class A:
    method A()
    method B()
    int i;

interface C
method F();

class B : A, C
    method F()
    int p;

B object (without C)

vptr ——> | A | B | F | |
int i;
int p;

addr 8 bytes
somewhere else

new A().A() ——> *(addr+0) ← jump too
(addr)

new B().A() ——> *(addr+0) ← jump to
(addr)

same interface

new A().i; → *(addr+8) ← read 4 bytes here

newB().i; → *(addr+8) ← "

Now let's say

```
class B : A, C
{
  method f()
  int p;
}
```

```
class D : C, A
{
  method f();
  int x;
}
```

now we have problem.

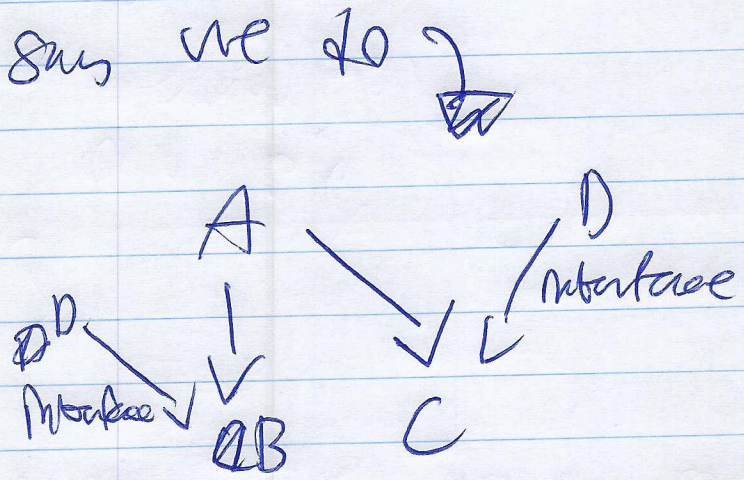
we can store the vtable items for f() in C at bottom and still have correct A behaviour.

However if we do so and say now f() does not exist then how do we locate?

Like how do we ever locate it.

Want, do I really understand
abstractions,

Says we do

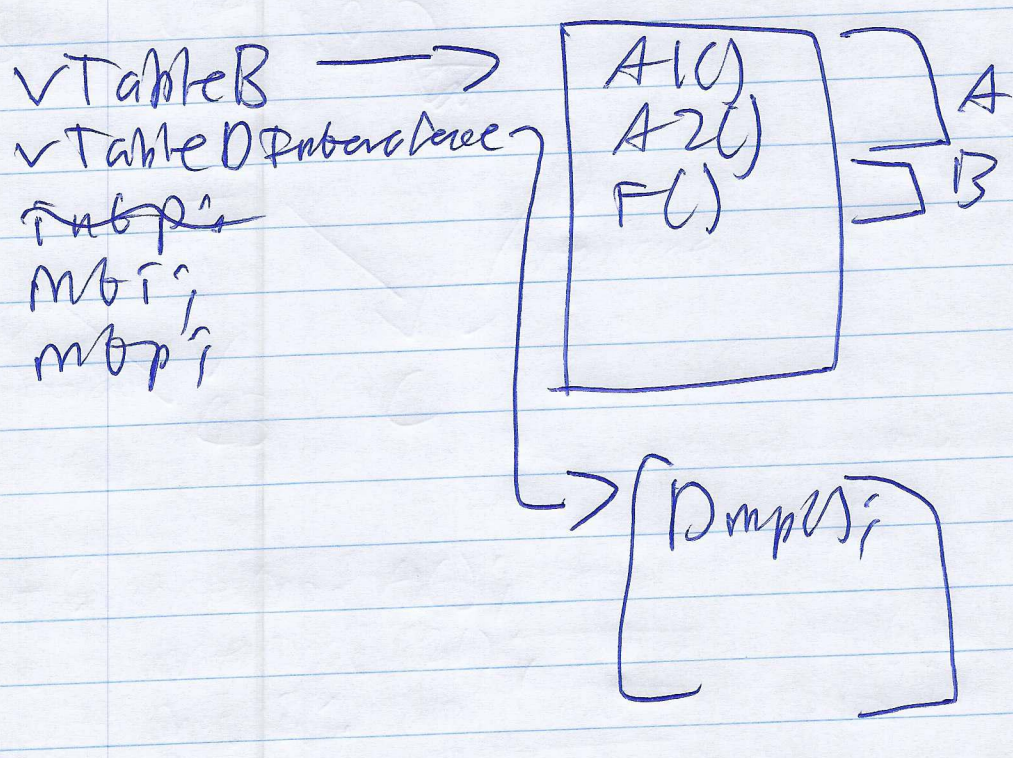


A: method A1();
method A2();
not();

