Fares Mestre, Université Paris-Cité

The semantics of Object-Oriented languages for memory management

Sources :

Uday Reddy ; *Objects as Closures: Abstract Semantics of Object Oriented Languages*

Tahina Ramananandro ; *Formal Verification of Object Layout for C++ Multiple Inheritance*

Why bother ?

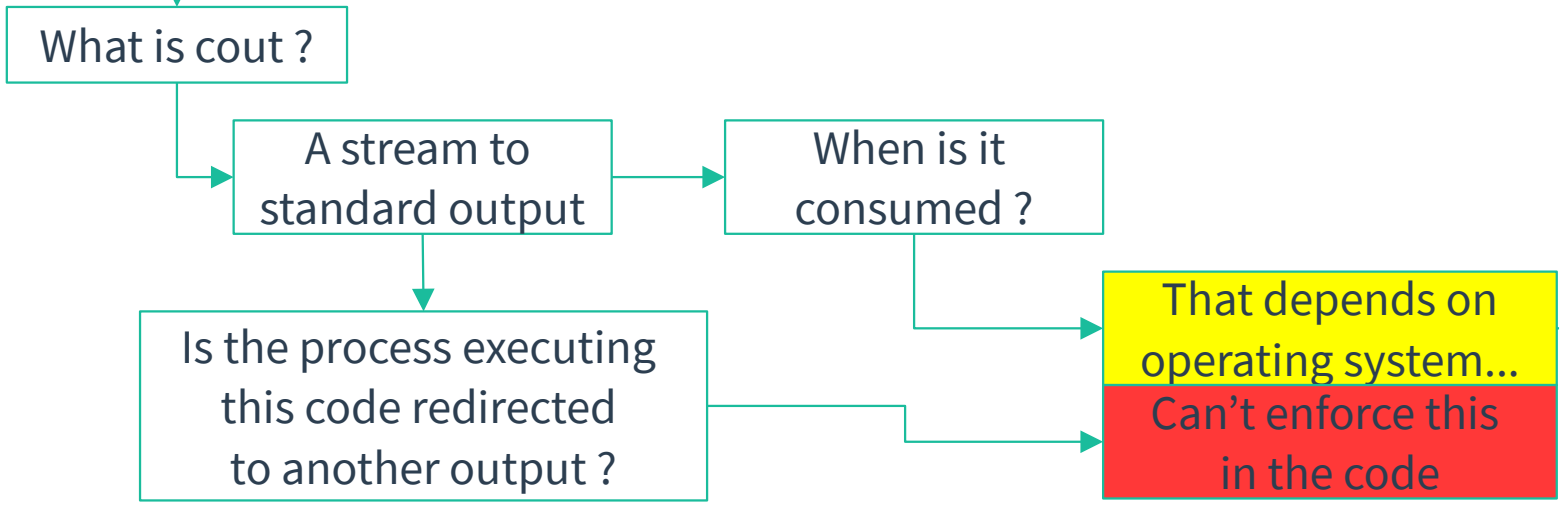
- Semantics give a formal framework with which one can reason with code
- Object-Oriented Programming is a dominant paradigm in Software-Engineering

Problem : Side-effects make it very difficult to reason on code

Attempting to reason on Hello World

```
void hello_world() {
    // Writes “Hello World!” to standard output
    std::cout << “Hello World!” << std::endl;
}
```

Is “Hello World” always written to standard output ?

