1 Introduction to OOAD

1.1 Overview and Schedule
1.2 What is OOAD?
1.3 Why OOAD in Physics?
1.4 What is an Object?
1.5 Objects and Classes
1.6 Object Interface Separation and Class Inheritance
1.7 Summary

1.1 Schedule

- Day 1: OO Concepts, UML
- Day 2: Principles of OO Class Design
- Day 3: Large Applications (Package Design)
- Day 4: OO Analysis and Project Management
- Day 5: Overflow Topics and Discussion

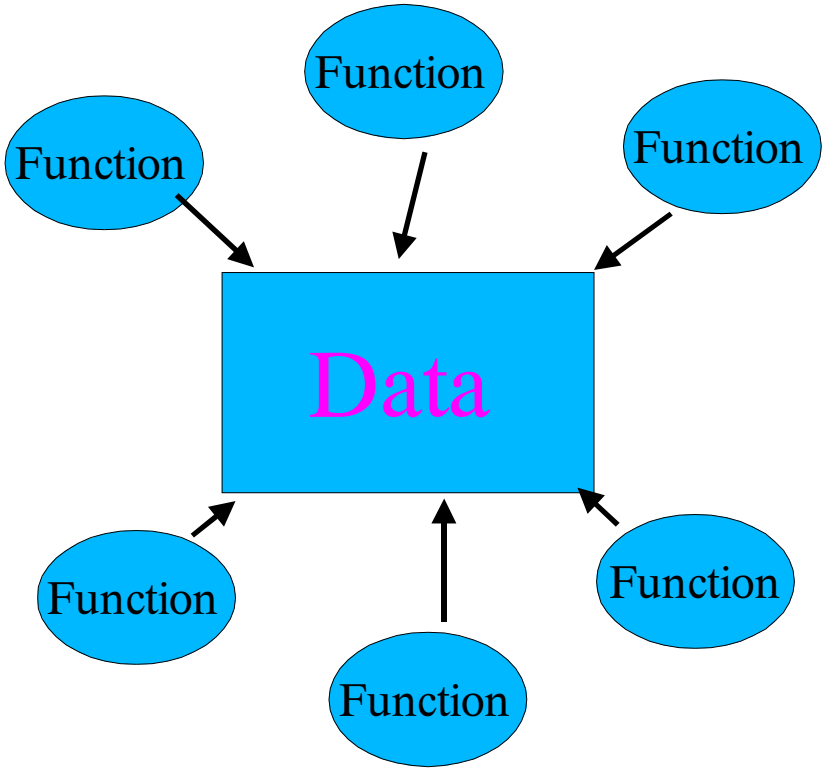
1.2 What is OO?

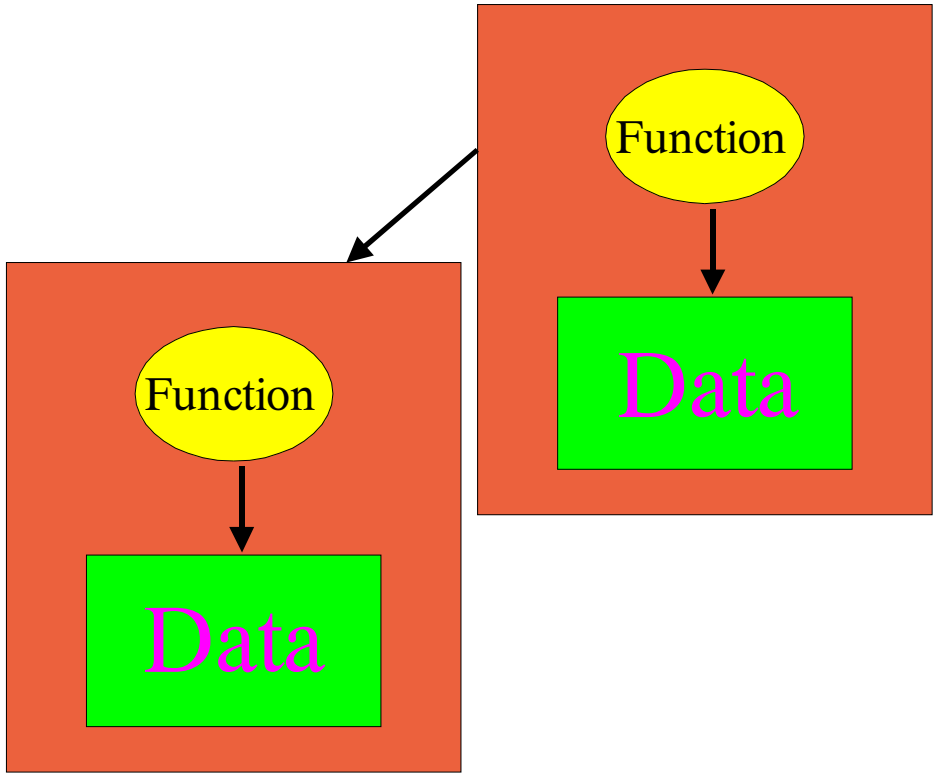
- A method to design and build large programs with a long lifetime
  - e.g. > 50k loc C++ with O(a) lifetime
  - Blueprints of systems before coding
  - Development process
  - Maintainance and modifications
  - Control of dependencies
  - Separation into components