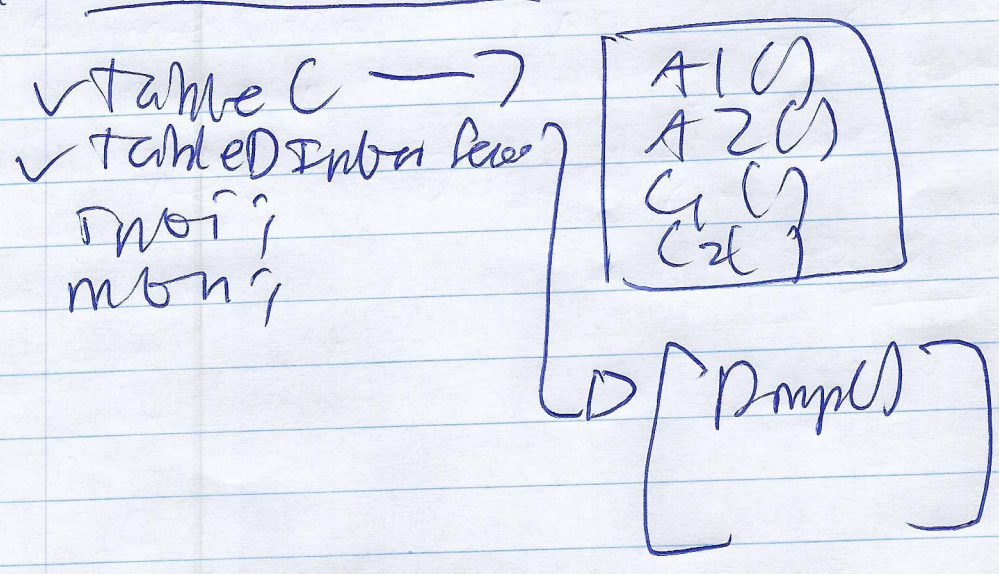
B: method F();
nop();
method Duplicate();

C: method C1();
method C2();
not();
method Duplicate

B object



C object



Obviously now calling A1, A2
B fine on both B object
and C object.

But common variable i
(from A) ~~can't be at different place, have to~~

is not same place.

In A object expected
at = (A - address) + 8 (strip variable)

Because A C and B have
variable for inheritance
used end up creating
a part of address (depending
on IS size)

Note

TypeID common to all impls of bsp.

1-way, ~~object kind of~~ ref type' A -> D easy.

Adjust ptr before passing by sizeof(typeID) + ~~4~~ (basically start at (vtableInterface-you-want).

~~As this~~

Now the interface methods can be accessed but 'this' is now incorrect (as it would try step 2 (not 1) vtable). And calling new methods would ~~fail~~ ('A and C - non interface overrides).

What we need is
the addition thunk.

At top of interface
vtables, we store
the real ones.

Can an interface methods
needing access to
our type T? is good ✓
our vtable for B ✓,
and inherited and been
vars ✓

Access to all methods
are stored at compile
time, so this is
dynamic but offsets
are known for inherited
interface vtables
non-inherited vtables,
vars and typeid)

Having typeid access seems easy, checks can be done (etc. typeid might be an array used to check hierarchy).
(ptr to an array)

Interfaces

casting to interfaces seems fine

however, what about

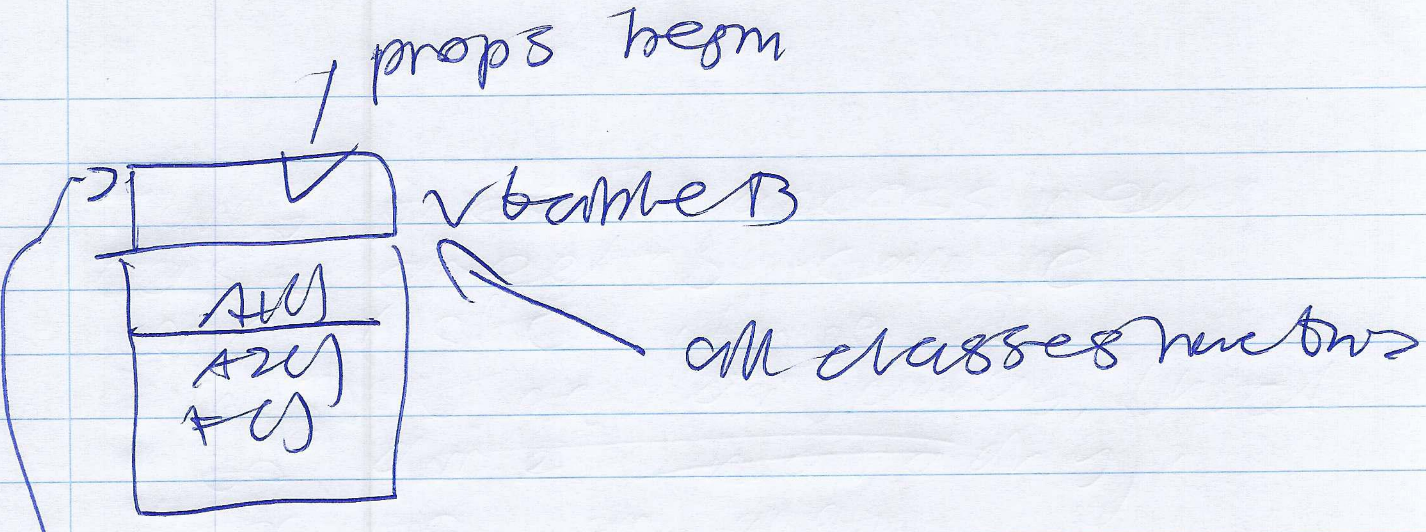
A a atype = new B();

atype.i; ← vtable not present
(need entry pointing to it!)

therefore store either this

(interfaces need local knowledge of complete tree)

→ we could for example do in vtableB (proper format)



So there A1() B at
~~A based offset everywhere
but if it B only implemented
in A call, overrides
that rename my~~

A1() offset same ✓

but reference, i field B
fetched now! The tons
B gap.