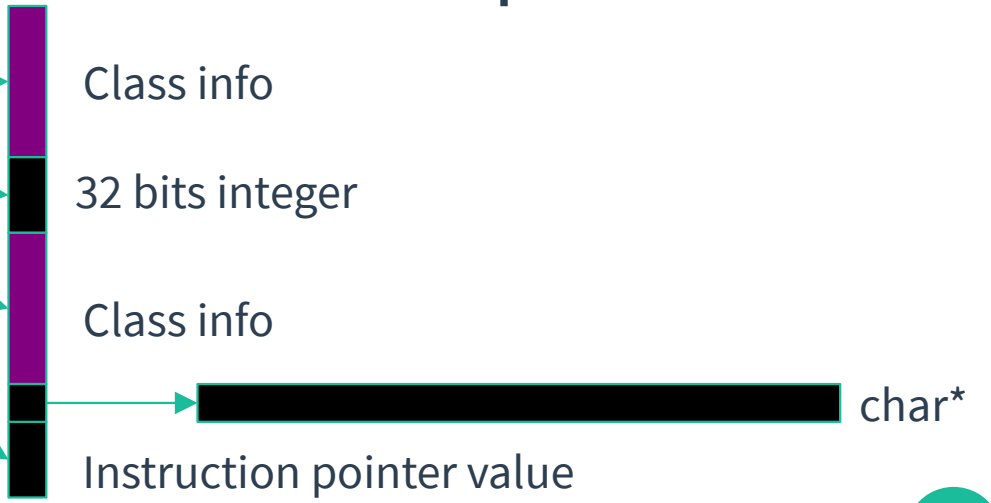


So what exactly is wrong with objects and verification ?

- Object-Oriented Programming relies on mutable objects

- Objects have to be stored somewhere and are complex structures

```
class Person {
    public:
        int age;
        std::string name;
        void introduction();
}
```



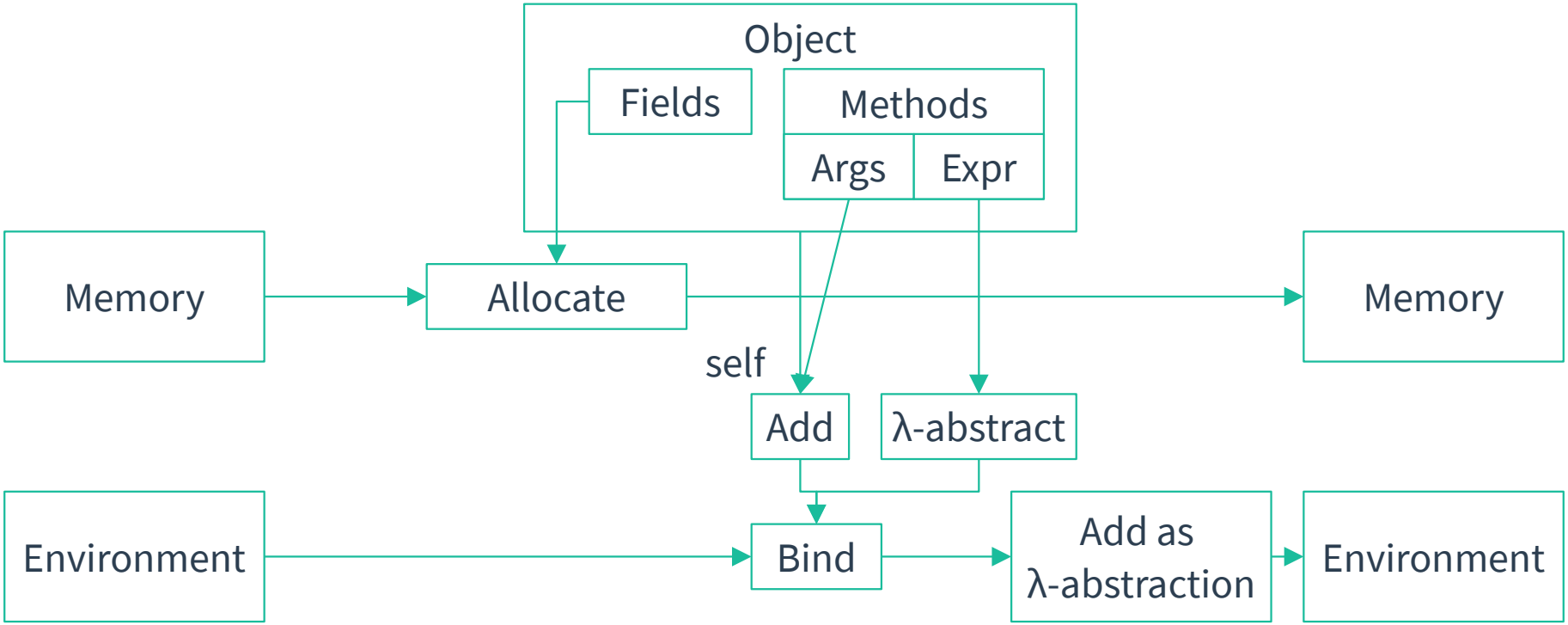
Perhaps we can reason on occuppied memory ?