1.2 Just another paradigm?

- Object-orientation is closer to the way problems appear in life (physical and non-physical
- These problems generelly don‘t come formulated in a procedural manner
- We think in terms of ”objects” or concepts and relations between those concepts
- Modelling is simplified with OO because we have objects and relations

1.2 SA/SD and OO





Top-down hierarchies of function calls and dependencies

Bottom-up hierarchy of dependencies

1.2 Common Prejudices

- OO was used earlier without OO languages
  - Doubtful. A well made procedural program may deal with some of the OO issues but not with all
  - OO without language support is at least awkward and dangerous if not quite irresponsible
- It is just common sense and good practices
  - It is much more than that, it provides formal methods, techniques and tools to control design, development and maintainance

1.3 Why OOAD?

- Software complexity rises exponentially:
  - 70‘s O(10) kloc (e.g. JADE)
  - 80‘s O(100) kloc (e.g. OPAL)
  - 90‘s O(1) Mloc (e.g. BaBar)
- Need for tools to deal with complexity ➔ OOAD provides these tools

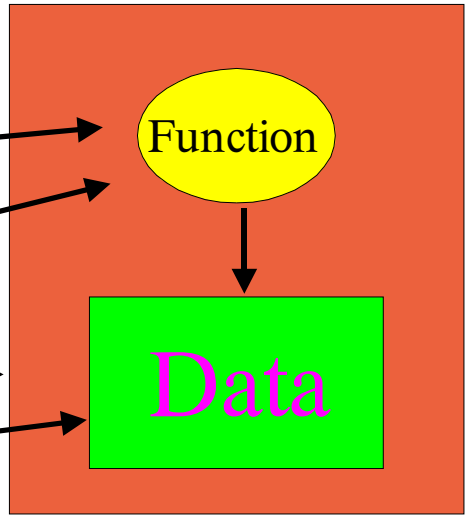
1.3 Why OOAD in Physics?

- Physics is about modelling the world:
  - Objects interact with each other according to laws of nature: particles/fields, atoms, molecules and electrons, liquids, solid states
- OOAD creates models by defining objects and their rules of interaction
  - This way of thinking about software is well adapted and quite natural to physicists
- OOAD is a software engineering practice
  - manage large projects professionally

1.4 What is an Object?

An object has:

