

(20)

## Shape Memory

class shapes

abstract for

↳ public:

↳ virtual void draw() = 0; // pure  
final in

{;

class polygon: public shapes

↳ public:

↳ void draw() & out<< "polygon" << endl;

{;

class closedconic: public shapes // Abstr.

{

{;

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 class (202)

```
{  
    public:  
        void draw();  
    }  
}
```

Shapes \*arr[] = { new Triangle, new  
Quadrilateral, new Circle, new Ellipse};

for (int i=0; i< sizeof(arr) / sizeof(

```
shapes *); ++i)  
    arr[i] → draw();
```

Obj. Triangle

Quadrilateral

Circle

Ellipse.

\* A pure virtual may have a body. If we provide the implementation ourselves purely due to class generality it is used to contain some common code.

class Shapes:

↳ public:

```
    virtual void draw() = 0  
    { cout << "Draw" << endl;  
    }
```

(203) class polygon: public shapes

↳ void draw()

↳ {  
    shape::draw();  
    cout << "polygon" << endl;  
}

};  
class closedFigure: public shapes

↳

};

class Triangle: public polygon

↳

public:

void draw()

↳ {  
    shape::draw();  
}

    cout << "Triangle" << endl;

}

};

class Quadrilateral: public polygon

↳

public:

void draw()

↳ {  
    shape::draw();  
}

    cout << "Quadrilateral" << endl;

}

};

class circle: public closedFigure

↳

public:

void draw()

{  
  shapes:: draw();  
  cout << "Circle" << endl;

(204)

2) class Ellipse: public closedComics

{  
  public:

    void draw()

{  
  shapes:: draw();  
  cout << "Ellipse" << endl;

}

Oly:

Drift

Triangle

Drift

Quadrilateral

Drift

Circle

Drift

Ellipse.

(205)

Exercise

e class A public:

A 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;      A \*pa

B b;      B \*pb

C c;


 A\*: 1  
 B\*: 1

Initialization.

invocation	PA = &a;	PA = &b	PA = &c
PA → f()	A::f	B::f	<del>B::f</del> C::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

206

Initialization

Invocation	$PB = 8a$	$PB = 8b$	$PB = 8c$
$PB \rightarrow f(2)$	Error Down cast $(A^*)$ to $(B^*)$	$B::f$	$B::f$
$PB \rightarrow g(3, 2)$		$A::g$	$C::g$
$PB \rightarrow h(8a)$	"	No conversion from $(A^*)$ to $(B^*)$	No conversion from $(A^*)$ to $(B^*)$
$PB \rightarrow h(8b)$	"	$B::h$	$C::h$

↗ polymorphic dispatch or dynamic binding

↙ calling a virtual function or pointer needs  
dynamic binding ↘

Virtual function pointer variable:  
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)

 $B * p = 8b$

(207)

### Derived class

class D: public A

int j;

public:

D(m1, m2): B(a), j(b) {}

void f(int); // D::f(B const, int)

void g(int); // D::g(B const, int)

By

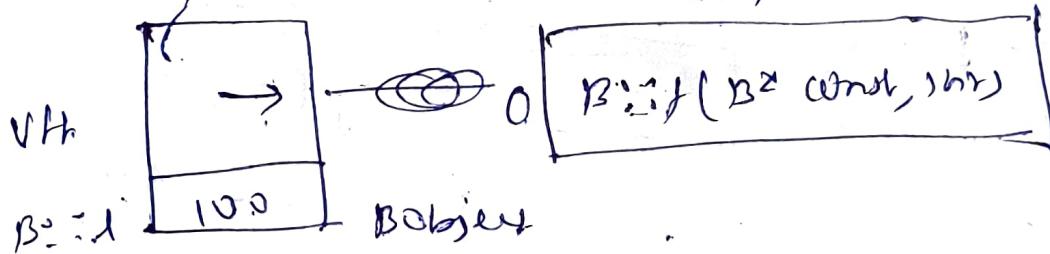
D d(200, 500);

$$B^* p = \&d$$

some exp

For Base class

pointers available for pointers



some exp

Compiled exp

b.f(15); ————— B::f(8b, 15);

p->f(25); ————— B::f(~~p~~, 25)

b.g(35); ————— B::g(8b, 35)

