Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 18/24

Session 18. Introduction to Object-Oriented Programming: Fundamental Concepts

- C++ and Object-orientation
- Inheritance: basic concepts
- Inheritance terminologies: C++ vs. Object-orientation
- Derivation: Classes vs. Structs
- Accessing base classes' members
- Object construction and destruction
- Multiple inheritance
- Q&A

150 min (incl. Q & A)

Conventional definition

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

• Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

- Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

C++ general rule:

General rule

C++ is a language not a complete system.

Conventional definition

- Many C++ design decisions have their roots in my dislike for forcing people to do things in some particular way.

  - Bjarne Stroustrup

1997-2010

C++ is a *general-purpose programming language* with a bias towards *systems programming* that

- is a better C → Macros, structures & functions
- supports data abstraction → Classes
- supports object-oriented programming → Inheritance & Polymorphism
- supports generic programming → Templates

C++ general rule:

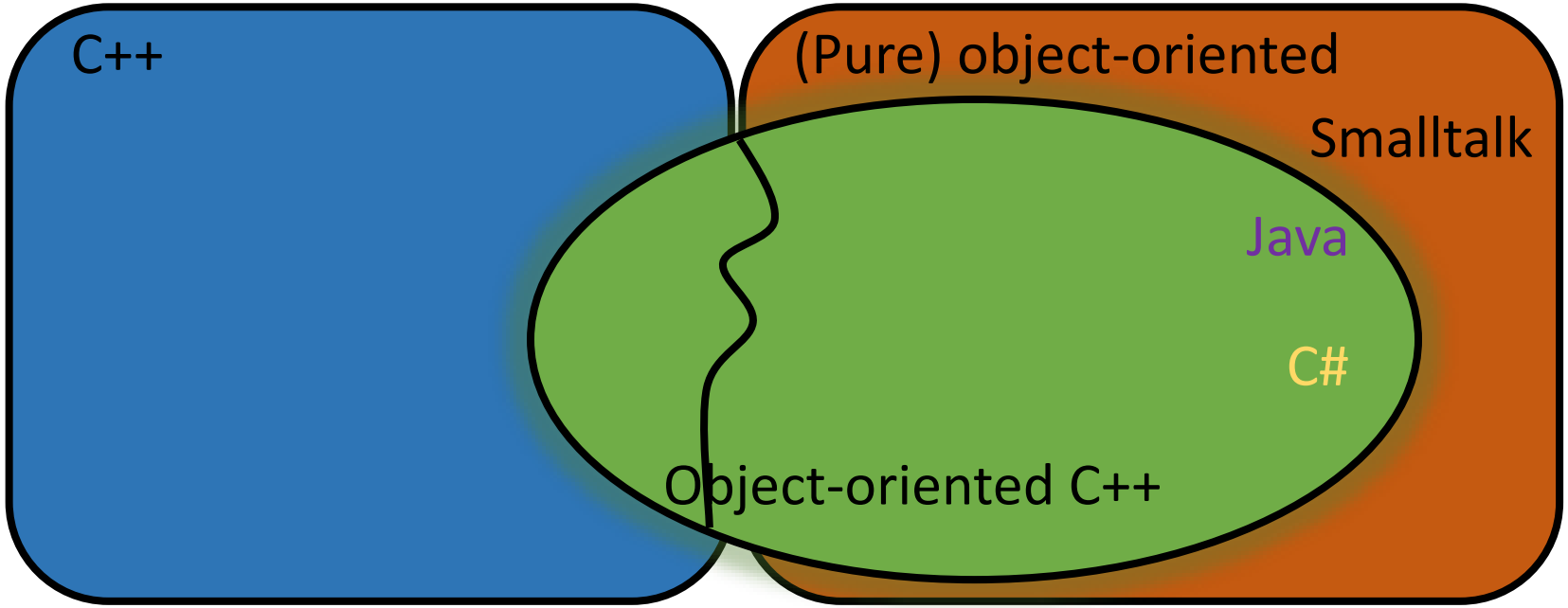
General rule

> C++ is a language not a complete system.

- C++ is a multi-paradigm/multi-style programming language.
- It's old, but still very useful definition.

C++ and Object-orientation

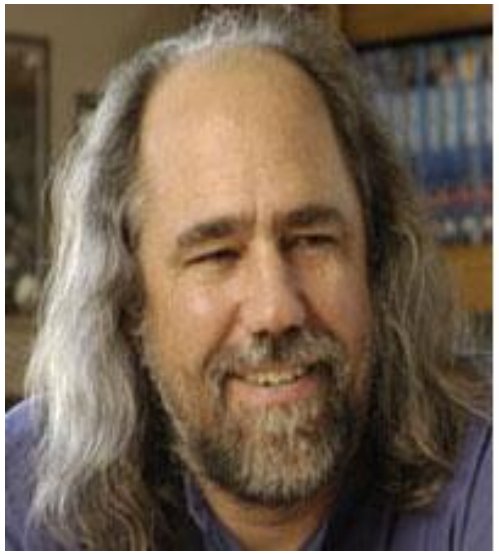
- C++ is a multi-style/paradigm language.



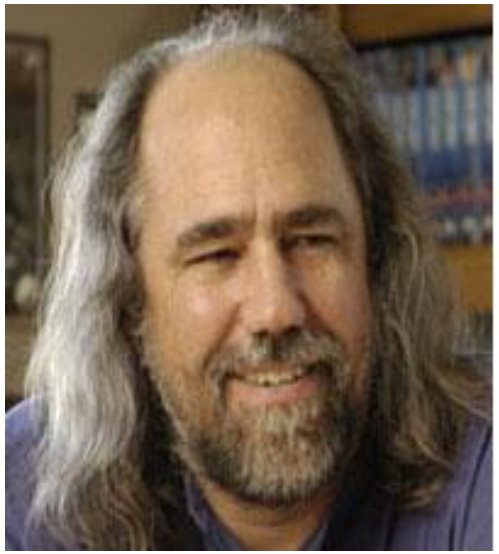
- The concepts and principles of class design from the point of pure object-oriented is beyond the scope of this course.

complex systems are Hierarchical

complex systems are Hierarchical

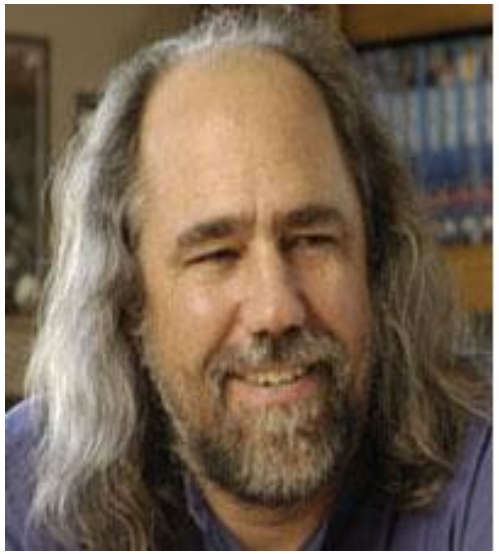


complex systems are Hierarchical



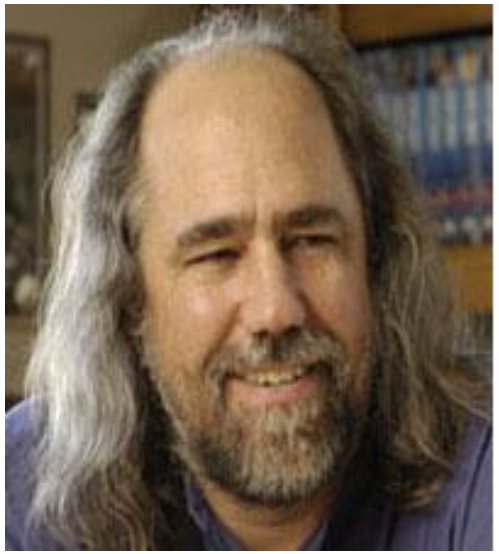
- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

complex systems are Hierarchical



- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

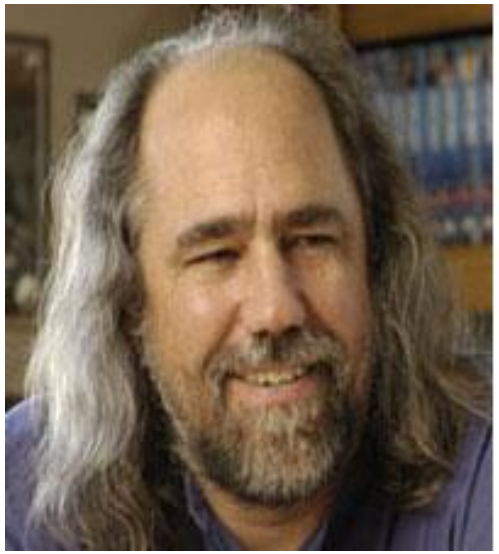
complex systems are Hierarchical



- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

- Complex systems decomposition:
  - Object structure → Part-of hierarchy → Has-a relationship
  - Class structure → Kind-of hierarchy → Is-a relationship

