1.6 Objects and Classes

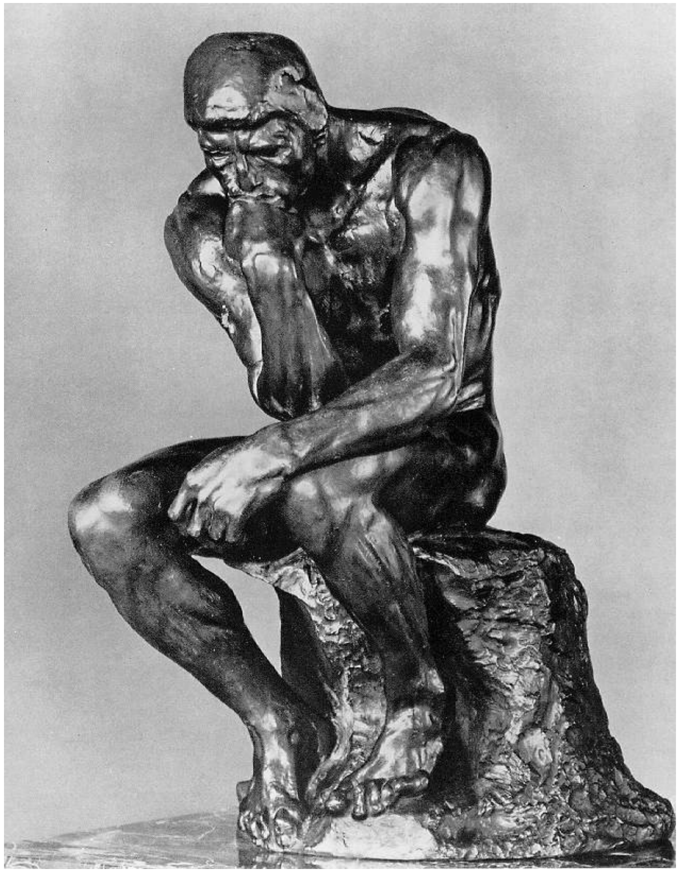
1.6 What is an Object?

1.7 Objects and Classes

1.8 Object Interface, Class Inheritance, Polymorphism

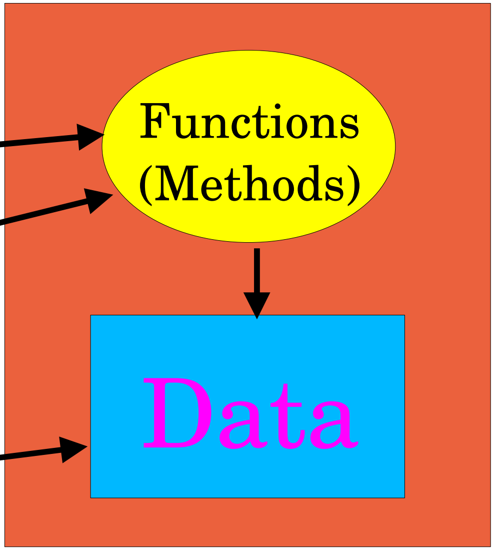
1.9 Summary

1.6 What is an Object?



An object has:

- interface
- behaviour
- identity
- state



Interface (how to use it):
- Method signatures

Behaviour (what it does):
- Algorithms in methods

Identity (which one is it):
- Address or instance ID

State (what happened before):
- Internal variables

1.6 Object Interface

How to use it

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

ThreeVector

```
+ThreeVector()
+ThreeVector(:const ThreeVector &)
+ThreeVector(:double,:double,:double)
+dot(:const ThreeVector &): double
+cross(:const ThreeVector &): ThreeVector
+mag(): double
```

A dot product

A cross product

Magnitude

And possibly many other member functions

1.6 Object Behaviour

What it does

```
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

1.6 Object Identity

Which one is it

```
...