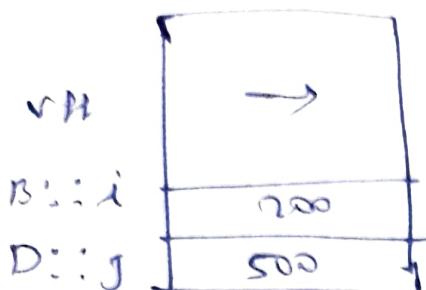
p->g(45); ————— p->VH[0] (p, 45)

for Derived class

(208)

D d (200, 500)

B \* p = 2d



D objects

VFT of class D  
D [ D::g (D::x, 17) ]  
↓  
Overwrite fnptr

source exp

d.f(15);

p->f(25);

d.g(35);

p->g(45);

compiled exp

D:: f(2d, 15);

D:: f(p, 25)

D:: g(2d, 35)

p->vft[0] (p, 45);

\* Compiler knowing that fn is virtual and it is called using pointers it does not generate the hardware / static link code. But it puts the call as it through VFT.

\* If a class is polymorphic or type is polymorphic i.e. if it has atleast one virtual function 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 defined on the class is specialized for the compiler creates derived class redefinition for if it does then it changes the corresponding entries in its own VFT virtual fn table and replaces it with overridden.

Note:

- ① When a class defines 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)
- ② VFT pointers are used at run time to invoke the appropriate fn implementation 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.
- ③ VFT is class specific - all instances of the class has the same VFT.
- ④ VFT contains RTTL (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 &amp; public:

void f(int) {}

virtual int h(B\*) {}

{};

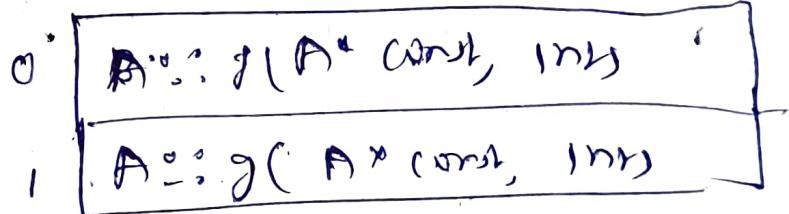
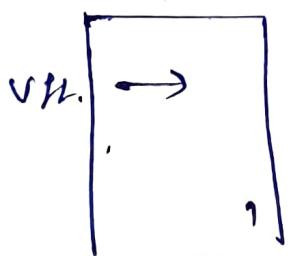
Class C: public B &amp; public:

void g(double) {}

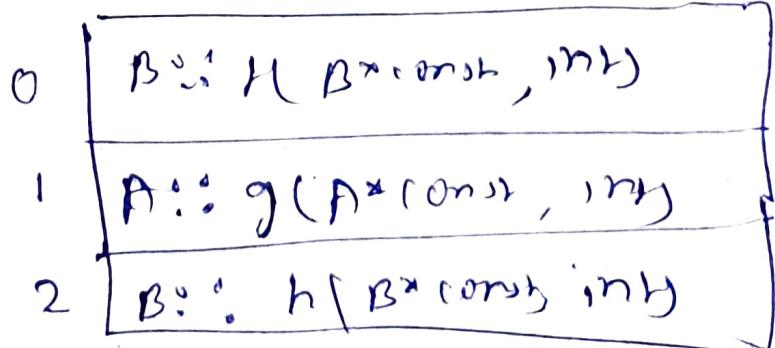
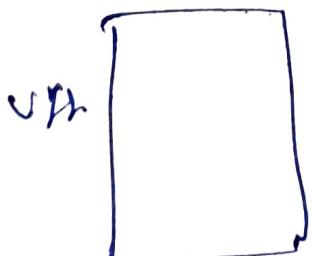
int h(B\*) {}