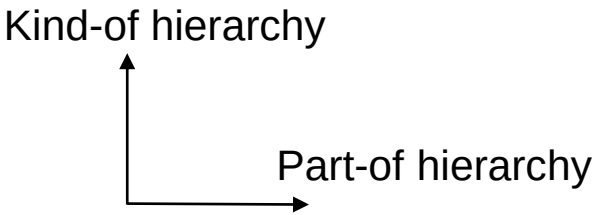
complex systems are Hierarchical



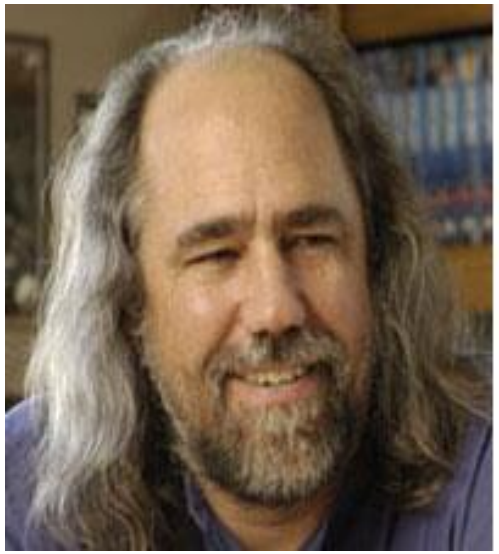
- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

- Complex systems decomposition:
  - Object structure → Part-of hierarchy → Has-a relationship
  - Class structure → Kind-of hierarchy → Is-a relationship

- These kind of relationships are *orthogonal*.

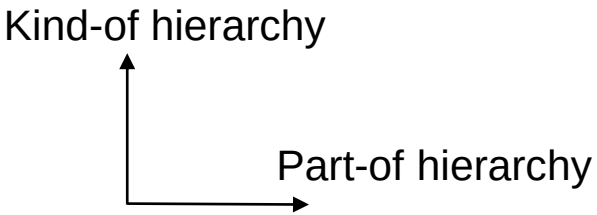


complex systems are Hierarchical



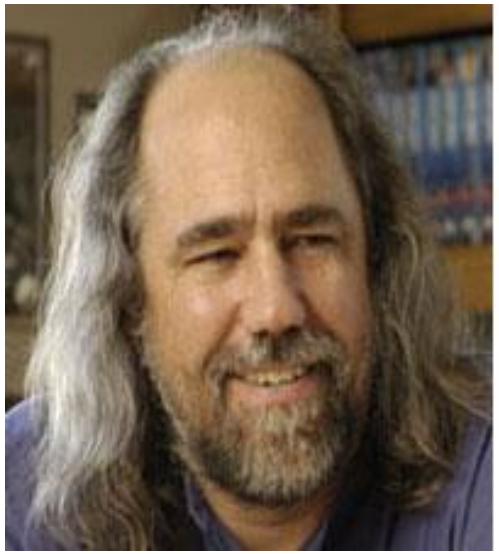
- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

- Complex systems decomposition:
  - Object structure → Part-of hierarchy → Has-a relationship
  - Class structure → Kind-of hierarchy → Is-a relationship



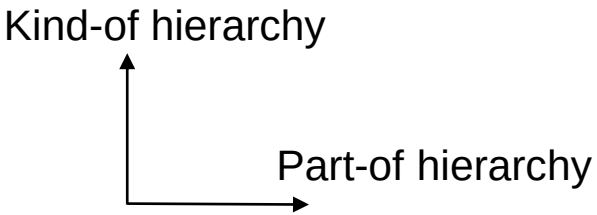
- These kind of relationships are *orthogonal*.
- Another point of view: Human beings abstract things on two dimensions: Part-of and Kind-of

complex systems are Hierarchical



- The five attributes of a complex systems:
  - Hierarchical structure
  - Relative primitives
  - Separation of concerns
  - common patterns
  - Stable intermediate forms

- Complex systems decomposition:
  - Object structure → Part-of hierarchy → Has-a relationship
  - Class structure → Kind-of hierarchy → Is-a relationship



- These kind of relationships are *orthogonal*.
- Another point of view: Human beings abstract things on two dimensions: Part-of and Kind-of
- Yet another point of view: A concept (idea, notion, etc.) does not exist in isolation.

Class relationship

Class relationship

- A concept does not exist in isolation.

Class relationship

- A concept does not exist in isolation.
  - The *car* example.

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition

  - Aggregation

  - Inheritance

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition
  - Aggregation
  - Inheritance

  → Has-a relationship → Concrete classes

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition
  - Aggregation

  → Has-a relationship → Concrete classes

  - Inheritance → Commonality → Derived classes

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition
  - Aggregation

  → Has-a relationship → Concrete classes

  - Inheritance → Commonality → Derived classes

- Hierarchy:

  - Part-of: Object Structure
  - Is-a: Class Structure

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition
  - Aggregation

  → Has-a relationship → Concrete classes

  - Inheritance → Commonality → Derived classes

- Hierarchy:

  - Part-of: Object Structure
  - Is-a: Class Structure

  - Car has-a engine.
  - Car is a kind-of vehicle.

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:

  - Composition
  - Aggregation

  → Has-a relationship → Concrete classes

  - Inheritance → Commonality → Derived classes

- Hierarchy:

  - Part-of: Object Structure
  - Is-a: Class Structure

  - Car has-a engine.
  - Car is a kind-of vehicle.

- `Rectangle`, `Circle`, `Shape`

Class relationship

- A concept does not exist in isolation.
  - The *car* example.
- Class relationships:
  - Composition
  - Aggregation

  → Has-a relationship → Concrete classes

  - Inheritance → Commonality → Derived classes
- Hierarchy:
  - Part-of: Object Structure
  - Is-a: Class Structure

  - Car has-a engine.
  - Car is a kind-of vehicle.
- **`Rectangle`, `Circle`, `Shape`**

Q Is inheritance important to C++?

A Yes. Inheritance is what separates abstract data type (ADT) programming from OO programming.

the Has-a relationship

the Has-a relationship

- Mostly concrete classes
- Mostly programming using data abstraction → Object-based programming

the Has-a relationship

- Mostly concrete classes
- Mostly programming using data abstraction → Object-based programming

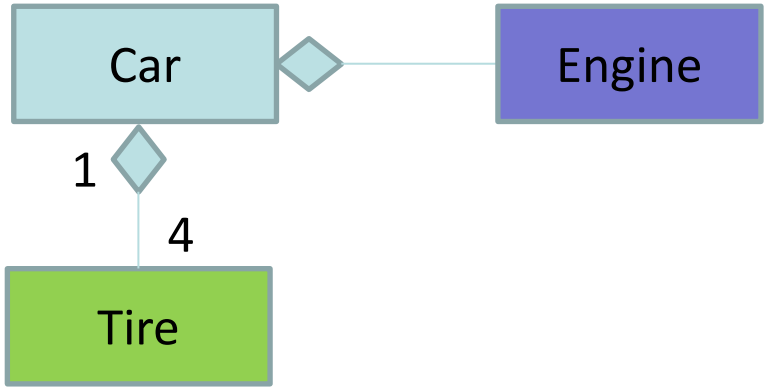
```
class Engine {
    // …
};

class Tire {
    // …
};

class Car { // composite object
    Engine engine; // a car has a engine
    array<Tire, 4> tire; // a car typically has 4 tires
    // …
};
```

the Has-a relationship

- Mostly concrete classes
- Mostly programming using data abstraction → Object-based programming

```
class Engine {
    // …
};

class Tire {
    // …
};

class Car { // composite object
    Engine engine; // a car has a engine
    array<Tire, 4> tire; // a car typically has 4 tires
    // …
};
```