- interface
- behaviour
- identity
- state



Interface:
Method signatures

Behaviour:
Algorithms in methods

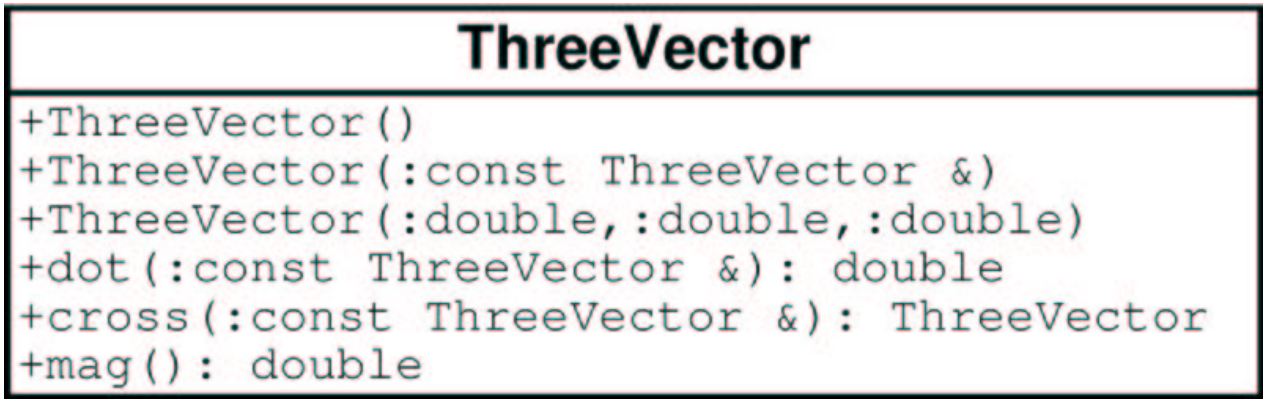
Identity:
Address or instance ID

State:
Internal variables

1.4 Object Interface

Create an object (constructors)
- from nothing (default)
- from another object (copy)
- from 3 coordinates

The object interface is given by its *member functions* described by the objects class



A dot product

A cross product

Magnitude

And possibly many other member functions

1.4 Object Identity

```
...

ThreeVector a;
ThreeVector b(1.0,2.0,3.0);

...

ThreeVector c(a);
ThreeVector d= a+b;

...

ThreeVector* e= new ThreeVector();
ThreeVector* f= &a;
ThreeVector& g= a;

...

double md= d.mag();
double mf= f->mag();
double mg= g.mag();

...
```

There can be many objects (instances) of a given class:

Symbolically:
$a \neq b \neq c \neq d \neq e$
but $f = g = a$

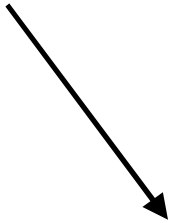
Pointer (*): Address of memory where object is stored; can be changed to point to another object

Reference (&): Different name for identical object

1.4 Object State

Different objects have
different identity
different state
possibly different behaviour
but always the same interface

The internal state of an object is given by its data members



p: ThreeVector
-x: double = 2.356 -y: double = 19.45 -z: double = -5.284 -n: int = 5
+ThreeVector() +ThreeVector(:const ThreeVector &) +ThreeVector(:double,:double,:double) +dot(:const ThreeVector &): double +cross(:const ThreeVector &): ThreeVector +mag(): double

Objects of the same class can share data (explicitly declared class variables)
e.g. static data members in C++

1.4 Object Behaviour

```
class ThreeVector {

    public:

        ThreeVector() { x=0; y=0; z=0 };
        ...

        double dot( const ThreeVector & ) const;
        ThreeVector cross( const ThreeVector & ) const;
        double mag() const;

        ...

    private:

        double x,y,z;

}
```

Default constructor sets to 0

Dot and cross are unambigious

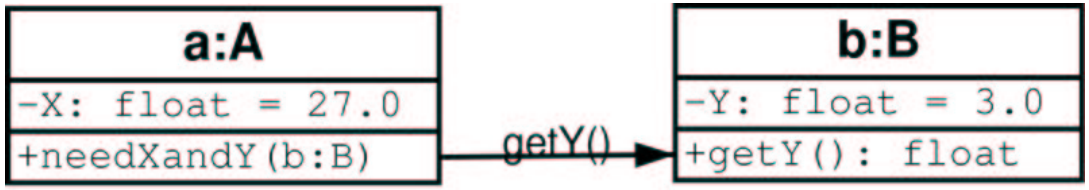
Magnitude, user probably expects 0 or a positive number

const means state of object does not change (vector remains the same) when this function is used

1.4 Object Interactions

Objects interact through their interfaces only

Objects manipulate their own data
but get access to other objects data
through interfaces