{};

a object layout



b object layout

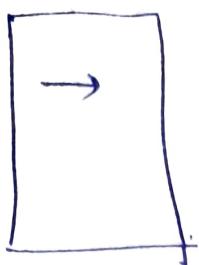


verb

(21)

## C object layout

VH:



0

1  
2

B:: f1 (char*, int)
C:: g (char*, int)
C:: h (char*, int)

A a; B a; C c;

A \*PA; B \*PB;

source exp

PA → f(2)

compiled exp

PA → vHEO (PA, 124);

PA → g(3, 2)

PA → vHTU (PA, 3, 2);

PA → h(2a)

A:: h (PA, 20);

PA → h(2b)

A:: h (PA, 2b);

PB → f(2)

PB → VHLO (PB, 2);

PB → g(3, 2)

PB → vHTU (PB, 3, 2);

PB → h(2a);

PB → VHLO (PB, 20);

PB → h(2b)

PB → VHLO (PB, 2b);

## Type casting

(212)

### Why casting

Casters used to convert the type of an object expression, in arguments or return values to that of another type.

### Types of 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 from 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) to return

(double) i / j ; }

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

(213)

compiler generates

double f1 (int i, int j)

2

double temp\_i = i; //  
double temp\_j = j; // <sup>conversion</sup> conversion  
return temp\_i / temp\_j;

3

Casting in C and C++

Costing in C

- \* implicit cast
- \* explicit "
- \* loses type information in several contexts
- \* unclear clarity of semantics

Casting in C++

- \* performs fresh inference of types without charge of value.
- \* perform fresh inference of types with charge of value.

Both can be done

- (1) Using implicit computation
- (2) Using explicit (user defined) computation

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

- (1) const - const (214)

(2) static - const

(3) reinterpret - const

(4) dynamic const

- \* Cast Operator can be greped in source
- \* @ style cast must be avoided in C++

## Cost operations

Cast Operators

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



Case C + 2) (i)

↑  
forgeth.

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

## (2) const-cast Operators

- \* const-cast Operators converts between types with different cv (const/volatile) qualification.
- \* Only const-cast may be used to cast away (remove) constness 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

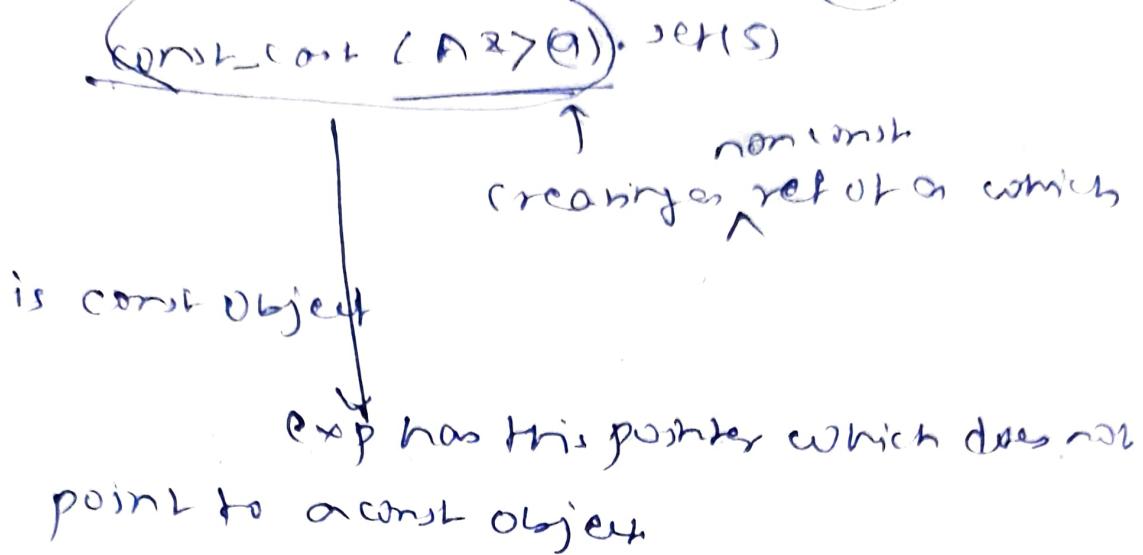
new from c

```
const A a(1);
```

a.get()