- The UML class/object diagram

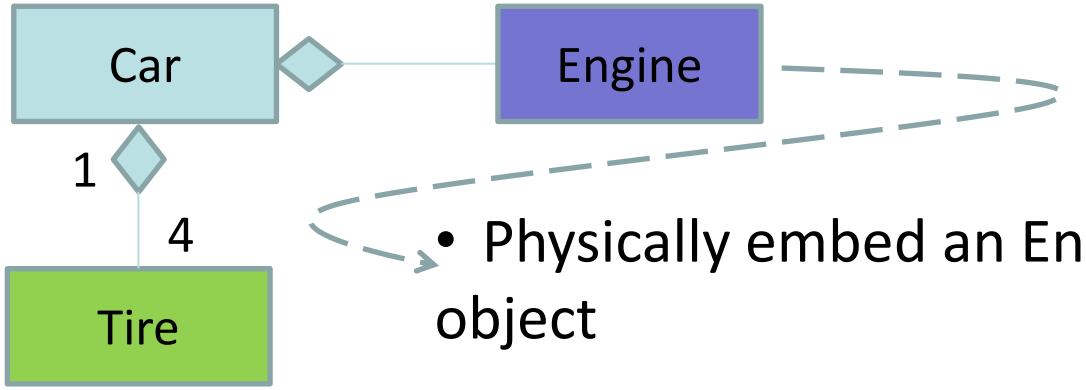


the Has-a relationship

- Mostly concrete classes
- Mostly programming using data abstraction → Object-based programming

```
class Engine {
    // …
};

class Tire {
    // …
};

class Car { // composite object
    Engine engine; // a car has a engine
    array<Tire, 4> tire; // a car typically has 4 tires
    // …
};
```

- The UML class/object diagram



- Physically embed an Engine object inside every car object

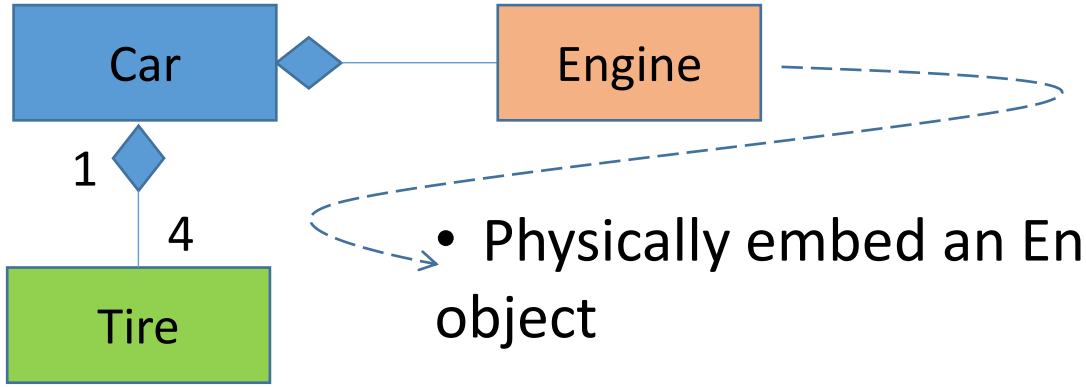
the Has-a relationship

- Mostly concrete classes
- Mostly programming using data abstraction → Object-based programming

```
class Engine {
    // …
};

class Tire {
    // …
};

class Car { // composite object
    Engine engine; // a car has a engine
    array<Tire, 4> tire; // a car typically has 4 tires
    // …
};
```

```
class Date {
    // …
};

class Time {
    // …
};

class Chrono {
    // composite object
    Date date;
    Time time;
    // …
};
```

- The UML class/object diagram



- Physically embed an Engine object inside every car object

Inheritance

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.
- Inheritance fundamental concept: to express commonality between classes.

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.
- Inheritance fundamental concept: to express commonality between classes.
- Commonality via object composition:
- An example: Employee & Manager

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.
- Inheritance fundamental concept: to express commonality between classes.
- Commonality via object composition:
- An example: Employee & Manager

```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager { // manager is an employee
     Employee emp; // manager's employee record
     list<Employee> group;
     short level;
};
```

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.
- Inheritance fundamental concept: to express commonality between classes.
- Commonality via object composition:
- An example: Employee & Manager

```
struct Employee {
    string first_name, last_name;
    Date hiring_date;
    int dept_no;
};

struct Manager { // manager is an employee
    Employee emp; // manager's employee record
    list<Employee> group;
    short level;
};
```

Implicit inheritance

Inheritance

- Inheritance is what separates programming with abstract data type from Object-oriented programming.
- object-based programming vs. Object-oriented programming.
- Inheritance fundamental concept: to express commonality between classes.
- Commonality via object composition:
- An example: Employee & Manager

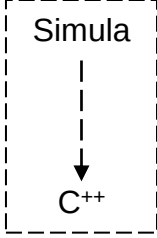
```
struct Employee {
    string first_name, last_name;
    Date hiring_date;
    int dept_no;
};

struct Manager { // manager is an employee
    Employee emp; // manager's employee record
    list<Employee> group;
    short level;
};
```

Implicit inheritance

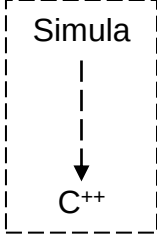
- the Compiler and other tools are unable to deduce Manager is a kind-of Employee.

Inheritance cont.

Inheritance cont.

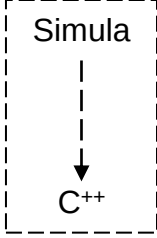


Inheritance cont.




```
struct Employee {
    string first_name, last_name;
    Date hiring_date;
    int dept_no;
};

struct Manager : public Employee { // manager is an employee
    list<Employee> group;
    short level;
};
```

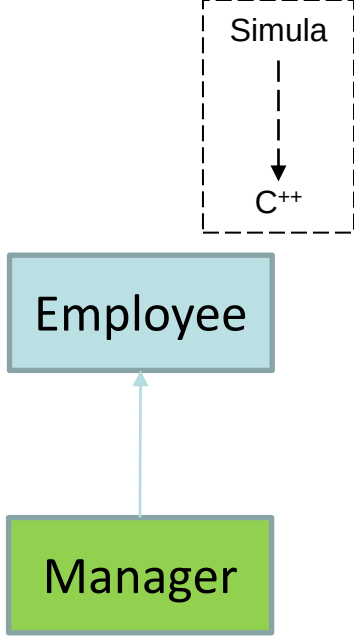
Inheritance cont.




```
struct Employee {
    string first_name, last_name;
    Date hiring_date;
    int dept_no;
};

struct Manager : public Employee { // manager is an employee
    list<Employee> group;
    short level;
};
```

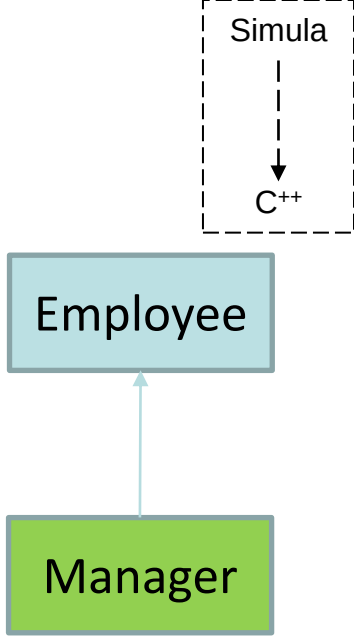
Inheritance cont.

```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```



Inheritance cont.

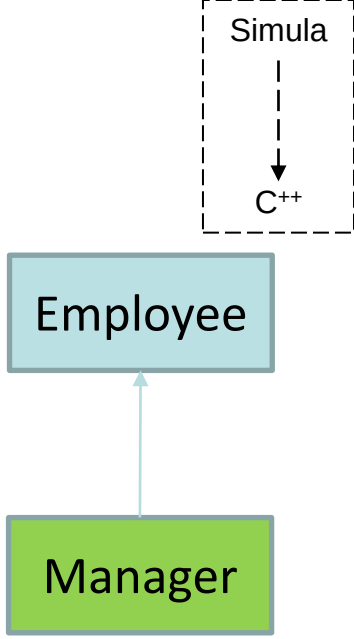
```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```



- Manager is *derived* from Employee.