So references stored proper
tons and we know offsets
statics (local translated).

Now need need

to (at runtime,

base + thunk test to
after allocating

Therefore any field
reference must do

↓

① Go to top of this (go to this)
read either offset
or address offseted

② goto *(base + thunk + offset(field))
card read n-bytes)

This will always be true

* that means then
that each the
interfaces could use
load translate
(i.e., we always use
this

interfaces ones add
new vtable interface)

and then

Function is compiled
with B local variables
we'd say now etc.

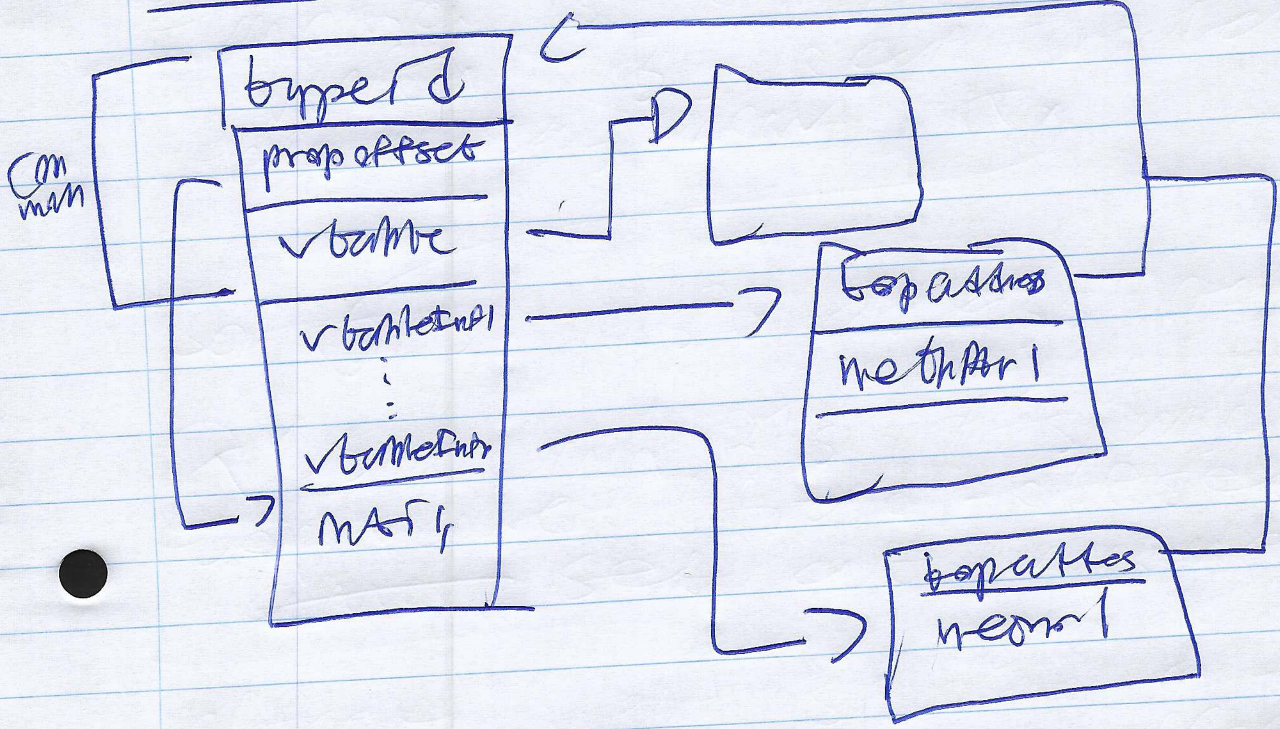
So this must always be used.

K's and B's, A's and C's
base value (fixed - offset ...)
~~are~~ set at runtime.

offset well is compile time.
Depends if address + offset
(runtime) or offset
(compile time)

Result from abs vs runtime
late add jump or push
jump.

Summary



- prop offset is compile time known
- ~~typ~~ address is ~~cas~~ ~~not~~ runtime known
- casting to interface
  - ~~if at~~ ~~at~~ on create to cast ~~cast to at~~
  - cast to offset of compile time, new known!!
  - typ addr preset to set any where
  - implemented methods local translator offset at compile one to any vtable

All pointer interference jmt after ~~A~~ P:C and D:C can m inter