a.set(s) // error

(218)

const-const (A>(a).set(s)) // error

\* we can not strip the constness of whole  
Object bcoz it mean a completely diff object

exp.2 const-const versus const-style const

int main()

```
const char *c = "Sample Text";
printf( const-const char*>(c));
printf( (char*)(c));
```

const const At 1

const A a(1);

```
const-const < A&>(a).set(s);
((A&)a).set(s)
```

217 const const < A> (a).size() // error  
((A)a).size() ✓ ok in ~~single~~ type cast

ex-3

struct type2 type U : i(3) { }

void mi (int v) const {

\* this->i=v // Not allowed

const\_cast < type2 > (this)->i=v // allowed

    {

        int i;

}

int l=3

const int & cret = l

const const < int > (cret) = 4 // ok

cout << "l = " << l << endl;

type +

~~const~~ m

for form(y):

cout << "type :: l = " << ~~l~~ <sup>f.i</sup> l <sub>endl</sub>

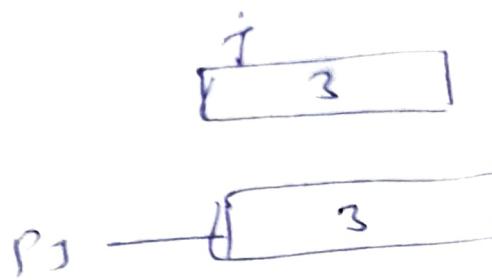
Undefined behaviour value of j and p<sub>j</sub> may differ

const int j = 3

int \*pj = const\_cast < int \* > (&j);

\*pj = 4

cout << j << " " << \*pj << endl;

bwoz

whild doing  
conversion  
will create  
10<sup>n</sup>

void (type:: \*mp)() const = &type::  
m1

const const void (type:: \*)(m1)  
(mp);

Q19: i = 4

type:: i = 4

3, 4

Note: constrem const  
be remove from function  
pointer

### Static-const operators

- static-const 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

~~(a)~~ static\_cast can perform conversion b/c it pointed to related classes.

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

(b) No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destn type.

\* Additionally, static cast can also perform the following:

- \* explicitly come single argument constructor or a conversion operator i.e. cast through user defined cast

- \* convert to rvalue references

- \* convert enum class values into integer or floating point values

- \* convert any type to void, eval using and discarding the value.

Ex. 1 static cast Operator built-in types

int main()

{ int i = 2;

double d = 3.7;

double x pd = 2.4;

i = d // want implicit

i = static\_cast<int>(d);

we are trying  
to use more  
information.

(22)

d = i

d = static\_cast<double>(i)

d = (double)i

i = pd // implicit cast

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

l = (int) pd; // C-style cast OK

Risky we should use  
reinterpret\_cast.

static const 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

Q2)  $q = \&a //$  implicit cast

$q = \text{static cast } B^*(\&a) //$

static cast OK.

Risky westward use

dyanmic cast.

$q = (B^*)(\&a) //$  OK - C-style  
cast.

static cast operator pitfall

class Window & public: virtual void  
onResize(); - }

class SpecialWindow : public Window,

{ public:

virtual void onResize();

static cast Window & ( \*this )  
onResize()

// ~~pre~~ cast cast & mix to  
Window and her calls  
In Resze this doesn't  
work

- // do SpecialWindow specific  
cast.