Inheritance cont.

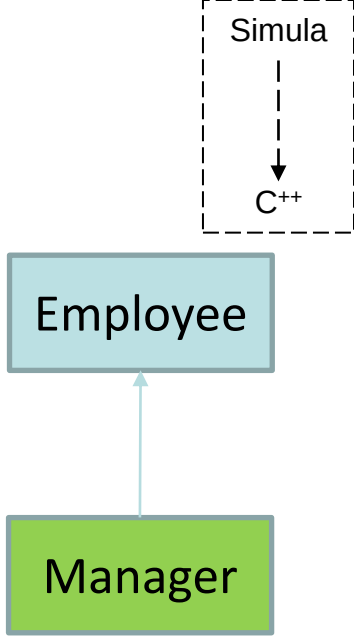
```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```



- Manager is *derived* from Employee.
- Employee is a base class *for* Manager.

Inheritance cont.

```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```

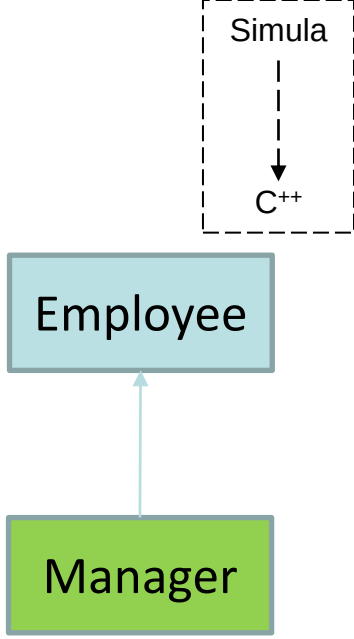


- Manager is *derived* from Employee.
- Employee is a base class *for* Manager.

- Another example: Vehicle and car

Inheritance cont.

```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```

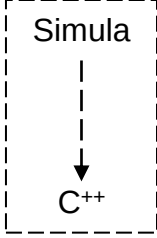


- Manager is *derived* from Employee.
- Employee is a base class *for* Manager.

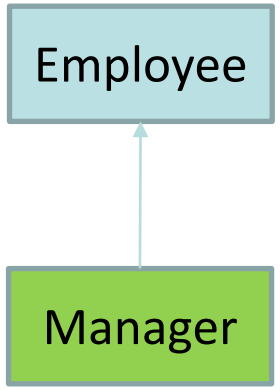
- Another example: Vehicle and car

```
class Vehicle {
  // …
};

class Car : public Vehicle {
  // …
};
```

Inheritance cont.



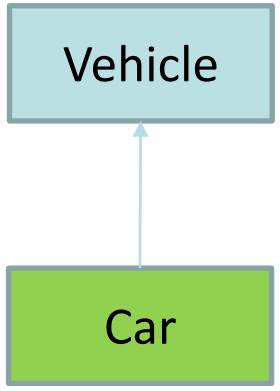

```
struct Employee {
     string first_name, last_name;
     Date hiring_date;
     int dept_no;
};

struct Manager : public Employee { // manager is an employee
     list<Employee> group;
     short level;
};
```



- Manager is *derived* from Employee.
- Employee is a base class *for* Manager.

- Another example: Vehicle and car

```
class Vehicle {
  // …
};

class Car : public Vehicle {
  // …
};
```



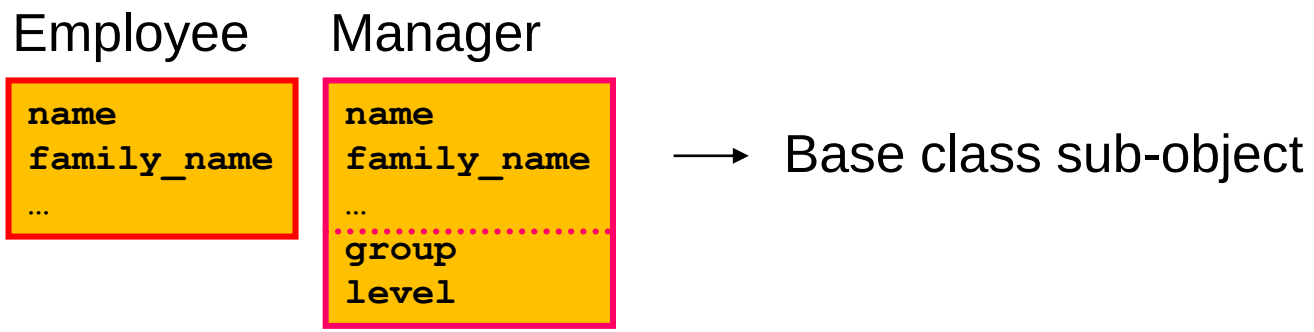
Derivation implementation: under the hood

Derivation implementation: under the hood

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

Derivation implementation: under the hood

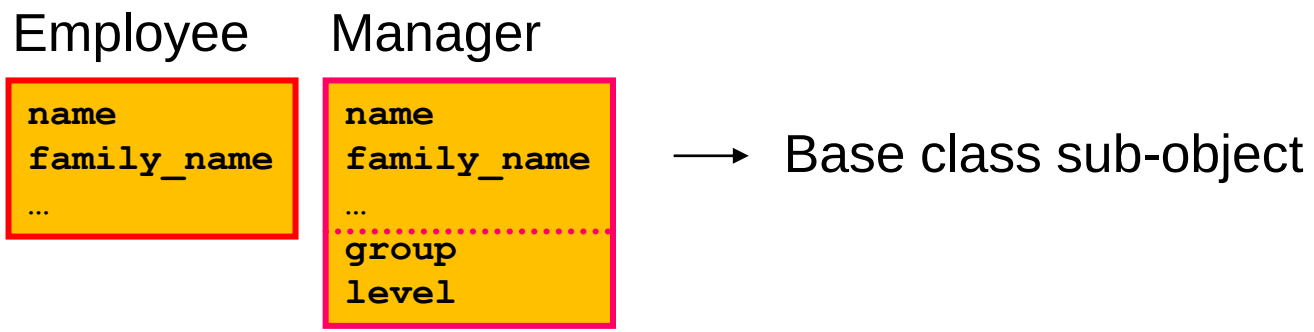
- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.



Derivation implementation: under the hood

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

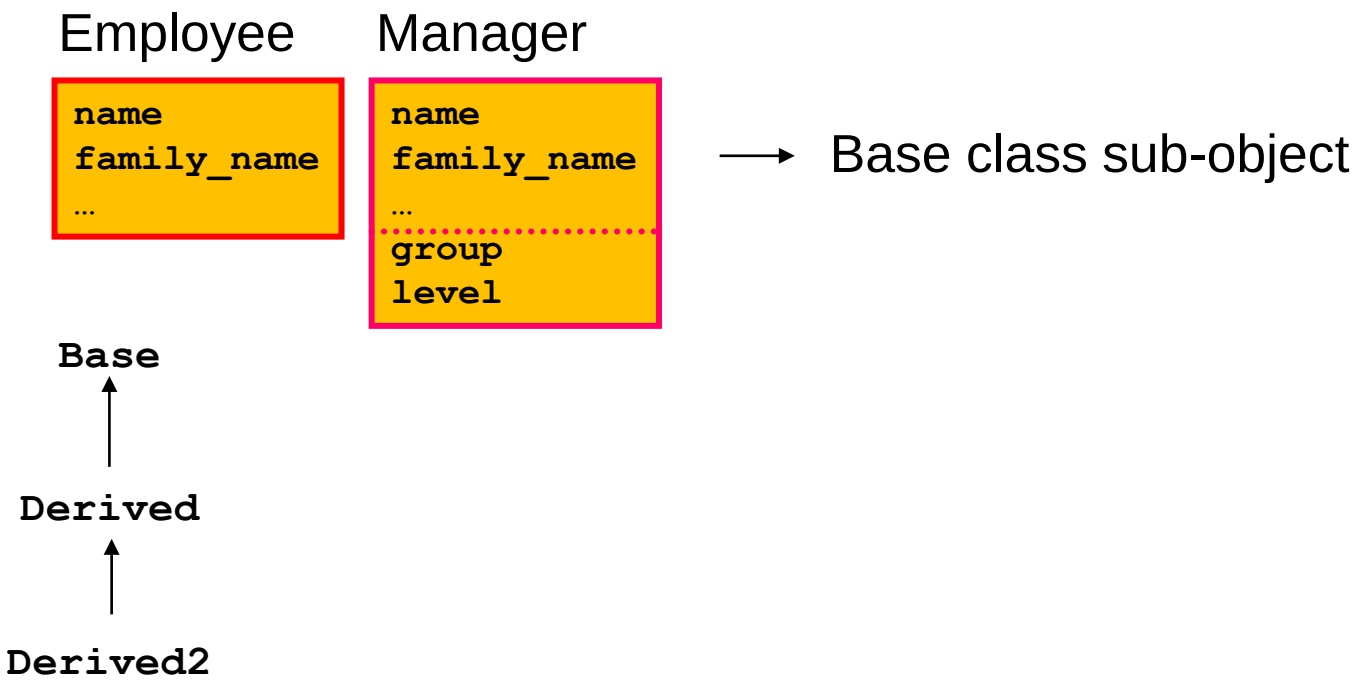
```
class Base {
private:
  int a, b, c;
};
class Derived : public Base {
public:
  int b;
};
class Derived2 : public Derived {
private:
  int c;
};
```



Derivation implementation: under the hood

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

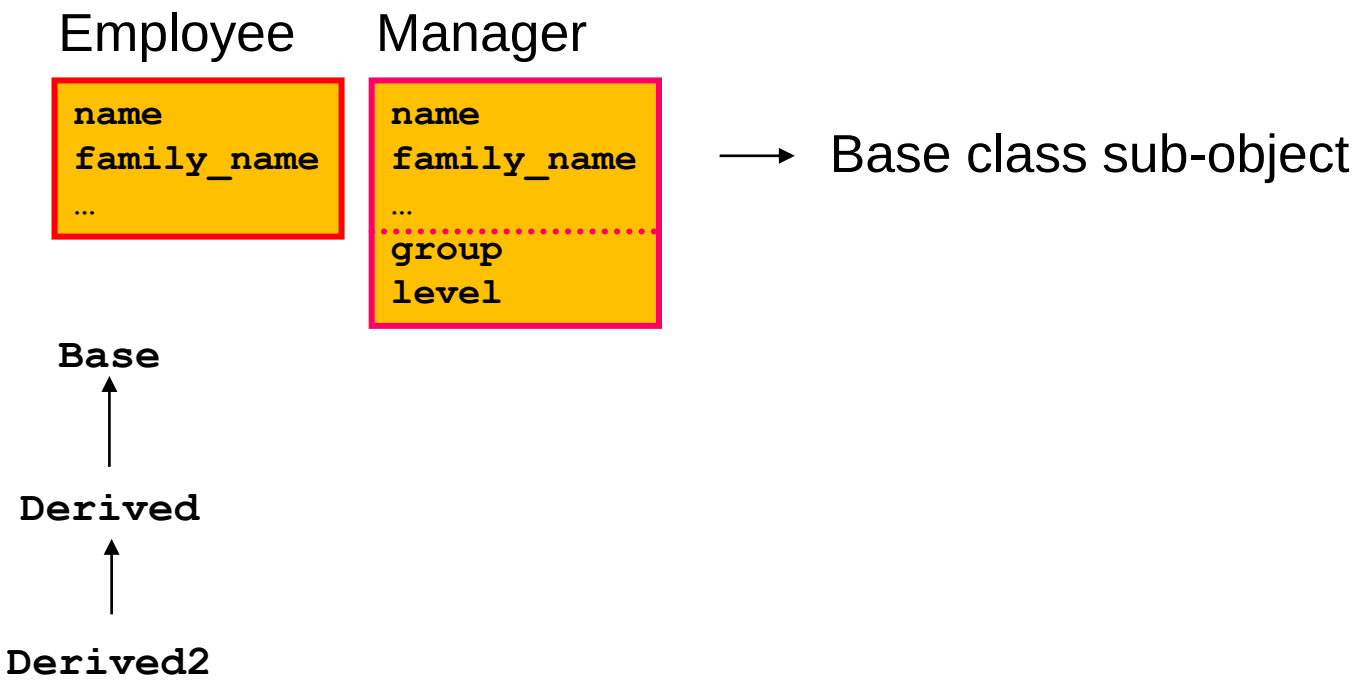
```
class Base {
private:
  int a, b, c;
};
class Derived : public Base {
public:
  int b;
};
class Derived2 : public Derived {
private:
  int c;
};
```



Derivation implementation: under the hood

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