Defining Objects

- **What is an Object, actually ?**
  - Fields (Instance Variables)
  - Methods (Messages)
- **How do methods work ?**
  - Methods only work within the context of their object

As such, Objects can be compared to a common closure for a set of functions

Defining Objects

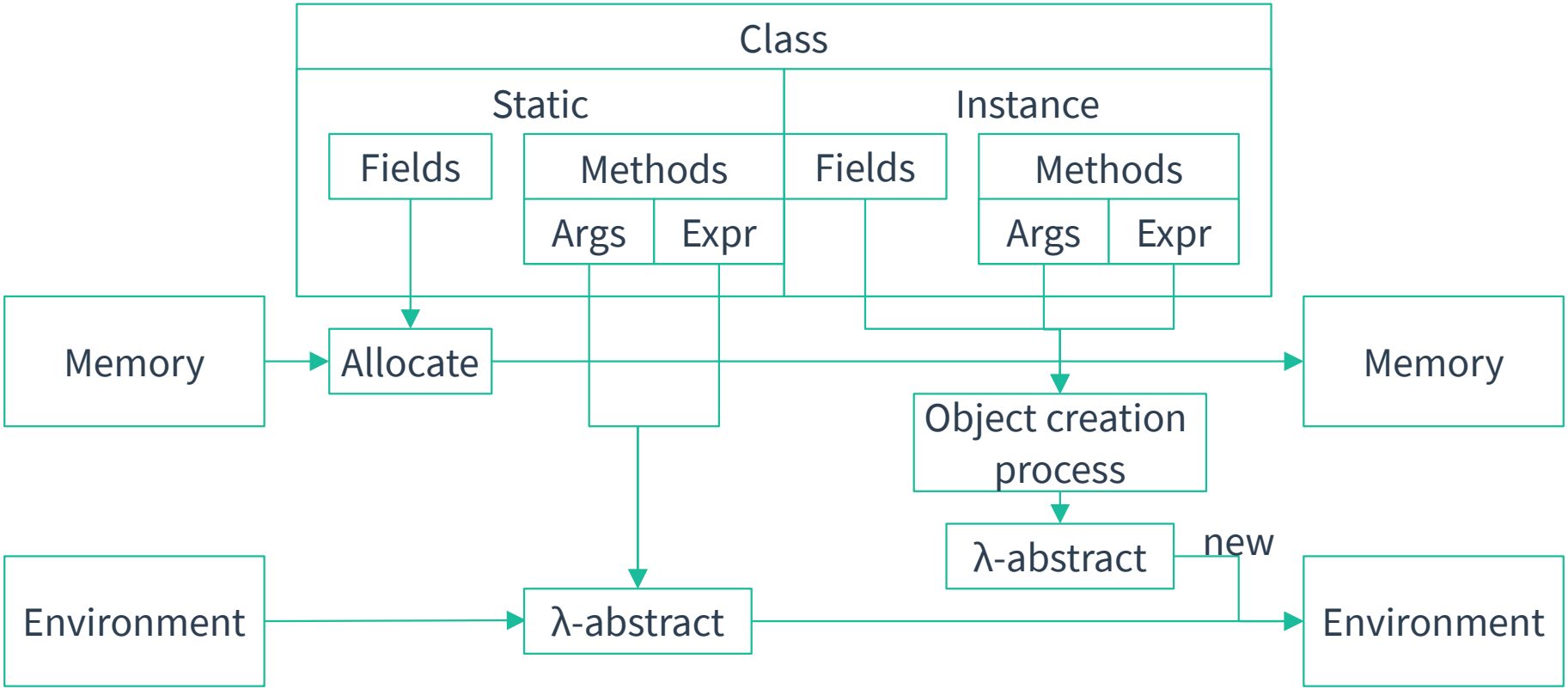


Defining Classes

- **What is the difference between Objects and Classes ?**
  - Classes have constructors that return objects
- **Are they just a template to create objects ?**
  - To some extent, but…
    - Static variables
    - Static functions