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

1.6 Object State

What happened before

Different objects of the same class 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

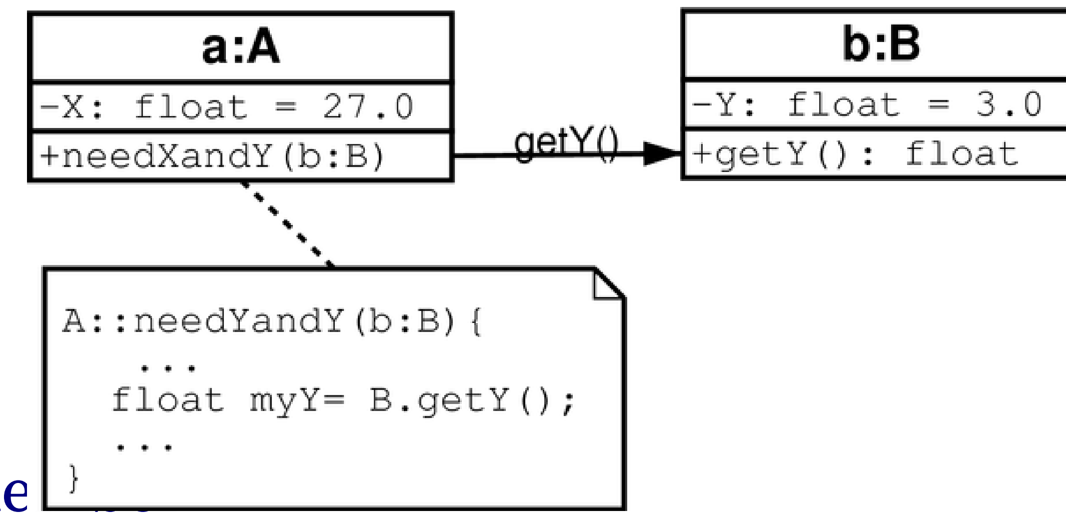
```
-x: double = 2.356
-y: double = 19.45
-z: double = -5.284
-n: int = 5
```

```
+ThreeVector()
+ThreeVector(:const ThreeVector &)
+ThreeVector(:double,:double,:double)
+dot(:const ThreeVector &): double
+cross(:const ThreeVector &): ThreeVector
+mag(): double
```

1.6 Object Interactions

Objects interact through their interfaces only

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

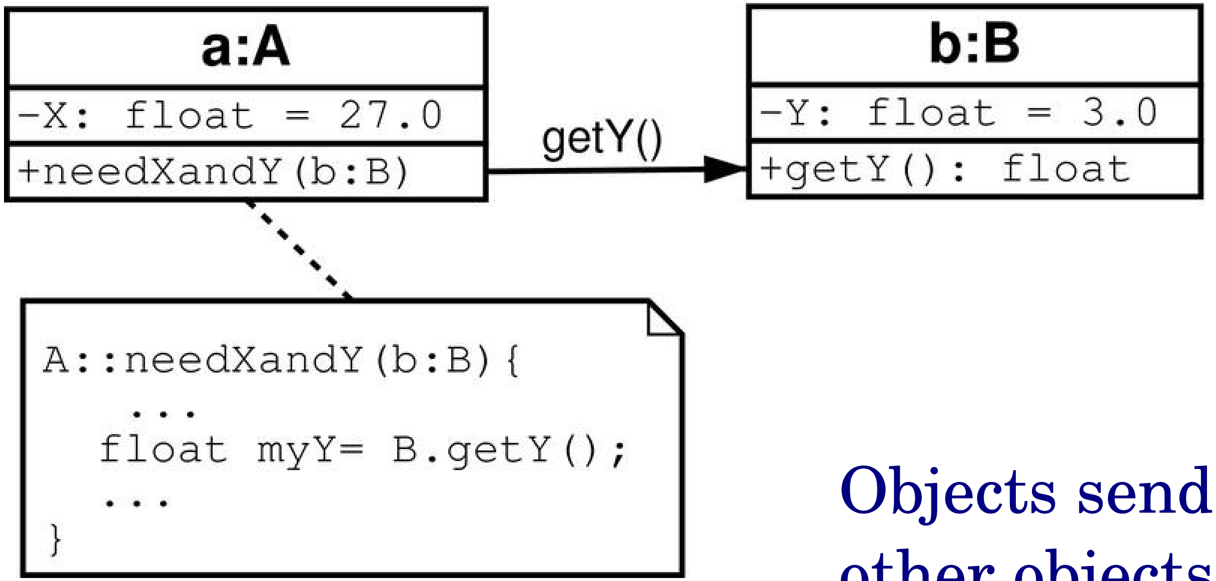


Most basic: get() / set(...) me functions, but usually better to provide "value added services", e.g.