```
class Base {
private:
  int a, b, c;
};
class Derived : public Base {
public:
  int b;
};
class Derived2 : public Derived {
private:
  int c;
};
```

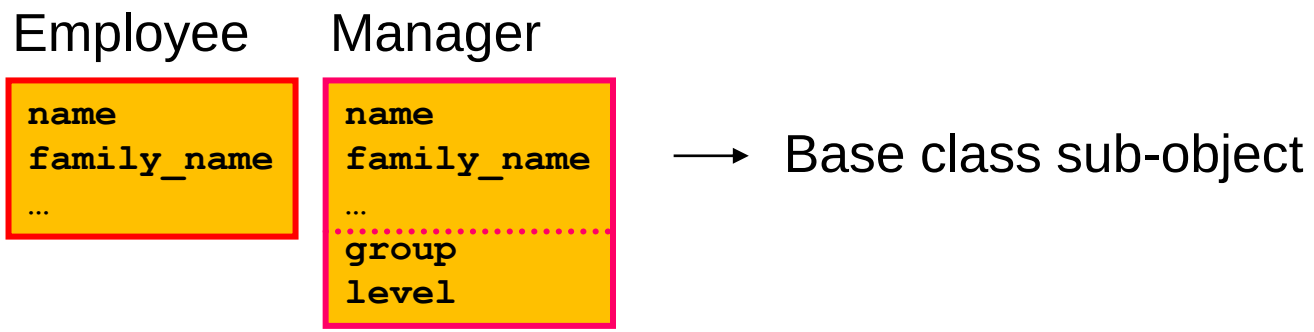


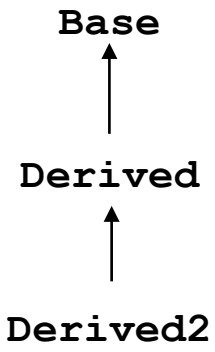
- A derived class and its base class sub-objects can be represented by a directed acyclic graph (DAG) where an arrow means “directly derived from.”

Derivation implementation: under the hood

- A popular and efficient implementation of the notion of derived classes has an object of the derived class represented as an object of the base class, with the information belonging specifically to the derived class added at the end.

```
class Base {
private:
  int a, b, c;
};
class Derived : public Base {
public:
  int b;
};
class Derived2 : public Derived {
private:
  int c;
};
```





- Base is *direct* base class for Derived1.
- Base is *indirect* base class for Derived2.
- Derived1 is derived *directly* from Base.
- Derived2 is derived *indirectly* from Base.

- A derived class and its base class sub-objects can be represented by a directed acyclic graph (DAG) where an arrow means “directly derived from.”
- Some terminology: Direct base class, Indirect base class, Derived class

Subobjects and DAG

Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

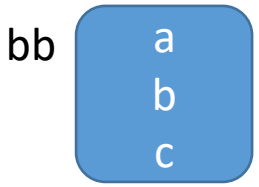
Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```

Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

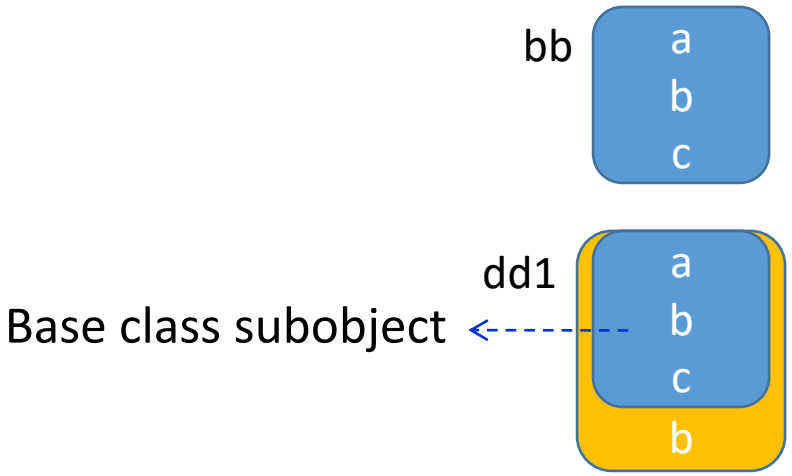



