

// A ⇒ int
i=a // error
i= static_cast<int>(a) //error
i= (int) a //error
```

⇓

Can be solved using ~~cast~~