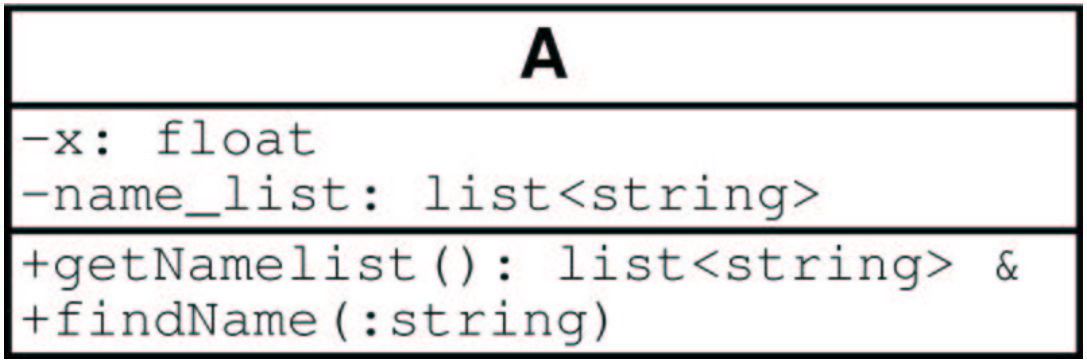


Most basic: get() / set(...) member functions, but usually better to provide "value added services", e.g.

- fetch data from storage
- perform an algorithm

```
A::needXandY( B b ) {
    ...
    float myY= b.getY();
    ...
}
```

1.4 Objects keep data hidden



Stop others from depending on the data model

Provide algorithms which use the data instead

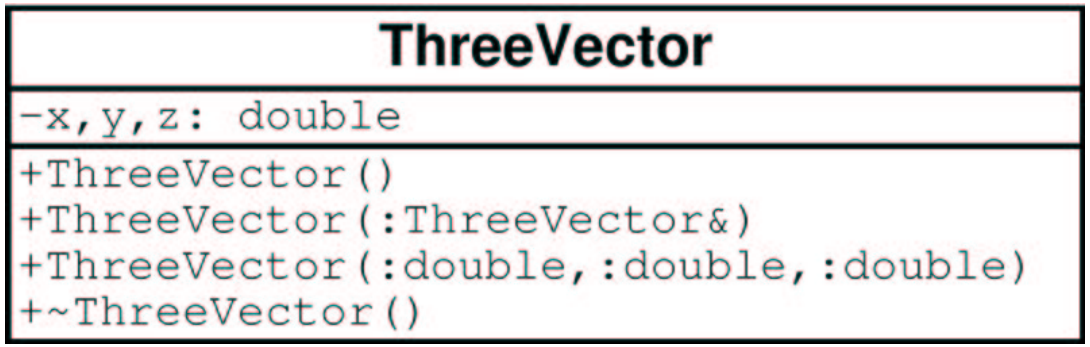
Can give direct and efficient access to data in controlled way

→ pass (const) references or pointers

Can change member data layout without affecting other objects

Can replace member data e.g. by database lookup

1.4 Object Construction/Destruction



Construction:

Create object at run-time

Initialise variables

Allocate resources

→ Constructor member functions

Destruction:

Destroy object at run-time

Deallocate (free) resources

→ Destructor member function

1.4 Objects Summary

- Objects have interface, behaviour, identity, state
- Objects collaborate
  - send messages to each other
  - use each other to obtain results
  - provide data and ”value-added services”
- Objects control access to their data
  - data private
  - access through interface

1.5 Objects and Classes

- Objects are described by classes
  - blueprint for construction of objects
  - OO program code resides in classes
- Objects have type specified by their class
- Classes can inherit from each other
  - implies special relation between corresponding objects
- Object interfaces can be separated from object behaviour and state

1.5 Classes describe Objects

- Class code completely specifies an object
  - interface (member function signature)
  - behaviour (member function code)
  - inheritance and friendship relations
- Object creation and state changes at run-time
- In OO programs most code resides in the class member functions
  - objects collaborate to perform a task

1.5 Classes = Types

- Class is a new programmer-defined data type
- Objects have type
  - extension of bool, int, etc
  - e.g. type complex doesn‘t exist in C/C++, but can construct in C++ data type complex using a class
- ThreeVector is a new data type
  - combines 3 floats/doubles with interface and behaviour
  - can define operators +, −, *, / etc.