```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```

Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

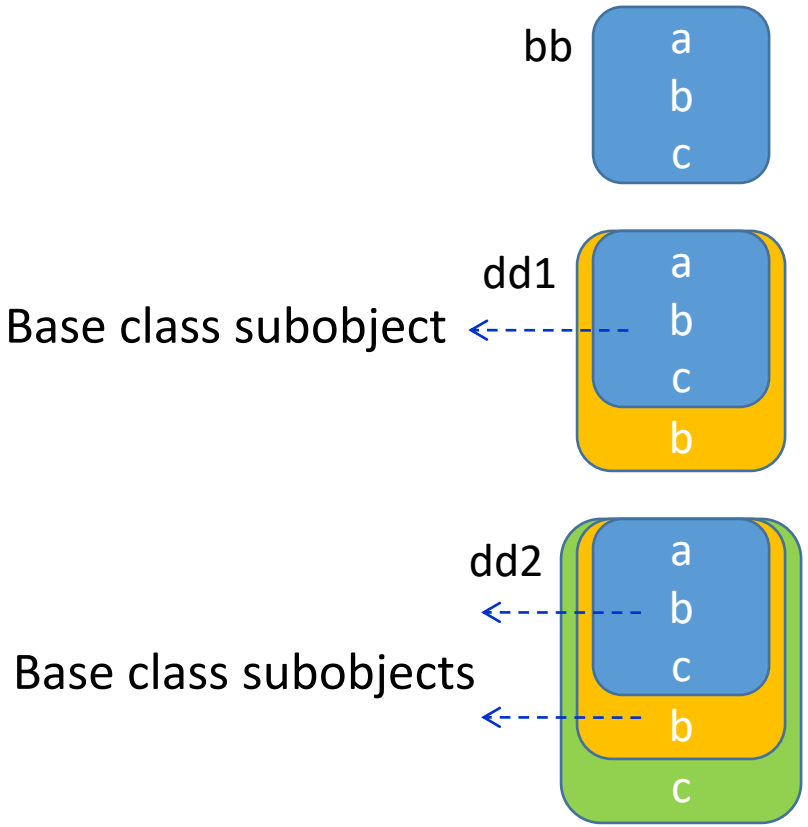
```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```



Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

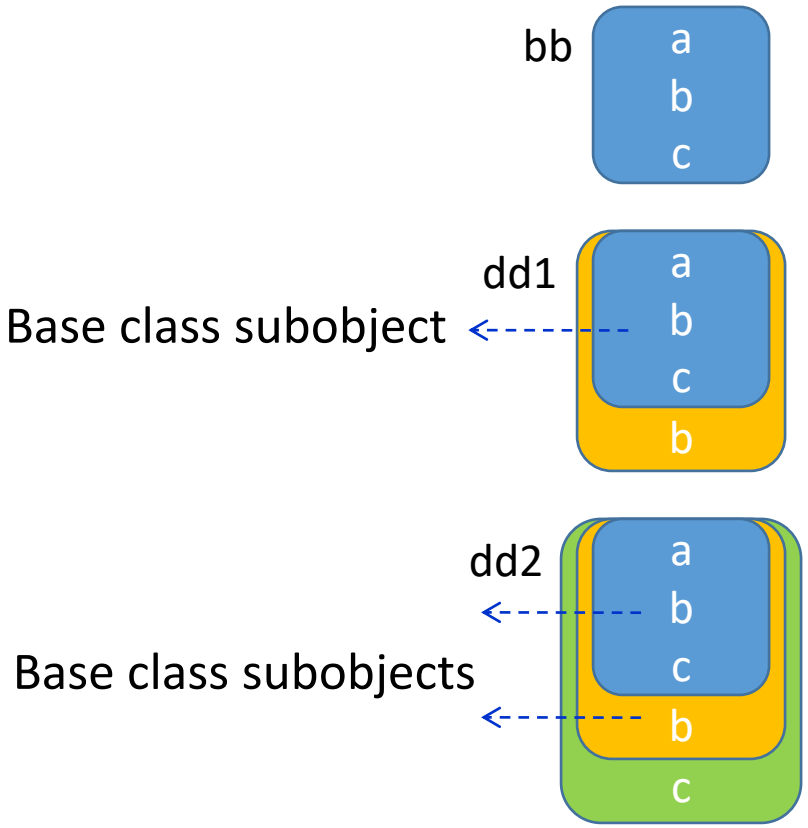
```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```



Subobjects and DAG

- Objects can contain other objects, called *subobjects*.

```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```

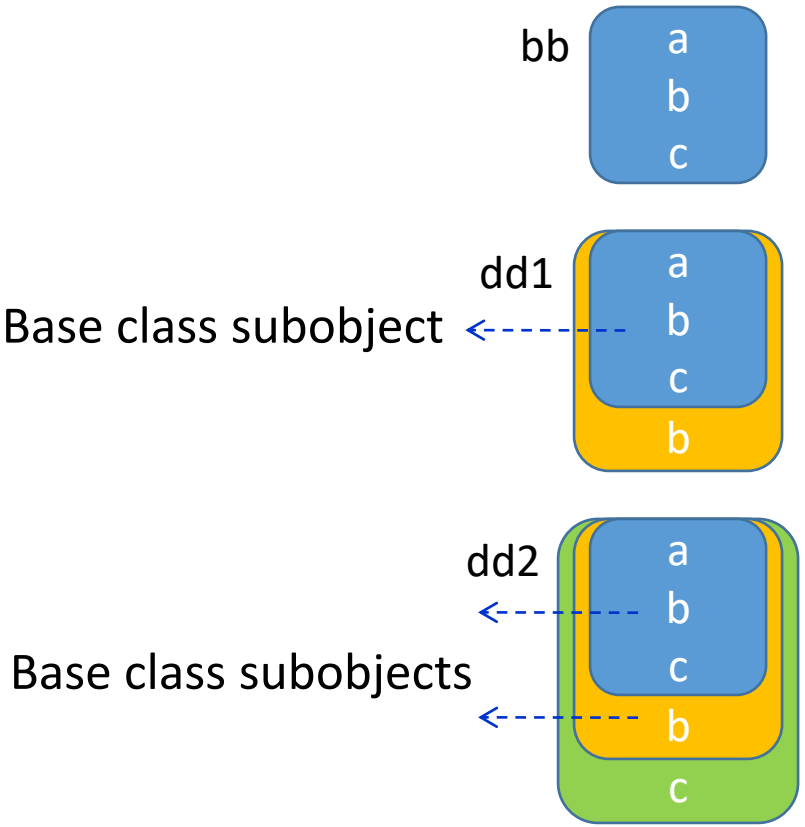


- The order in which the base class subobjects are allocated in the most derived object is *unspecified*.

Subobjects and DAG

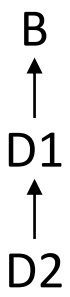
- Objects can contain other objects, called *subobjects*.

```
struct B {
  int a, b, c;
} bb;

struct D1 : B {
  int b;
} dd1;