- fetch data from storage
- perform an algorithm

1.6 Message Passing

Objects pass messages to each other



Objects send messages to other objects and react on messages

a says to b "getY" and receives a value for Y in return

1.6 Objects keep data hidden

Stop others from depending on the class 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 class data model without affecting other objects

Can replace member data e.g. by database lookup

1.6 Private Object Data

- Object state a priori unknown
- The object knows and reacts accordingly
- Decisions (program flow control) encapsulated
- User code not dependent on algorithm internals, only object behaviour
- Object state can be queried (when the object allows it)

1.6 Object Construction/Destruction

ThreeVector
-x,y,z: double
+ThreeVector() +ThreeVector(:ThreeVector&) +ThreeVector(:double,:double,:double) +~ThreeVector()

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

1.6 Object Lifetime

```
A* myA= new A();
...
delete myA;
```

Allocation creates new instance “on the heap”

constructor called

Must free resources by hand

destructor called

```
{
  A myA();
  ...
}
```

Declaration creates new instance “on the stack”

constructor called

Object will be deleted automatically when scope is left

destructor called

Or the language provides garbage collection (Java, Perl, Python)

1.6 Objects Summary

- Object: 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, state hidden
  - access through interface
- Objects have lifetime

1.7 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
  - Special relation between corresponding objects
- Object interfaces can be separated from object behaviour and state

1.7 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 happen at run-time
- In OO programs most code resides in the class member functions (methods)
  - objects collaborate to perform a task

1.7 Classes = Types

- Class is new programmer-defined data type
- Objects have type
  - extension of bool, int, float, etc
  - e.g. type complex didn't exist in C/C++, but can construct in C++ data type complex as class
- ThreeVector is a new data type
  - combines 3 floats/doubles with interface and behaviour
  - can define operators +, -, *, / etc.

1.7 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
  - add new member functions and member data
  - replace (overload) inherited member functions
  - interface of new class must be compatible
  - class has own type and type(s) of parent(s)

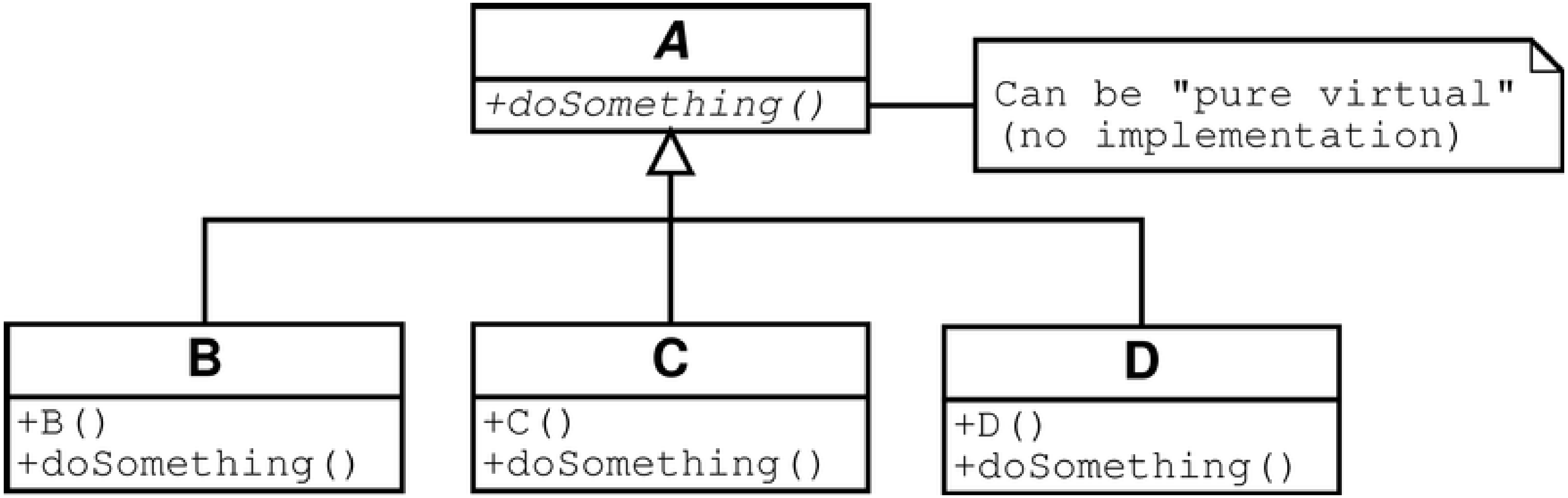
1.7 Classes Summary

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

1.8 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 subclasses (B, C, D) 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