}

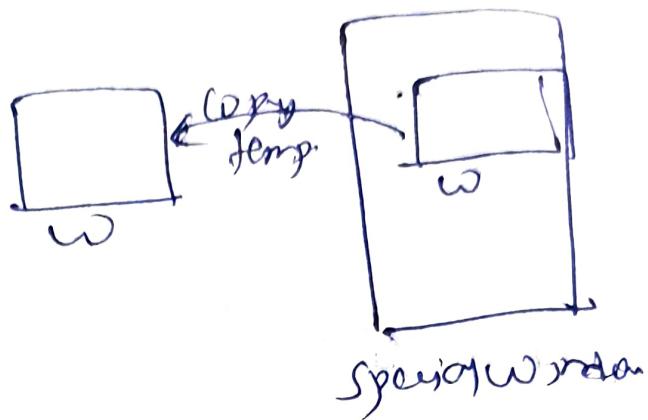
↓ This casting the whole object

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

window w (x,y)

(22)

this slices the object, creates a temporary window from base port or special window and calls the method



- 501^n

class SpecialWindow : public Window {

public:

virtual void onResize() {

SpecialWindow::onResize();

- - // do window specific work

{

    - - Here this pointer of special

window gets implicit compared to window

(223)

static cast operator unrelatedclassics

class B;

class A {

public:

{};

class D {};

int main() {

A a;

B b;

int i=5; we have an object of type B  
 and we need to use it  
 $\textcircled{1} \text{ B} \Rightarrow \text{A}$  to create an object  
 of type A.

a=b // error

a = static\_cast&lt;A&gt;(b) // error

a = (A)b; // error

// int  $\Rightarrow$  A

a = i // error

a = static\_cast&lt;A&gt;(i) // error

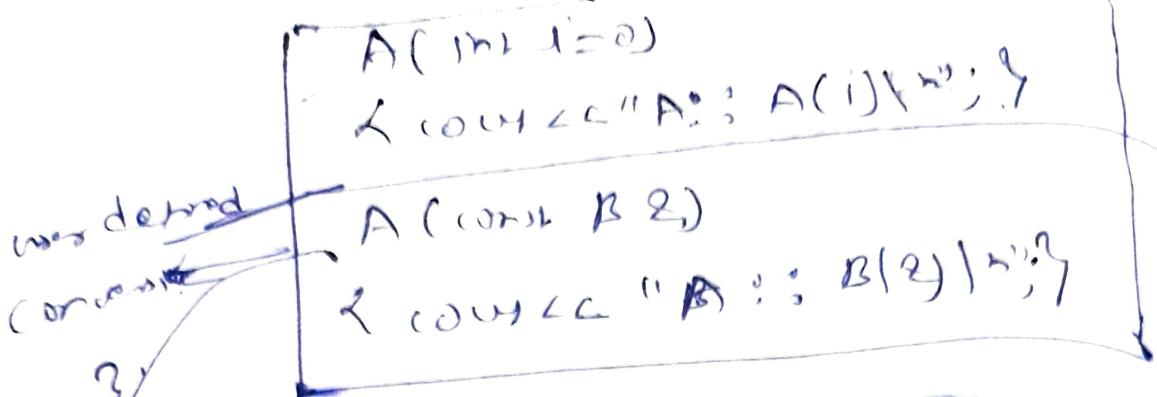
a = (A)i // error

This can be done using ~~as~~ user defined  
 conversion in the form of constructor  
 provides a constructor of target type which  
 takes a source type.

class B:

(224)

class A : public B



class B & {

A a

B b

int i =

a = b // uses A::B(2)

a = static\_cast<A>(b) // uses A::B(2)

a = ~~(A&)~~ (A)b // uses A::B(2)

$b = b' \text{ (type A)}$

$A = b'$

It takes a B object and create b' //  
object of type A and assign it to  
a copy free copy assignment operator

$a = i$  // uses A::A(i)

$a = \text{static\_cast<} A > (b)$  // uses A::A(i)

$a = (A)i$  // uses A::A(i)

(225) Was defined conversion provide or approximate type of ~~conversion~~<sup>constructor</sup> for ~~data~~ 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  $\Rightarrow$  A

a = b; // error

a = static\_cast<A>(b); // error

a = (A)b // error

// A  $\Rightarrow$  int

i = a // error

i = static\_cast<int>(a) // error

i = (int)a // error

↓

Can be solved using cast