A& operator+ (...)

const ?

① void foo(const int * A)

int const * A

int x;

foo2(x);

x = 5;

class B{

private:
int mem;
};

② Making the members read-only:

B b;
b.foo2(...);

a = (b = c)

· Active recall
· Spaced repetition

const B * const this;

const B * this

(*this)

const

void foo2(int const & A)
{
this = 0x500
A = ...
this->mem = ...
}

void setMem(int a)
{
mem = a;
}

Bconst * const this

How do we pass parameters through a function interface?

Signature

Copy constr not invoked

	Speed?	Safe?	Can it be nullptr?
void A(B b)	SLOW	✓	X
void A(B * b)	FAST	X	✓
void A(const B * b)	"	✓	✓
void A(B& b)	"	X	X
void A(const B& b)	"	✓	X

Inheritance : A class can inherit properties from a more general class,

built-in

Eg. Shopping List can inherit ↙ int

properties from a more general List class.

Array a;

↓

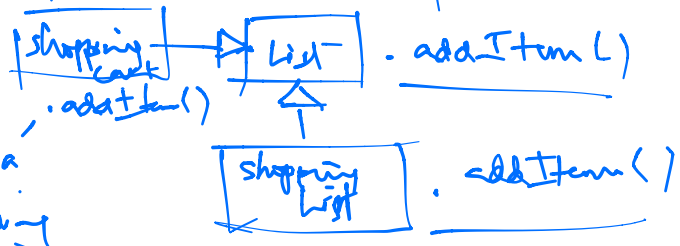
Abstract Data types.

↓

Invariant

Polymorphism: One method call works on several different classes, even if the classes need different implementations.

Eg. addItem() method works on every kind of List, even though adding an item to a ShoppingList is very diff. from adding an item to a ~~list~~ ShopingCart.



Object-oriented: Each object knows its own class and methods.

Eg. Each shoppingList and shoppingCart knows which addItem() method is used.

---

class base { → Abstract class

base b; ✗

public:

void f1(); // f1 cannot be overwritten

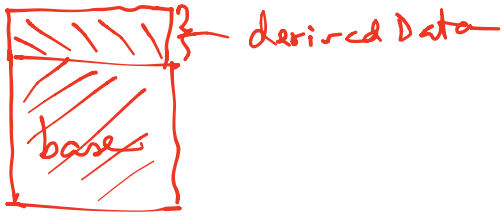
virtual void f2(); // f2 can be overwritten

virtual void f3() = 0; // f3 must be overwritten

};

↓

Pure virtual function



class derived: public base {

public: → void f4();

void f1();

virtual void f2();

virtual void f3();

* Relationship:

"derived is a kind of base"

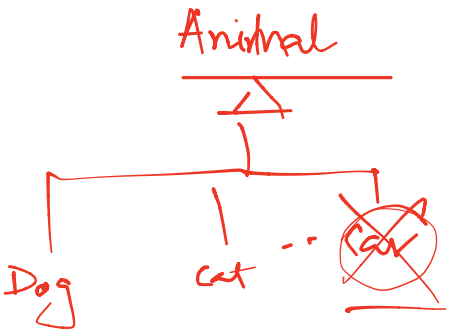
* derived class is kind of all

};

private:
int derivedData;

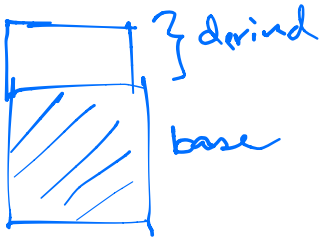
a base class.

* A derived class can access only the **protected** and the public members of the base class.



* If a class contains a pure virtual function, such a class cannot be instantiated

⇒ Abstract classes

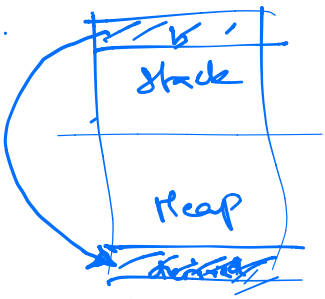


Heap

```
base * b = new derived(..);
```

Non-virtual → Static type of (*b) → base

Dynamic type of (*b) → derived.