1.8 Dynamic Polymorphism



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

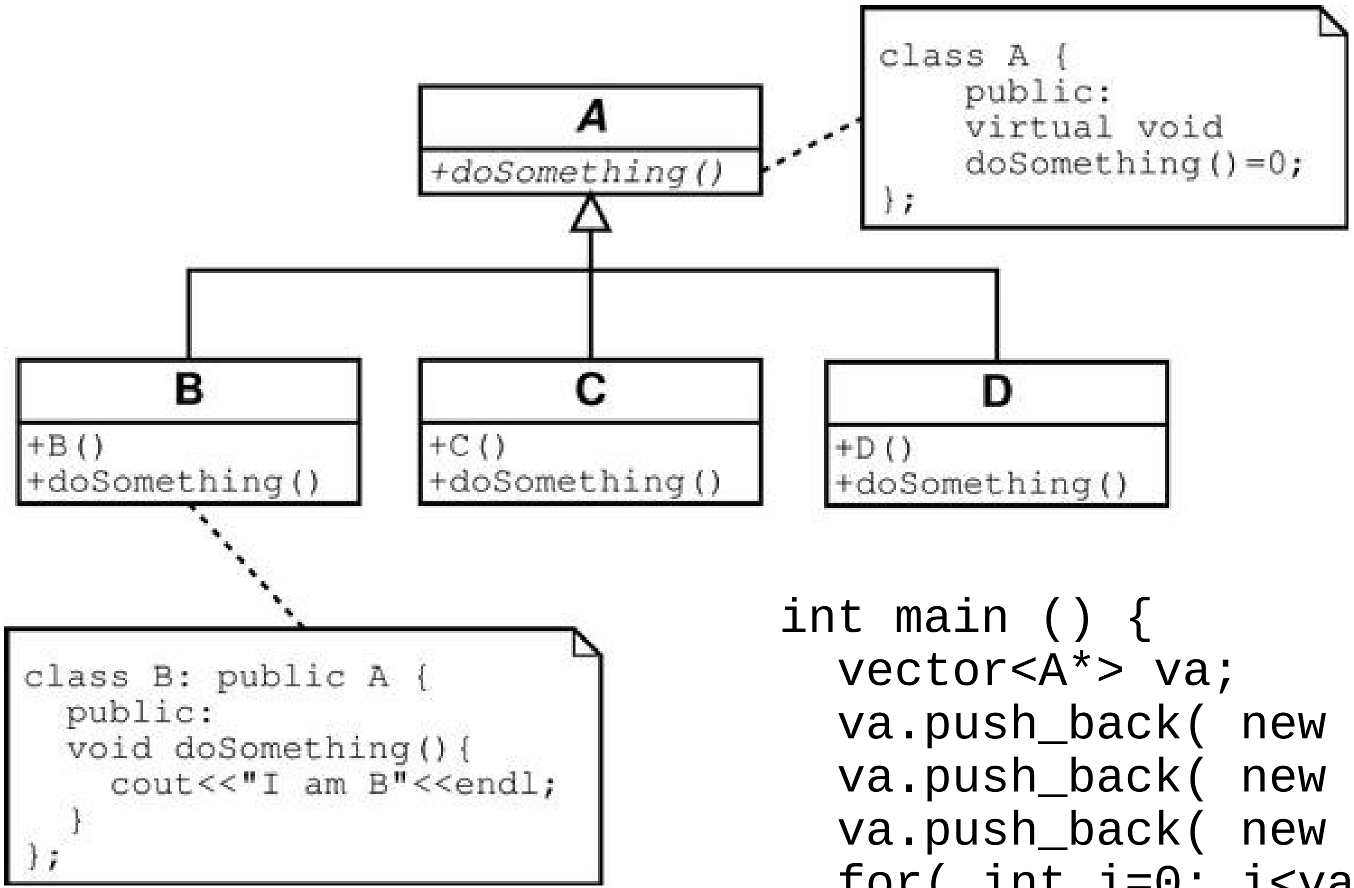
Objects of type A can take many forms, they are polymorph

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)

1.8 Dynamic Polymorphism




```
int main () {
  vector<A*> va;
  va.push_back( new B() );
  va.push_back( new C() );
  va.push_back( new D() );
  for( int i=0; i<va.size(); i++ ){
    va[i]->doSomething();
  }
}
```