1.5 Class Inheritance

- Objects are described by classes, i.e. code
- Classes can build upon other classes
  - reuse (include) an already existing class to define a new class
  - the new class can add new member functions and member data
  - the new class can replace (overload) inherited member functions
  - interface of new class must be compatible

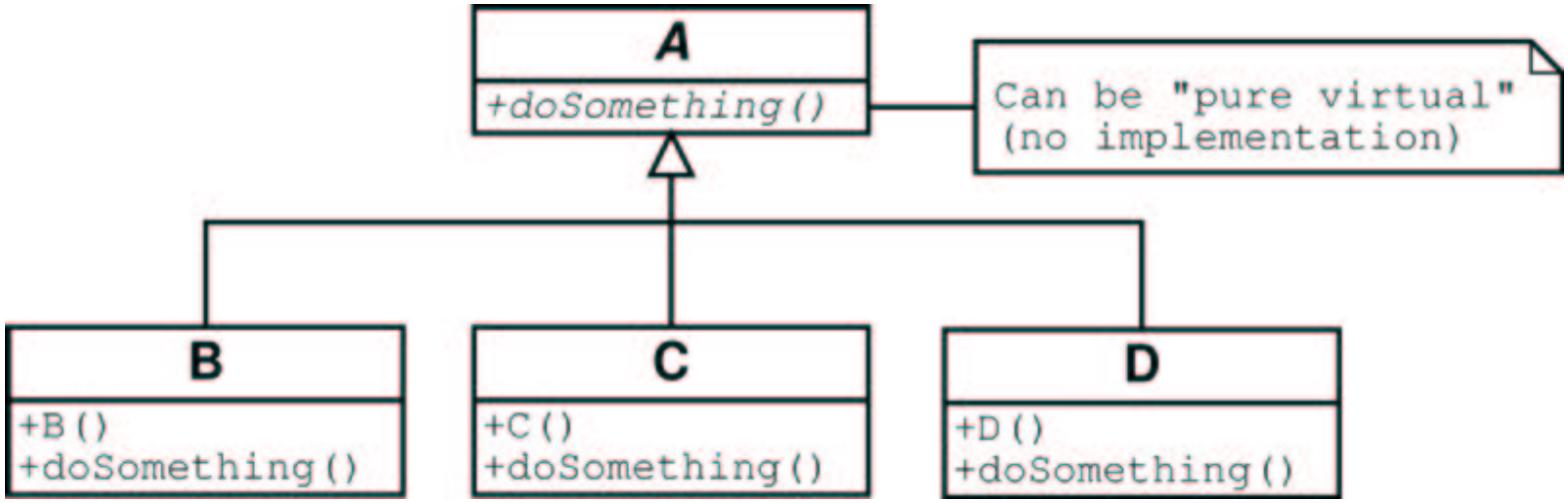
1.5 Classes Summary

- Classes are blueprints for contruction of objects
- Class = data type of corresponding objects
- Classes can inherit (build upon) other classes

1.6 Separation of Interfaces

- Interface described by class A with no (or little) behaviour
  - member function signatures
  - perhaps not possible to create objects of type A
- Now different (sub-) classes (B, C, D) can inherit from A and provide different behaviour
  - can create objects of type B, C or D with identical interfaces but different behaviour
  - code written using class A can use objects of type B, C or D