Virtual

```
base * b;
b = new derived(..);
```

b → f1() invokes base::f1();

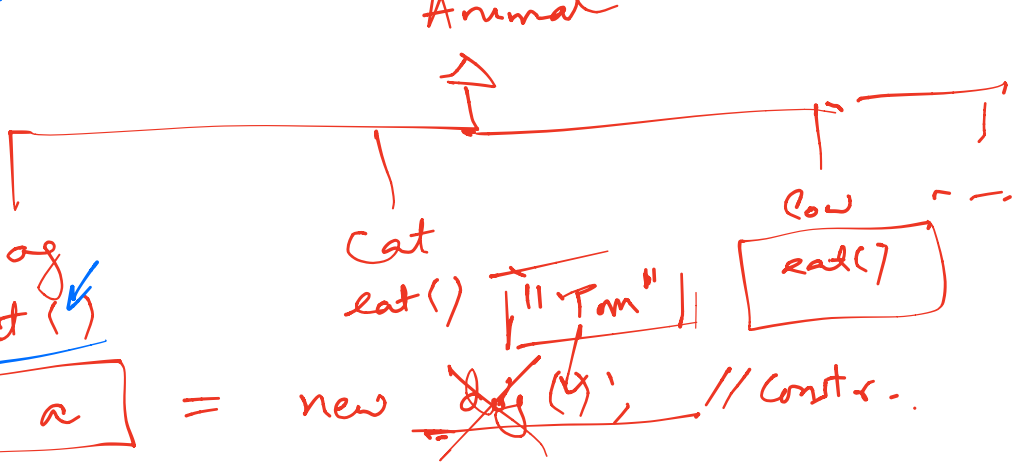
b → f2() invokes derived::f2();

b → f3() " derived::f3();

```
Base * b =
    new base();
```

```
Derived * d = new derived();
```

```
virtual eat() = 0
```




```
Animal * a = new Dog(); // Constr..
```

a → eat();

Cow

base

derived

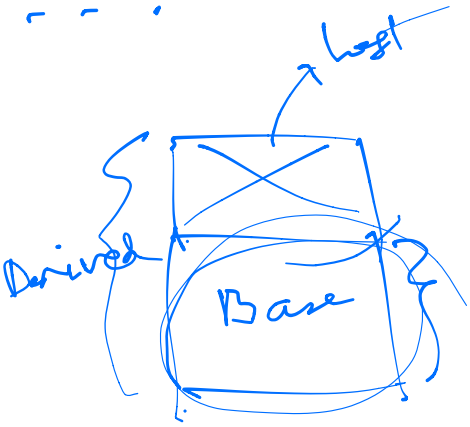
derived 2

class Dog : public Animal {
public:
virtual void eat();
private:
DogFood df;
};

class DogFood { item1() { ... }
public: string
private:
string item1;
string item2;
}