1.8 Interface Abstraction

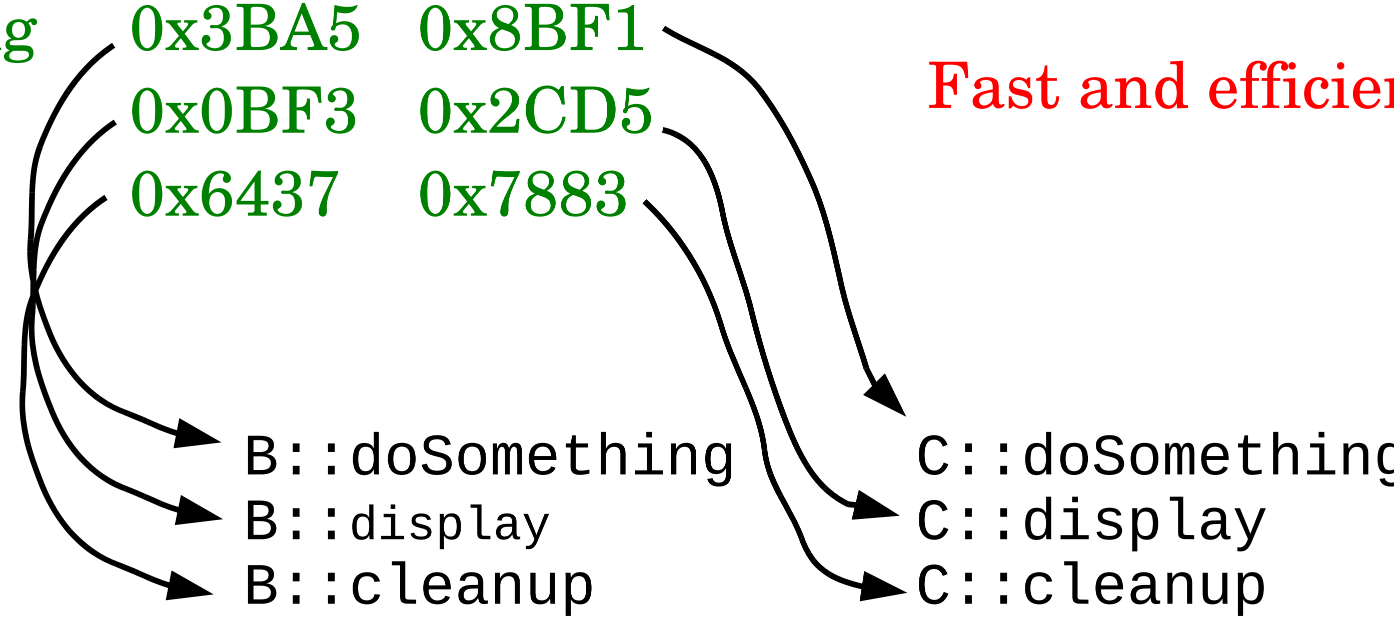
- Common interface of group of objects is an abstraction (abstract class, interface class)
  - find commonality between related objects
  - express commonality formally using interfaces
- Clients (other objects) depend on the abstract interface, not details of objects
  - Polymorphic objects can be substituted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

1.8 Mechanics of Dynamic Polymorphism

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

	A	B	C
1	doSomething	0x3BA5	0x8BF1
2	display	0x0BF3	0x2CD5
3	cleanup	0x6437	0x7883

Fast and efficient!



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

1.8 Static Polymorphism (Templates)

```
Template <class T> class U {
  public:
  void execute() {
    T t;
    t.init();
    t.run();
    t.finsish();
  }
}
```

```
Class B {
  public:
  void init();
  void run();
  void finish();
}
```

Template class U contains generic algorithm

Class B implements

No direct dependence between U and B, but interface must match for U<B> to compile

```
#include “B.hh”
#include “U.hh”
int main {
  U<B> ub;
  ub.execute();
}
```

Can't change types at run-time

Using typed collections difficult

→ don't use static polymorphism unless proven need

1.8 Interfaces Summary

- Interface can be separated from object
  - Abstract (interface) classes
- Find commonality between related objects
  - Abstraction is the key
- Clients depend on abstractions (interfaces), not on specific object details
- Dynamic polymorphism: fast+efficient
  - Static polymorphism (templates)
- Polymorphic objects replace code branches

1.8 Inheritance SA/SD vs OO

SA/SD (procedural):

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

1.8 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
  - class inheritance and object polymorphism
- Not required for OOAD (but useful)
  - templates, lots of convenient operators

1.9 Summary

- Software can be a complex system
  - object and class views
  - hierarchy and abstractions
- Object model
  - Abstraction, encapsulation, modularity, hierarchy, type
  - objects have interface, behaviour, state, identity
- Class inheritance and object polymorphism
  - build hierarchies of abstractions