struct D2 : D1 {
  int c;
} dd2;
```



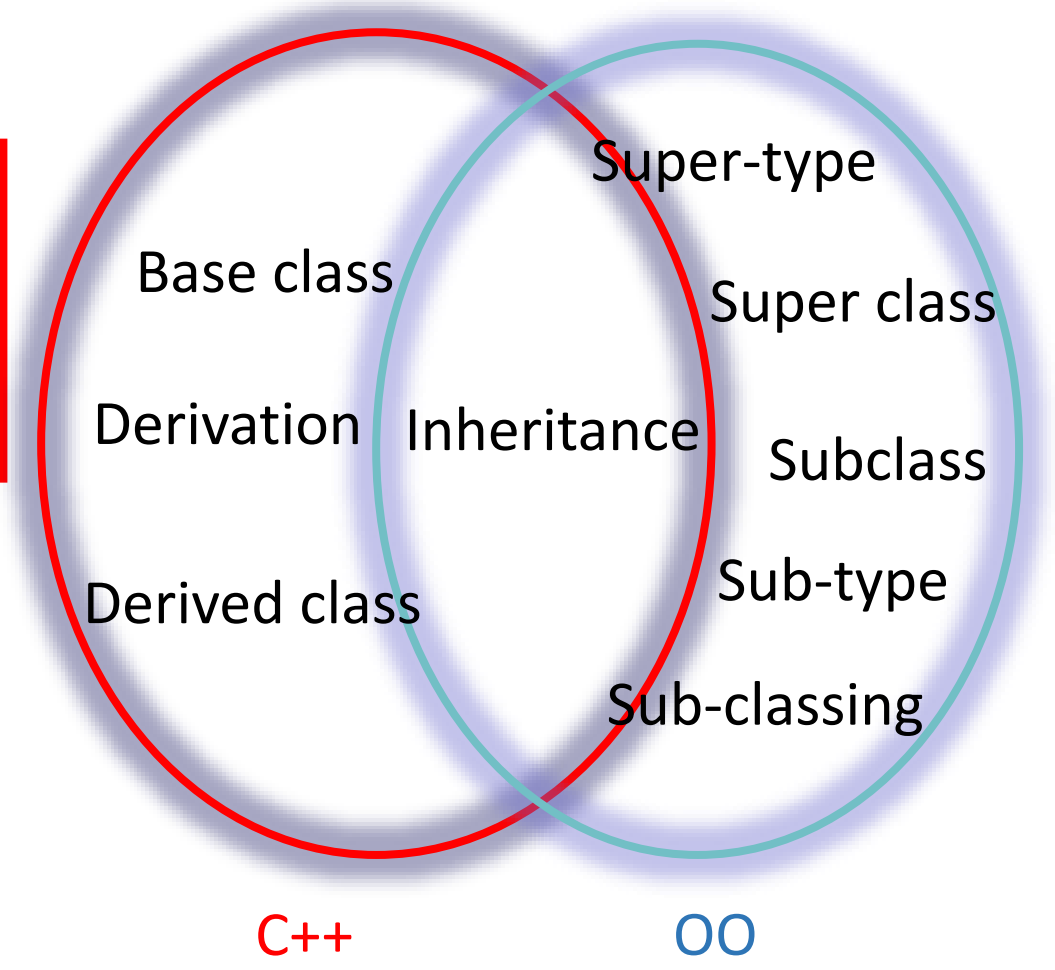
- The order in which the base class subobjects are allocated in the most derived object is *unspecified*.

- A derived class and its base class sub-objects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from."



Inheritance- terminology

```
class Vehicle {
  // …
};

class Car :public Vehicle {
  // …
};
```



- Car is *"a kind of a"* Vehicle.
- Car is *"derived from"* Vehicle.
- Car is *"a specialized"* Vehicle.
- Car is a *"derived class"* of Vehicle.
- Vehicle is the *"base class"* of Car.
- Vehicle members "are inherited" by Car.
- Car is the *"sub-class"* of Vehicle.
- Vehicle is the *"super-class"* of Car.

- Using superclass and subclass terminology is somehow confusing, because the data in derived class object is the superset of the data of an object of its base classes. A derived class object is typically larger (and never smaller) than its base class in the sense that it holds more data and provides more functions.

Inheritance: basic usages

- General rule:

> An object of a derived class can be treated as an object of its base class when manipulating through pointers and references.

- A pointer to a derived class can be implicitly converted to a pointer to base class.
- A **Manager** is (also) an **Employee**, so a **Manager*/Manager&** can be used as a **Employee*/Manager&**.
- A **Car** is (also) an **Vehicle**, so a **Car*/Car&** can be used as a **Vehicle*/Vehicle&**.
- An **Employee** is not necessarily a **Manager**, so an **Employee*** cannot be used as a **Manager***.
- A **Vehicle** is not necessarily a **Car**, so a **Vehicle*/Vehicle&** cannot be used as a **Car*/Car&**.

Inheritance: basic usages

1

```
void f(Manager m1, Employee e1)
{
  vector<Employee *> v;
  v.push_front(&m1);
  v.push_front(&e1);
  // ...
}
```

```
void f()
{
  Car c;
  Vehicle& v = c;
}
```

2

```
void g(Manager mm, Employee ee)
{
  Employee* pe = &mm; // ok: every Manager is an Employee
  Manager* pm = &ee; // error: not every Employee is a Manager
  pm->level = 2; // disaster: ee doesn't have a 'level'
  pm = static_cast<Manager*>(pe); // brute force: works because
                                  // pe points to the Manager mm
  pm->level = 2; // fine: pm points to the Manager mm
                 // that has a 'level'
}
```

- a class must be defined in order to be used as a base.

EmployeeAndManagerTest

Prog.

```
class Employee; // declaration only, no definition
class Manager : public Employee { // error: Employee not defined
  // ...
};
```

Accessibility of base class private members

- A member of a derived class has no special permission to access private members of its base class.

Q

Why can't my derived class access private: things from my base class?

A

1. The concept of a private member would be rendered meaningless by allowing a programmer to gain access to the private part of a class simply by deriving a new class from it.

2. To protect you from future changes to the base class. Derived classes do not get access to private members of a base class. This effectively "seals off" the derived class from any changes made to the private members of the base class.

Member functions

1

```
class Employee {
     string first _name, family_name;
     char middle_initial;
     // ...
public:
     void print() const
     {
          cout << full_name() << '\n';
     }
   string full_name() const {
     return first_name + ´ ´ + middle_initial + ´ ´ + family_name ;
   }
   // ...
};
```

2

```
class Manager : public Employee {
   // ...
public:
   void print() const;
   // ...
};
```

3

```
void Manager::print() const
{
   cout << "name is" << full_name() << ´\n´;
   cout << level << '\n';
}
```

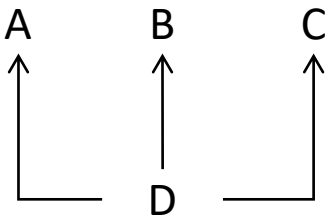
4

```
void Manager::print() const
{
   cout << "name is" << family_name << ´\n´; // error
   // ...
}
```

EmployeeAndManagerTest

Multiple base classes

- A class can be derived from any number of base classes.




```
class A { / … / };
class B { / … / };
class C { / … / };
class D : public A, public B, public C { / … / };
```

```
// without MI (1)
class ABC { / The union of A, B and C members / };
class D : public ABC {/ … / };
```