1.6 Object Polymorphism



Objects of type A are actually of type B, C or D

Objects of type A can take many forms, they are polymorph

Code written in terms of A will not notice the difference

but will produce different results

Can separate generic algorithms from specialisations

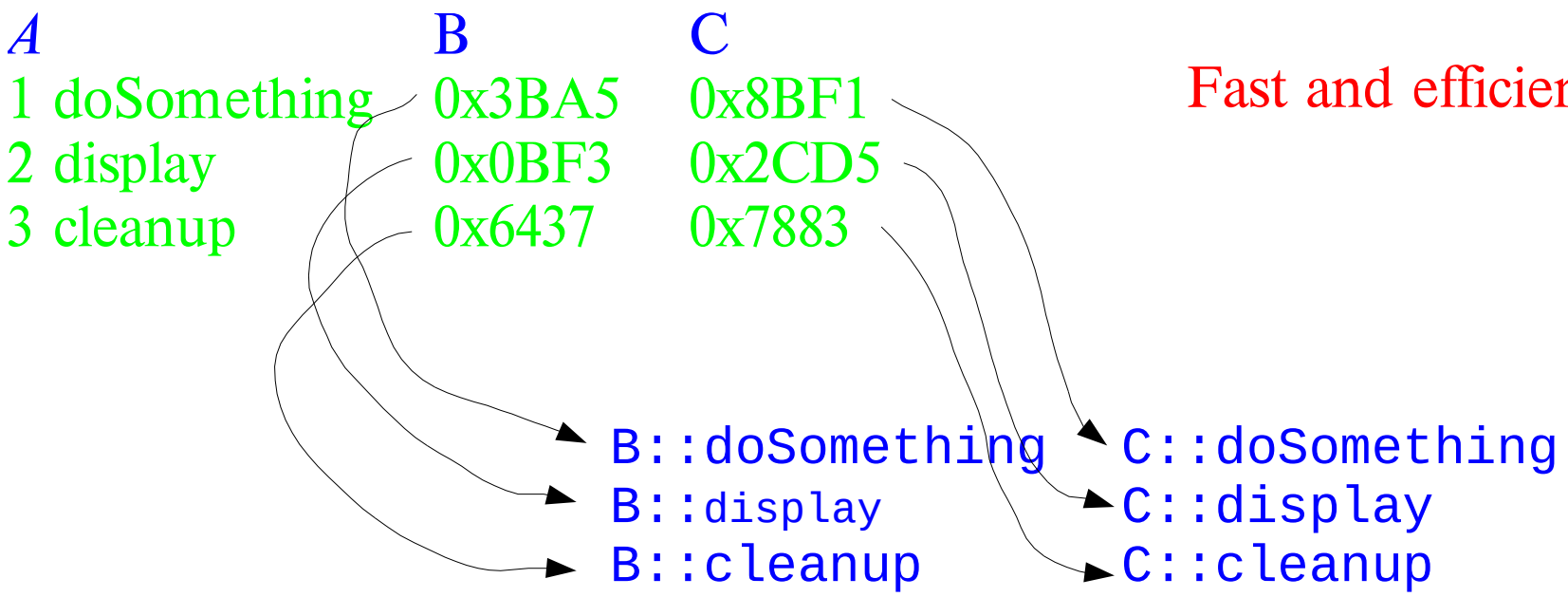
Avoids explicit decisions in algorithms (if/then/else or case)

1.6 Interface Abstraction

- The common interface of a group of objects is an abstraction (abstract class, interface class)
  - find commonality between related objects
  - express commonality formally using interfaces
- Clients (other objects) depend on the abstract interface, not details of individual objects
  - Polymorphic objects can be substituted
- Need abstract arguments and return values
  - or clients depend on details again

1.6 Mechanics of separated interfaces

Virtual function table with function pointers in statically (strongly) typed languages, e.g. C++, Java



Fast and efficient!

Lookup by name in hash-tables in dynamically typed languages (Perl, Python, Smalltalk)

1.6 Separated Interfaces Summary

- Interface can be separated from object
  - Abstract (interface) classes
- Express commonality between related objects
  - Abstraction is the key
- Clients depend on abstractions (interfaces), not on specific object details
- Mechanism is simple, fast and efficient
- Polymorphic objects can replace code branches

1.7 Inheritance SA/SD vs OO

SA/SD:

Inherit for functionality

We need some function, it exists in class A → inherit from A in B and add some more functionality

OO:

Inherit for interface

There are some common properties between several objects → define a common interface and make the objects inherit from this interface

1.7 Tools for OOAD

- A (graphical) modelling language
  - allows to describe systems in terms of classes, objects and their interactions before coding
- A programming language
  - Classes (data+functions) and data hiding
  - Interface separation (class inheritance and member function overload)
- Not required for OOAD (but useful)
  - templates, lots of convenient operators