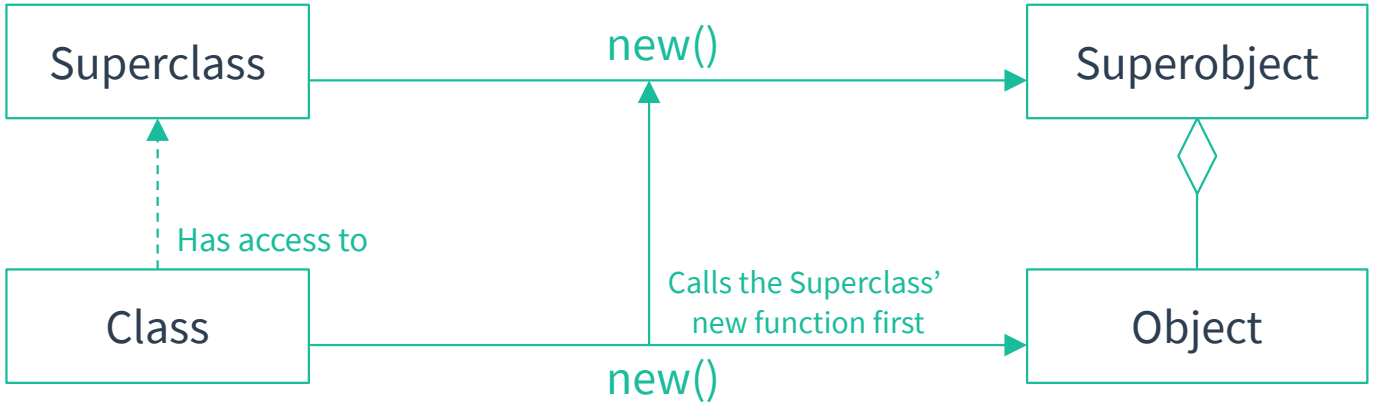
Classes themselves are Objects (and types)

Defining Classes



Defining simple inheritance

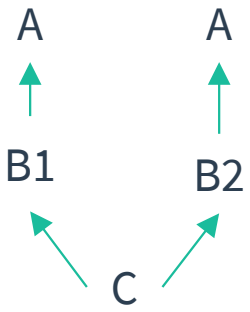
- **A whole new class based off of a superclass ?**
  - Not quite… it's actually nested
- **How does that reflect between classes and instances ?**



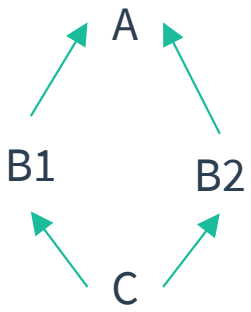
Defining multiple inheritance

- **It extends the principles of simple inheritance to an abritrary number of parent classes**

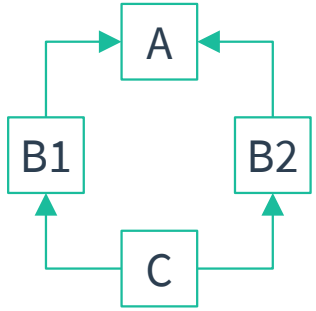
Instantiation process



Normally



With virtual inheritance



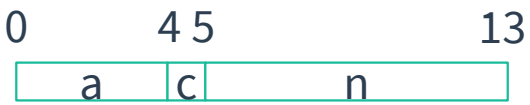
Class hierarchy

Layout of complex types in C

- Types are mostly used for storing allocation sizes

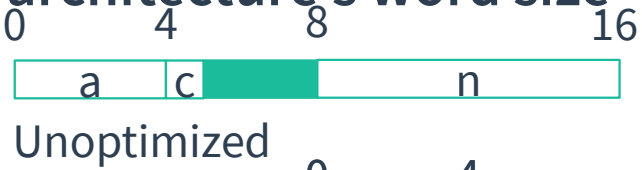
- Composite types like arrays and structs have their elements stored contiguously

```
struct s {
        int a;
        char c;
        long n;
}
```



The alignment problem

- Structs may not be aligned with the architecture's word size



- Tail padding is added to fit

- When embedded, structs have individual tail-padding

From C structs to C++ classes

- For regular methods, we could just use function pointers

```
class A {
        int x;
        int double()
};
```

What about self reference ?

```
struct A {
        int x;
        int (*double)(*struct A);
};
```

- Inheritance can be simulated with embedding

```
class B : A {
        char c;
};
```

```
struct B {
        A a;
        char c;
};
```

What about virtual methods and virtual inheritance ?

From C structs to C++ classes

- To work, virtual functions need two things :

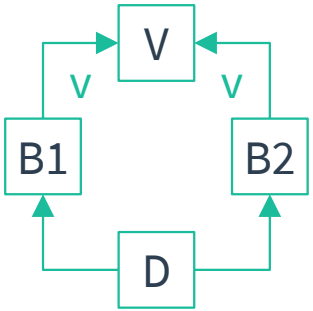
```
class B : A {
        char c;
        virtual void f();
};
```

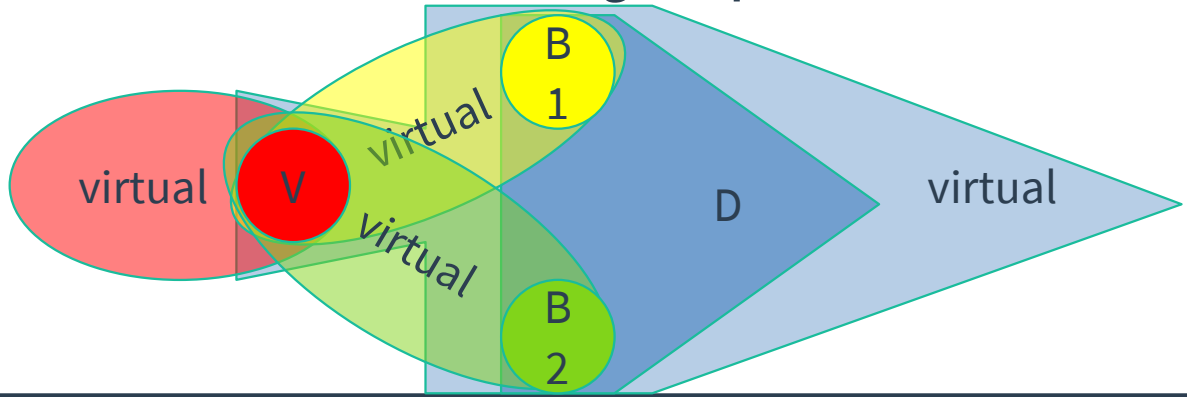
Dynamic type data

Type	Name
A	f
B	f

Virtual Table

- Virtual inheritance splits classes in virtual and regular parts





Layout of multiple inheritance

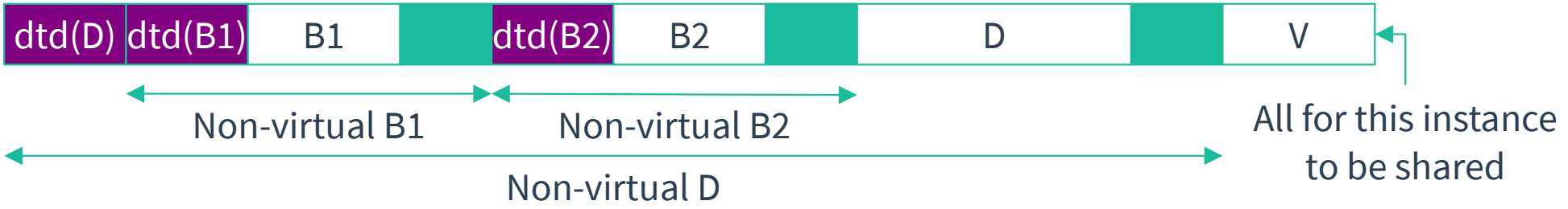
- The layout for an instance of D would look like...

```
struct v {
        non_virtual_v v;
};
```

```
struct b1 {
        non_virtual_b1 b1;
        non_virtual_v v;
};
```

```
struct b2 {
        non_virtual_b2 b2;
        non_virtual_v v;
};
```

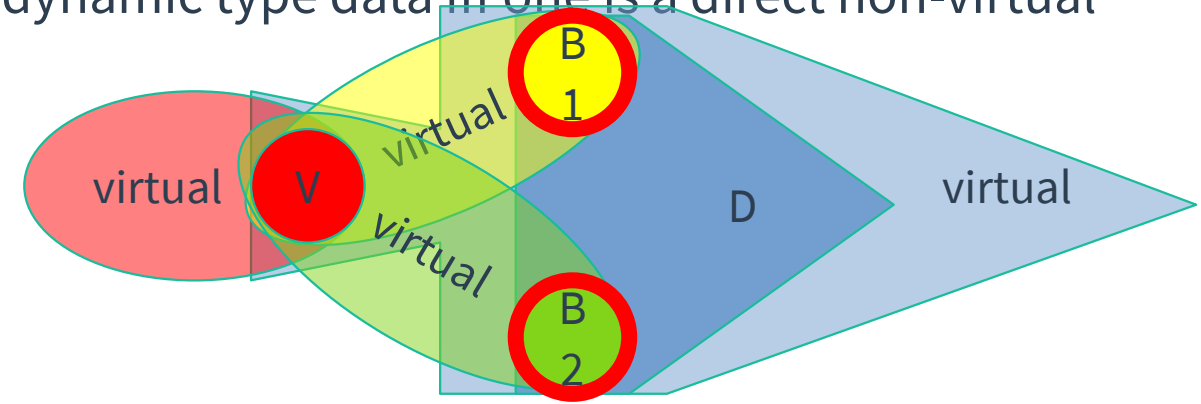
```
struct d {
        non_virtual_b1 b1;
        non_virtual_b2 b2;
        non_virtual_v v;
};
```

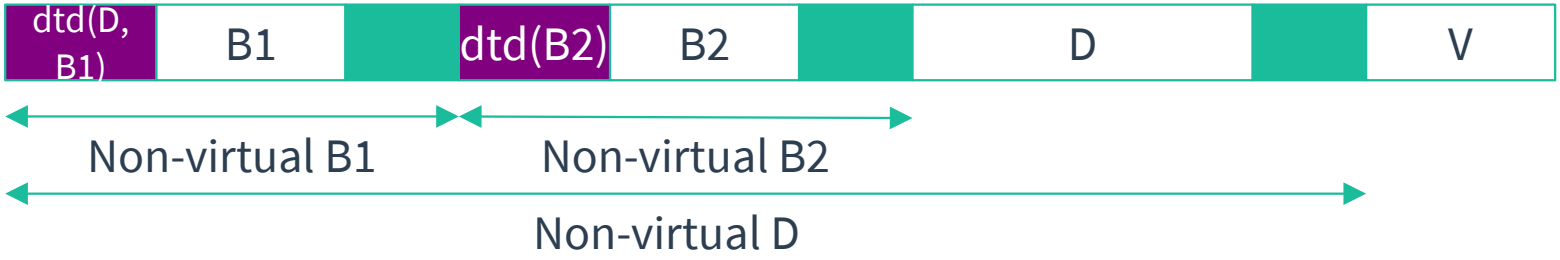


Perhaps we can refine the semantics to save space...

Optimizing the object layout

- **Sharing dynamic data types**
  - Two subobjects may share dynamic type data iff one is a direct non-virtual base of the other





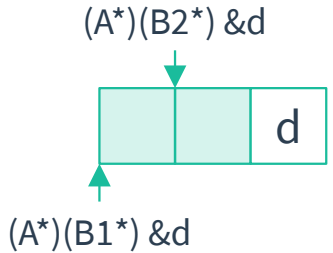
Optimizing the object layout

- **Reusing tail padding**
  - Defining the non-virtual size of a class helps avoid the excessive use of tail padding like embedded C structs

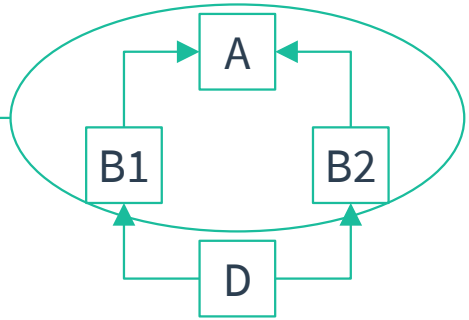
Optimizing the object layout

- **Optimizing empty base classes**

  Naive approach for d, an instance of D

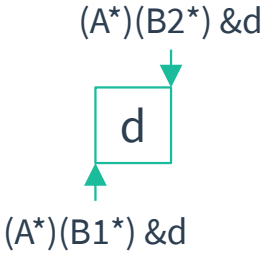


Empty

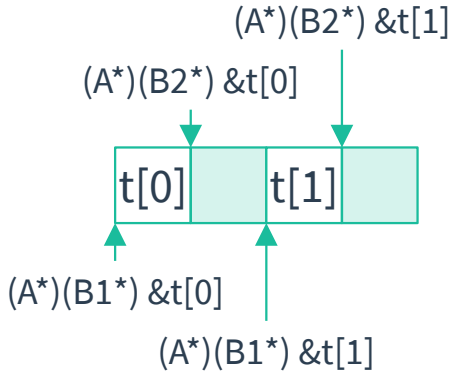


Empty classes have a size of 1

Overlapping empty bases



Caveat : structs will need a bit of foot padding



In conclusion

- In spite of being a side effect, we can still reason on memory allocation
- Formally defining Object Oriented concepts helps lay them out efficiently
- There is often room for semantic improvements to save space


Questions ?
Yes please :)