void Dg::eat() {
cout <<
df.item1();
- - -
}

Slicing → Information loss



void func(base b)
{ ... }

derived d;
func(d); // Copy constructor ?
// base b = d

pass by value

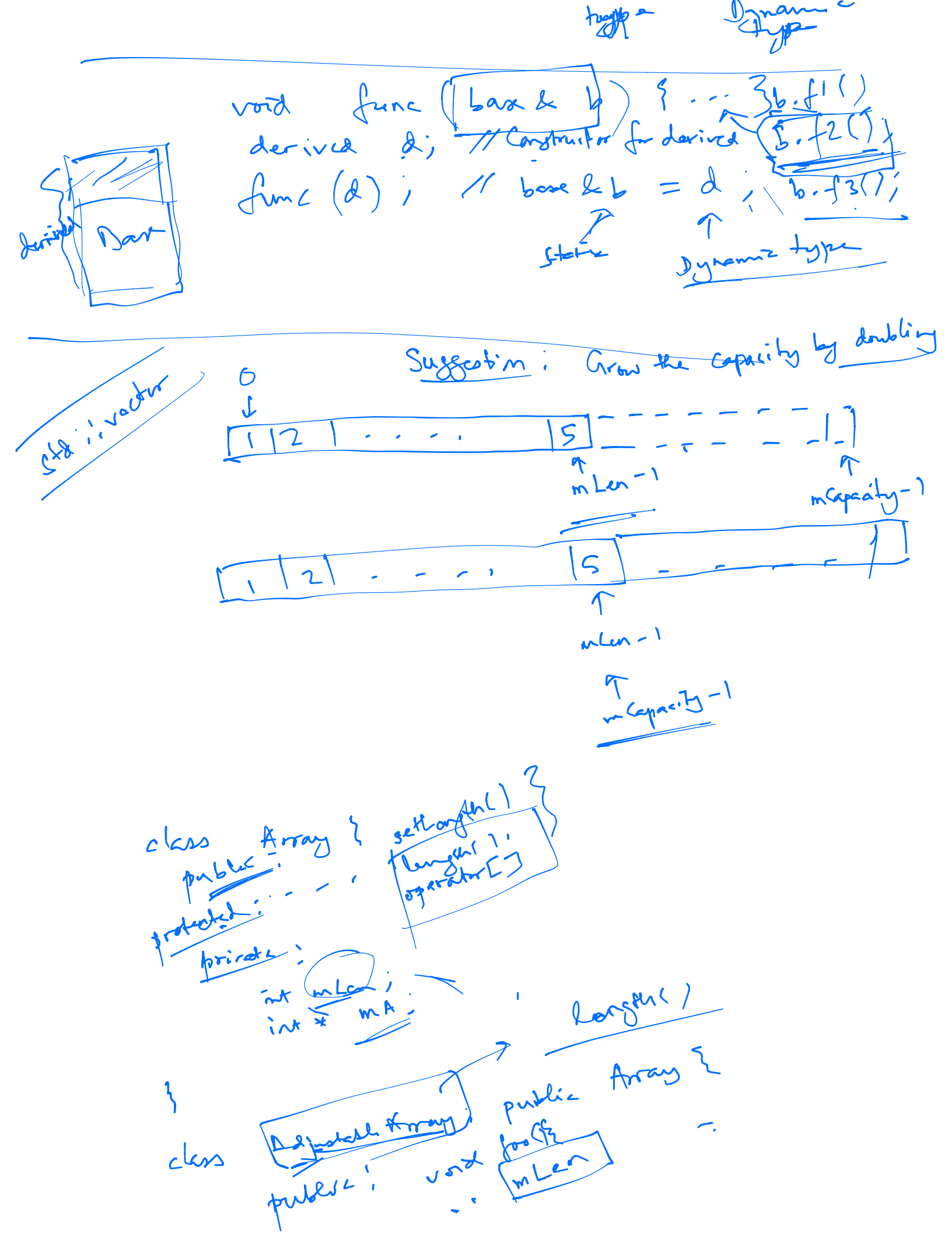
Lossless ways of passing the derived class

derived d2;
derived *d = &d2;

void func(base * b) { ... }

derived * d = - - - ; new derived;
func(d); // base * b = d;

static


Dynamic type
void func (Base & b) { ... b.f1()
derived d; // Constructor for derived
func (d); // Base & b = d;
b.f2()
b.f3();
Base
derived
Static
Dynamic type
Suggestion: Grow the capacity by doubling
std::vector
0
1 2 5
mLen-1
mCapacity-1
1 2 5
mLen-1
mCapacity-1
class Array {
public:
protected:
private:
int mLen;
int * mA;
setLength()
length()
operator[]
}
class AdjustableArray
public:
void
public
Array {
foo()
mLen
length()