```
// without MI (2)
class A { / … / };
class B :public A { / … / };
class C :public B { / … / };
class D :public C { / … / };
```

- A class shall not be specified as a direct base class of a derived class more than once.

Exercise

Why?

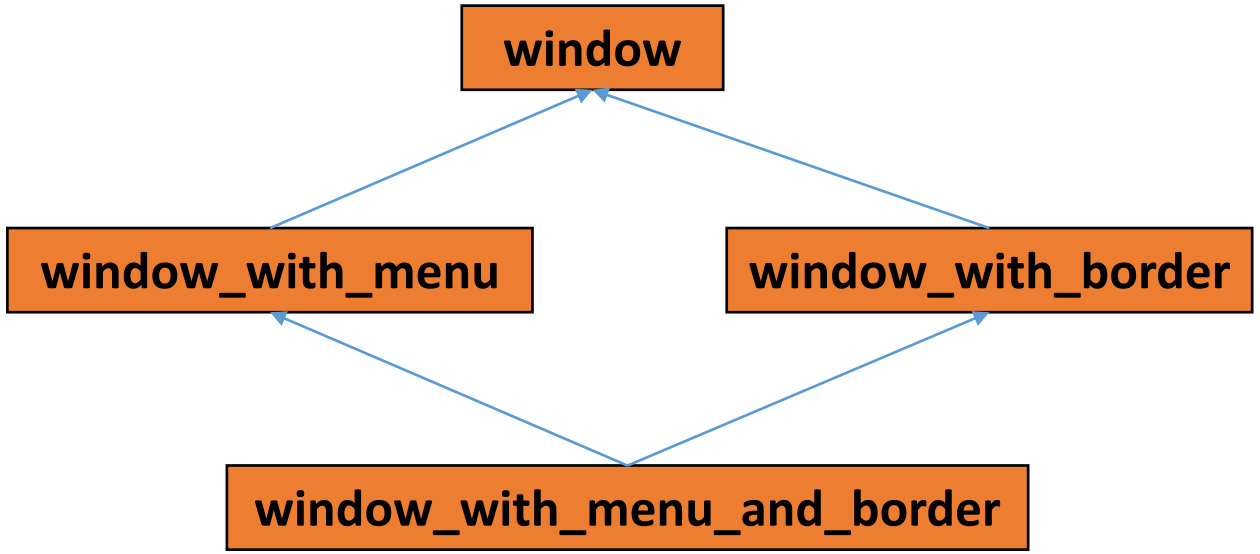
```
class Y : public X, public X { /* … */ };
```

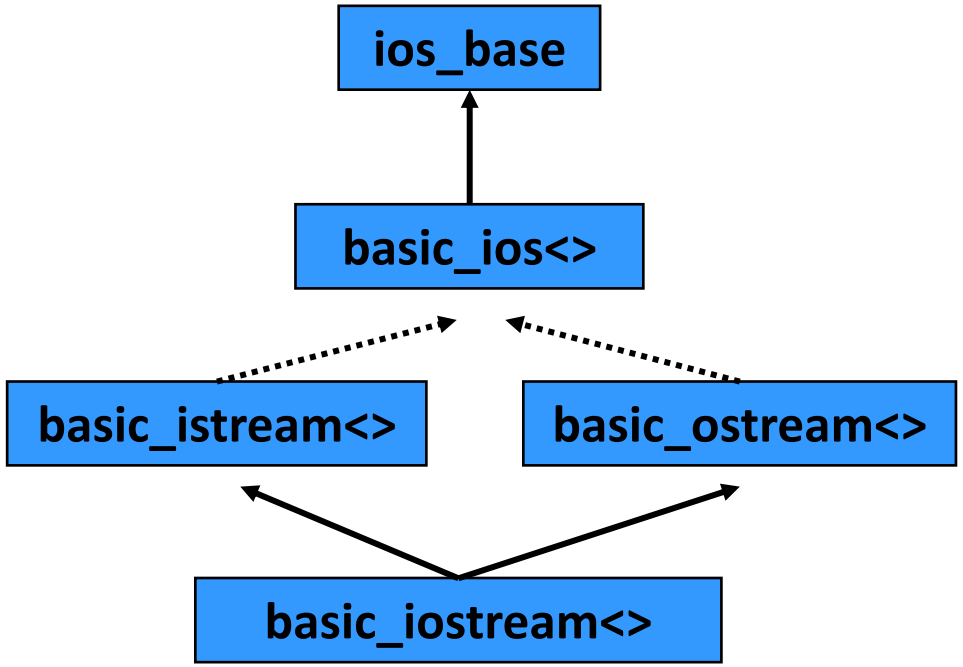
Multiple inheritance

Because you have a mother and a father :)

- comp.lang.c++

- the use of more than one immediate base class for a derived class.
- A typical windowing system



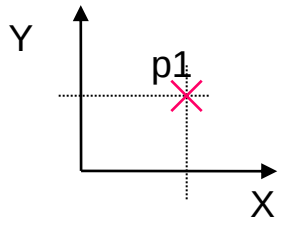


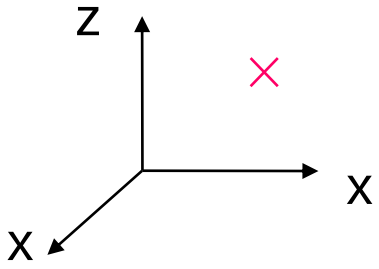
- C++ standard library: I/O stream class hierarchy
- Dotted lines mean virtual base classes
- One typical use is to have one base define an interface and another providing help for the implementation.

2D and 3D points

1

```
class Point {
public:
  Point() : x(0), y(0) {}
  Point(int x_, int y_) : x(x_), y(y_) {}
  int X() const;
  int Y() const;
private:
  int x, y;
};
```





2

```
#include "Point.h"
class Point3D : public Point {
public:
  Point3D() : Point(), z(0) {}
  Point3D(int x_, int y_, int z_) :
                Point(x_,y_), z(z_) {}
private:
  int z;
};
```

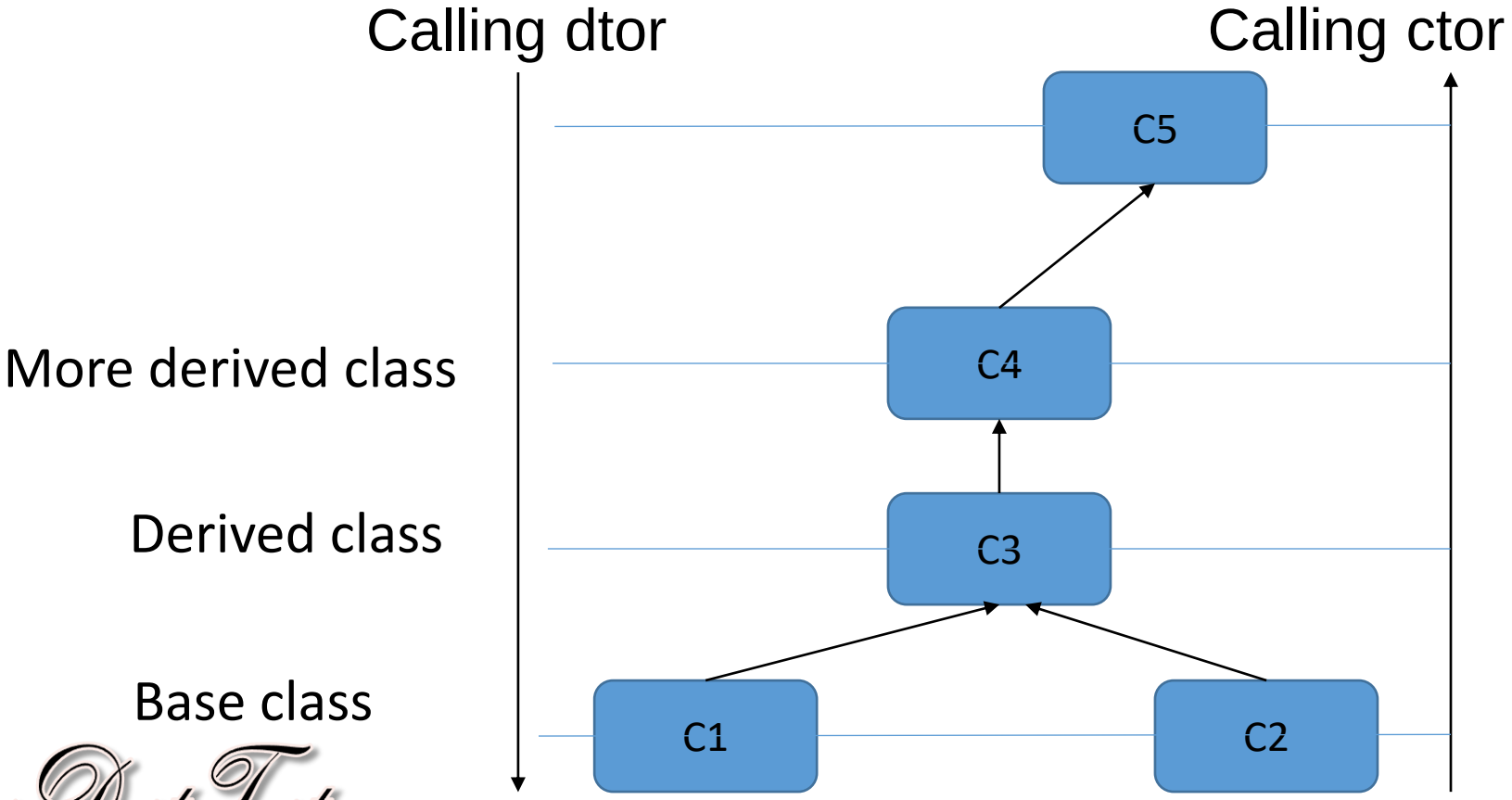
3

```
// use Point & Point3D
void f()
{
  Point3D p1(1, 2, 3);
  Point* p2 = new Point();
  int y = p2->Y();
  delete p2;
}
```

Object construction and destruction

- Objects are constructed from the *bottom up*: first the base, then the members, and then the derived class itself. They are destroyed in the *top down*: first the derived class itself, then the members, and then the base. Members and bases are constructed in order of declaration in the class and destroyed in the reverse order.

```
class C1 {
    // …
};

class C2 {
    // …
};

class C3 :public C1, :public C2 {
    // …
};
class C4 :public C3 {
    // …
};
class C5 :public C4 {
    // …
};
```



Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant