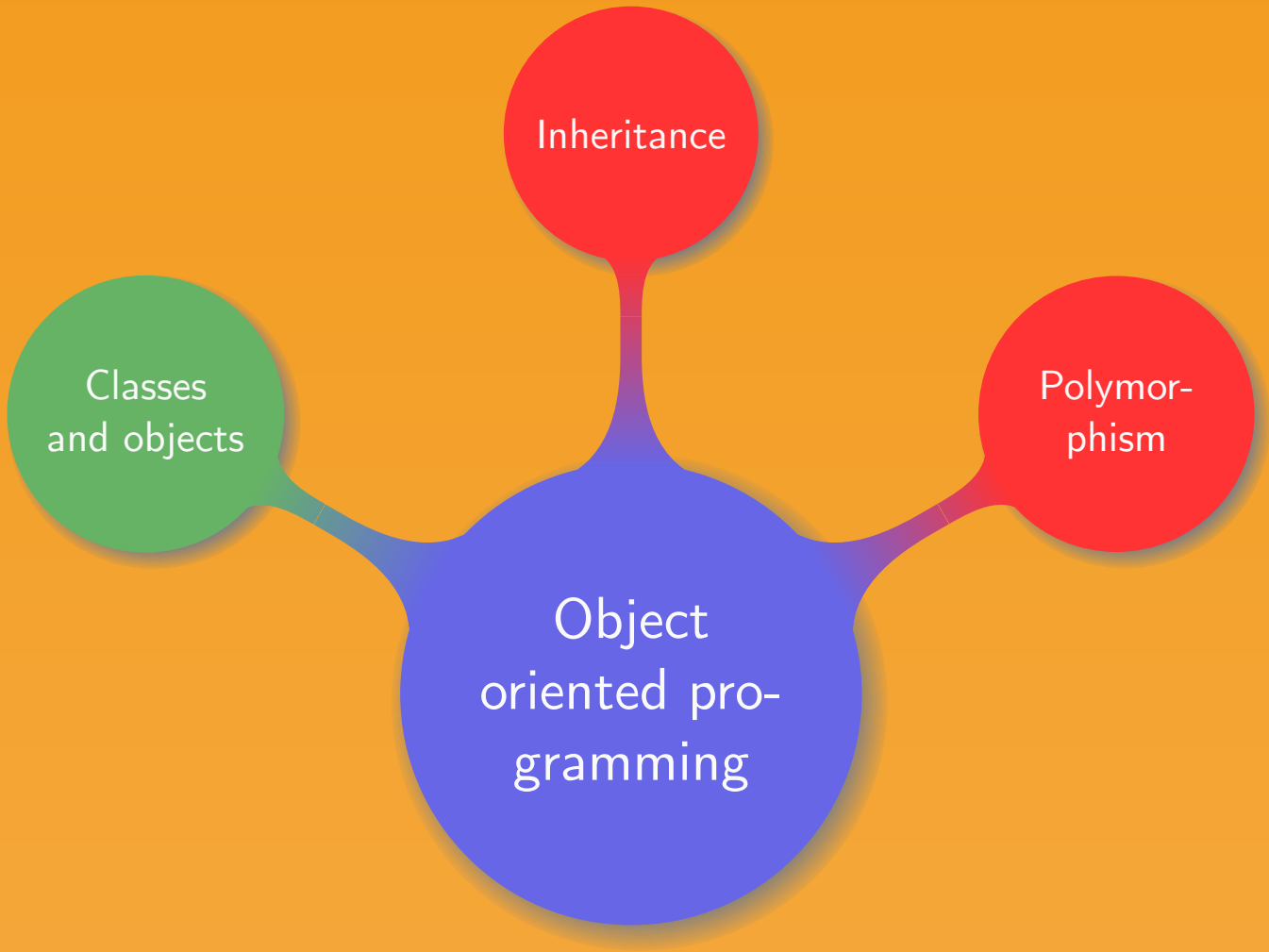
Introduction to Computer and Programming

10. Object oriented programming

Manuel – Summer 2019

Chapter organisation



Procedural programming

Programming approach used so far:

- Program written as a sequence of procedures
- Each procedure fulfills a specific task
- All tasks together compose a whole project
- Further from human thinking
- Requires higher abstraction

Object oriented programming

A new approach:

- Everything is an object
- Objects communicate between them by sending messages
- Each object has its own type
- Object of a same type can receive the same message

Object

An object has two main components:

- Its behavior, what can be done with it, its *methods*
- The data it contains, what it knows, its *attributes*

Object

An object has two main components:

- Its behavior, what can be done with it, its *methods*
- The data it contains, what it knows, its *attributes*

Example.
Given a simple TV:

- Methods:
  - High level actions, e.g. on-off, channel, volume
  - Low level actions, e.g. on internal electronics components
- Attributes:
  - High level elements:, e.g. button on the remote control
  - Low level elements, e.g. internal electronics components

Class and instance

Class:

- Defines the family, type or nature of an object
- Equivalent of the type in “traditional programming”

Instance:

- Realisation of an object from a given class
- Equivalent of a variable in “traditional programming”

Example.
Two same TV models can be represented as two instances of one class

Class specification

Oder of definition:

1. Define the methods
2. Define the attributes

Class specification

Oder of definition:

1. Define the methods
2. Define the attributes

Example.
Create an object `circle`:

1. What it can do, i.e. the methods:
   - `move`
   - `zoom`
   - `area`
2. What is needed to achieve it, i.e. its attributes:
   - Position of the center (x, y)
   - Radius of the circle

Class interface

The interface of a class:

- Is equivalent to `header.h` file in C
- Contains the description of the object
- Splits into two main parts
  - Public definition of the class: user methods
  - Private attributes and methods: not accessible to the user but necessary to the "good functioning"

Example.
In the case of a TV:

- Public methods: on/off, change channel, change volume
- Public attributes: remote control and buttons
- Private methods: actions on the internal components
- Private attributes: internal electronics

A note on visibility

Private or public:

- Private members can only be accessed by member functions within the class
- Users can only access public members

Benefits:

- Internal implementation can be easily adjusted without affecting the user's code
- Accessing private attributes is forbidden: more secure

Only render a member public when necessary

Circle interface

Example.

circle_v0.h

```
1  class Circle {
2  /* user methods (and attributes)*/
3    public:
4      void move(float dx, float dy);
5      void zoom(float scale);
6      float area();
7  /* implementation attributes (and methods) */
8    private:
9      float x, y, r;
10 };
```

Circle interface

Example.

circle_v0.h

```
1  class Circle {
2  /* user methods (and attributes)*/
3    public:
4      void move(float dx, float dy);
5      void zoom(float scale);
6      float area();
7  /* implementation attributes (and methods) */
8    private:
9      float x, y, r;
10 };
```

Understanding the code:

- What is defined as private and public?
- If the circle does not move, what attribute are necessary?

Class instantiation

Using the created objects:

- Include the class using the header file
- Declare one or more instances
- Classes similar to structures in C:
  - Structure only contains attributes
  - Class also contains methods
- Calling a method on an object: `instance.method`

Circle instantiation

Example.

circle_main_v0.cpp

```
1  #include <iostream>
2  #include "circle_v0.h"
3  using namespace std;
4  int main () {
5    float s1, s2;
6    Circle circ1, circ2;
7    circ1.move(12,0);
8    s1=circ1.area(); s2=circ2.area();
9    cout << "area: " << s1 << endl;
10   cout << "area: " << s2 << endl;
11   circ1.zoom(2.5); s1=circ1.area();
12   cout << "area: " << s1 << endl;
13 }
```

Circle instantiation

Example.

circle_main_v0.cpp

```
1  #include <iostream>
2  #include "circle_v0.h"
3  using namespace std;
4  int main () {
5    float s1, s2;
6    Circle circ1, circ2;
7    circ1.move(12,0);
8    s1=circ1.area(); s2=circ2.area();
9    cout << "area: " << s1 << endl;
10   cout << "area: " << s2 << endl;
11   circ1.zoom(2.5); s1=circ1.area();
12   cout << "area: " << s1 << endl;
13 }
```

Understanding the code: why is this program not compiling?

Class implementation

Getting things ready:

- Class interface is ready
- Instantiation is possible
- Does not compile: no implementation of the class yet
- Syntax: `classname::methodname`

Class implementation

Example.

circle_v0.cpp

```
1  #include "circle_v0.h"
2  static const float PI=3.1415926535;
3  void Circle::move(float dx, float dy) {
4    x += dx;
5    y += dy;
6  }
7  void Circle::zoom(float scale) {
8    r *= scale;
9  }
10 float Circle::area() {
11   return PI * r * r;
12 }
```

Class implementation

Example.

circle_v0.cpp

```
1  #include "circle_v0.h"
2  static const float PI=3.1415926535;
3  void Circle::move(float dx, float dy) {
4    x += dx;
5    y += dy;
6  }
7  void Circle::zoom(float scale) {
8    r *= scale;
9  }
10 float Circle::area() {
11   return PI * r * r;
12 }
```

Understanding the code: can this file be compiled alone?

Constructor and destructor

Automatic construction and destruction of objects:

- Object not initialised by default (same as `int i`)
- Constructor: method that initialises an instance of an object
- Used for a proper default initialisation
- Definition: no type, name must be `classname`
- Important note: can have more than one constructor
- Destructor: called just before the object is destroyed
- Used for clean up (e.g. release memory, close a file etc...)
- Definition: no type, name must be `~classname`

Class interface

Example.

circle_v1.h

```
1  class Circle {
2  /* user methods (and attributes)*/
3    public:
4      Circle();
5      Circle(float r);
6      ~Circle();
7      void move(float dx, float dy);
8      void zoom(float scale);
9      float area();
10 /* implementation attributes (and methods) */
11   private:
12     float x, y;
13     float r;
14 };
```

Class implementation

circle_v1.cpp

```
1  #include "circle_v1.h"
2  static const float PI=3.1415926535;
3  Circle::Circle() {
4    x=y=0.0; r=1.0;
5  }
6  Circle::Circle(float radius) {
7    x=y=0.0; r=radius;
8  }
9  Circle::~Circle() {}
10 void Circle::move(float dx, float dy) {
11   x += dx; y += dy;
12 }
13 void Circle::zoom(float scale) {
14   r *= scale;
15 }
16 float Circle::area() {
17   return PI * r * r;
18 }
```

Class instantiation

circle_main_v1.cpp

```
1  #include <iostream>
2  #include "circle_v1.h"
3  using namespace std;
4  int main () {
5    float s1, s2;
6    Circle circ1, circ2((float)3.1);
7    circ1.move(12,0);
8    s1=circ1.area(); s2=circ2.area();
9    cout << "area: " << s1 << endl;
10   cout << "area: " << s2 << endl;
11   circ1.zoom(2.5);
12 //  cout << circ1.r <<endl;
13   s1=circ1.area();
14   cout << "area: " << s1 << endl;
15 }
```

Overloading

Better definitions:

- Two constructor defined: `circle()` and `circle(float)`
- Proper one automatically selected

Another strategy is to set a default value in the specification.

```
1 Circle(float radius=1.0);
```

Example.
A 2D geometry library is updated to support 3D. As a result the function `move` now takes three arguments: *dx*, *dy*, *dz*. For the old instantiations to remain valid adjust the interface (header file).

```
1 move(float dx, float dy, float dz=0.0);
```

Problem

Exercise.

Write a new main file with two pointers: one for the two circles and one for their areas. The `main` function should not perform any real work.

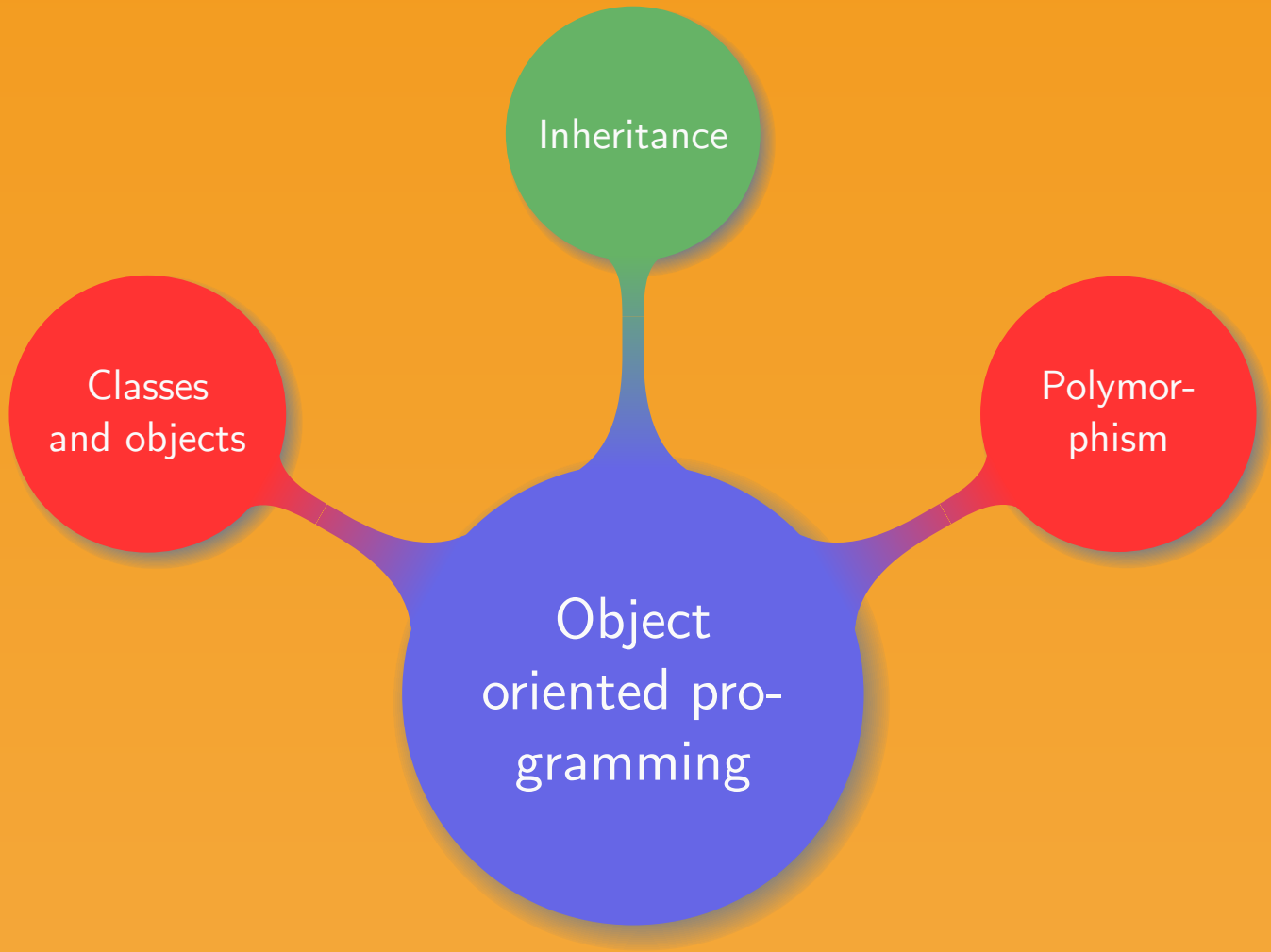
Problem

Exercise.
Write a new main file with two pointers: one for the two circles and one for their areas. The `main` function should not perform any real work.

main_ptr.cpp

```
1  #include <iostream>
2  #include "circle_v1.h"
3  using namespace std;
4  void FctCirc(Circle *circ, float *s) {
5    *(circ+1)=Circle(3.1);
6    *s=circ->area(); s[1]=circ[1].area();
7    cout << "area: " << s[0] << endl;
8    cout << "area: " << *(s+1) << endl;
9    circ[0].zoom(2.5); *s=circ->area();
10   cout << "area: " << s[0] << endl;
11 }
12 int main () {
13   float *s=new float[2]; Circle *circ; circ=new Circle[2];
14   FctCirc(circ,s);
15   delete[] s; delete[] circ; return 0;
16 }
```

Chapter organisation



Why using classes?

Benefits of classes:

- Object are not too abstract
- Closer from the human point of view
- Methods only applied to object which can accept them
- Things are organised in a simple and clear way

Managing a cow

Lets construct a zoo and work with cows...

cows_0.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4    public:
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() {
7        if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8        else cout << "I'm hungry\n";}
9      Cow(int f=0){grass=f;}
10   private: int grass;
11 };
12 int main () {
13   Cow c1(1);
14   c1.Speak(); c1.Eat(); c1.Eat();
15 }
```

Managing a sick cow

A sick cow does:

- Everything a cow does
- Take its medication

Managing a sick cow

A sick cow does:

- Everything a cow does
- Take its medication

Two obvious strategies:

- Add a `TakeMediaction()` method to the cow
- Recopy the `cow` class, rename it and add `TakeMedication()`

What are the limitations of those strategies?

Managing a sick cow

A sick cow does:

- Everything a cow does
- Take its medication

Two obvious strategies:

- Add a `TakeMediaction()` method to the cow
- Recopy the `cow` class, rename it and add `TakeMedication()`

What are the limitations of those strategies?

The solution consists in getting a sick cow to *inherits* the attributes and methods of a cow, while allowing it to add some more

Managing a sick cow

cows_1.cpp

```cpp
#include <iostream>
using namespace std;
class Cow {
  public: Cow(int f=0){grass=f;}
    void Speak () { cout << "Moo.\n"; }
    void Eat() {
      if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
      else cout << "I'm hungry\n";}
  private: int grass;
};
class SickCow : public Cow {
  public: SickCow(int f=0,int m=0){grass=f; med=m;}
    void TakeMed() {
      if(med > 0) { med--; cout << "I feel better\n";}
      else cout << "I'm dying\n";}
  private: int med;
};
int main () {
  Cow c1(1); SickCow c2(1,1);
  c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
}
```

Private

Reminder on private members:

- Everything private is only available to the current class
- Derived classes cannot access or use them

Private inheritance:

- Default type of class inheritance
- Any public member from the base class becomes private
- Allows to hide “low level” details to other classes

Public

Reminder on public members:

- They are available to the current class
- They are available to any other class

Public inheritance:

- Anything public in the base class remains public
- Nothing private in the base class can be accessed

Public

Reminder on public members:

- They are available to the current class
- They are available to any other class

Public inheritance:

- Anything public in the base class remains public
- Nothing private in the base class can be accessed

Problem:

- Private is too restrictive while public is too open
- Need a way to only allow derived classes and not others

Protected

Protected members:

- Compromise between public and private
- They are available to any derived class
- No other class can access them

Protected

Protected members:

- Compromise between public and private
- They are available to any derived class
- No other class can access them

Possible to bypass all this security using keyword `friend`:

- Valid for both functions and classes
- A class or function declares who are its friends
- Friends can access protected and private members
- As much as possible do not use `friend`

Summary on visibility

Attributes and methods:

Visibility	Classes		
	Base	Derived	Others
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Summary on visibility

Attributes and methods:

Visibility	Classes		
	Base	Derived	Others
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Inheritance:

Base class	Derived class		
	Public	Private	Protected
Private	-	-	-
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Properly managing a sick cow

cows_2.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4    public: Cow(int f=0){grass=f;}
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() {
7        if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8        else cout << "I'm hungry\n";}
9    protected: int grass;
10 };
11 class SickCow : public Cow {
12   public: SickCow(int f=0,int m=0){grass=f; med=m;}
13     void TakeMed() {
14       if(med > 0) { med--; cout << "I feel better\n";}
15       else cout << "I'm dying\n";}
16   private: int med;
17 };
18 int main () {
19   Cow c1(1); SickCow c2(1,1);
20   c1.Speak(); c1.Eat(); c1.Eat(); c2.Eat(); c2.TakeMed(); c2.TakeMed();
21 }
```

Inheritance or not inheritance?

A cow is a *mammal, while a zoo* has *mammals and reptiles*

```
1  class Cow : public Mammal {
2    ...
3  }
```

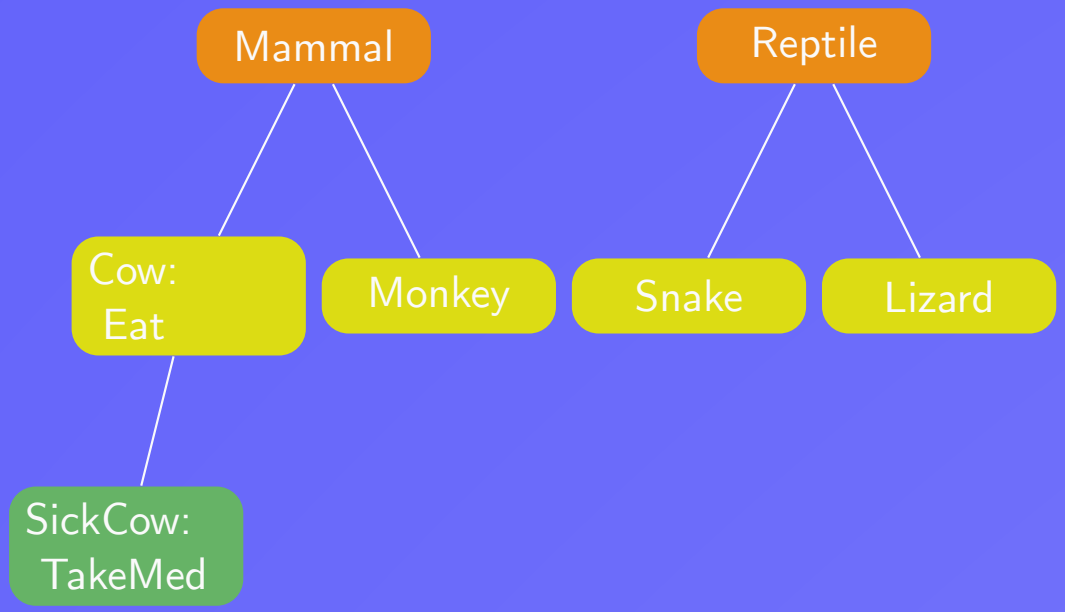
```
1  class Zoo {
2    public:
3      Mammal *m; Reptile *r;
4      ...
5  };
```

Remark.
On a drawing:

- A cow *is* a figure, a cage is a figure, a zoo is a figure...
- A cow is composed of (*has*) figures, e.g. ellipsis for the body, circle for the head, rectangles for the legs and tail
- What to choose, *is a* or *has a*?

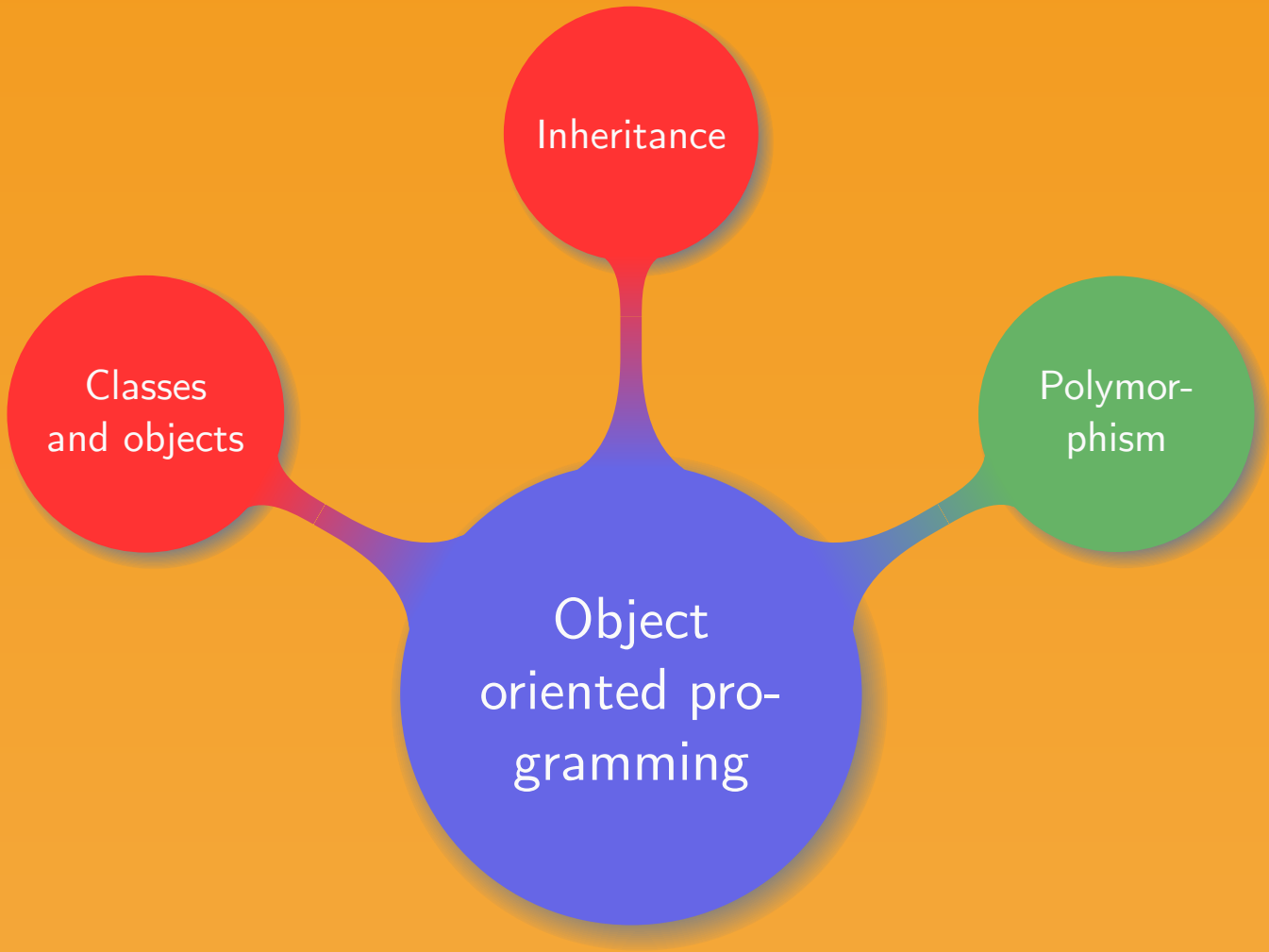
Hierarchy diagram

Representing the relationships using diagrams:

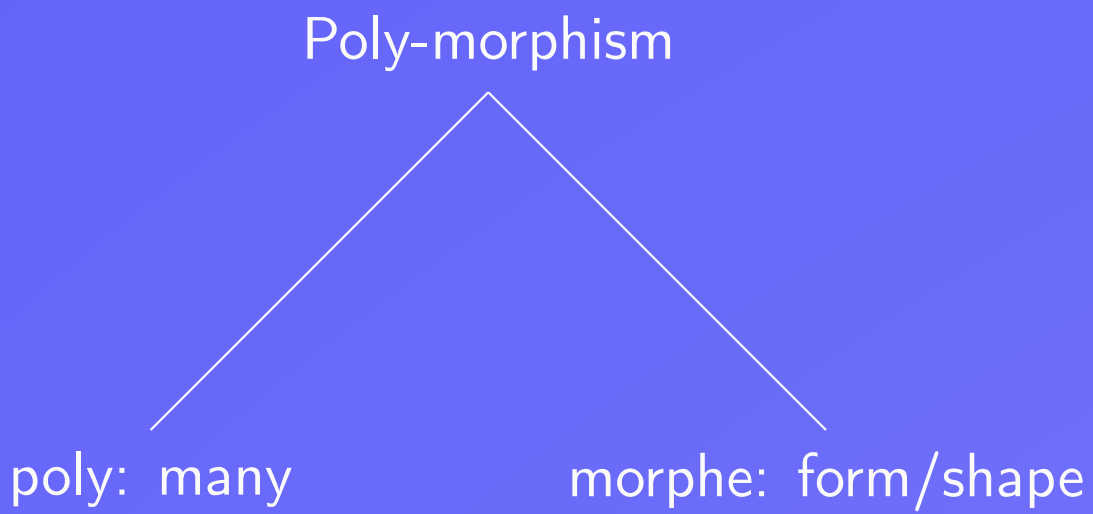


Zoo:
Reptile
Mammal
...

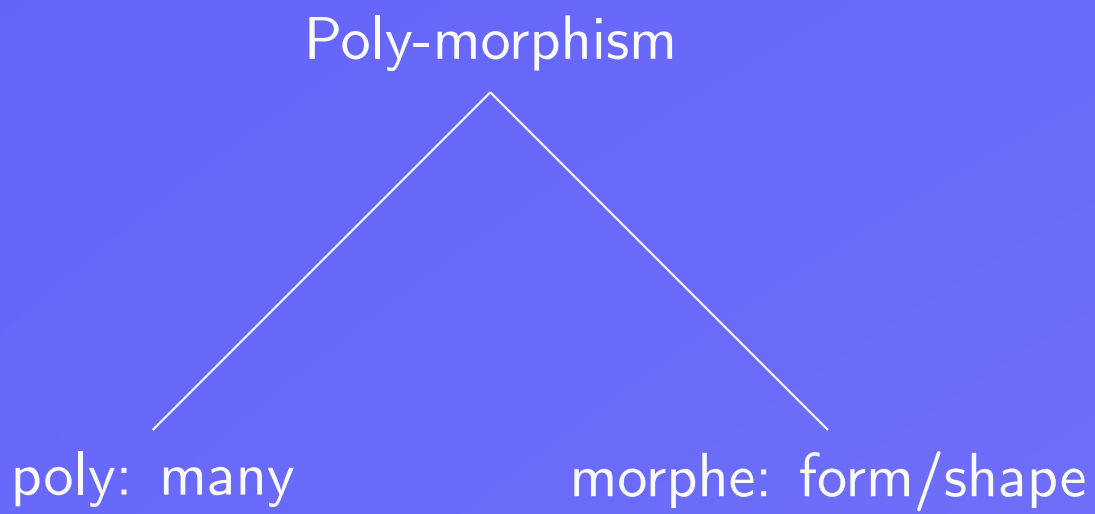
Chapter organisation



Polywhat????



Polywhat????



Simple idea:

- Arrays cannot contain different data types
- A sick cow is *almost like* a cow
- Goal: handle sick cows as cows while preserving their specifics

Function overloading

cows_3.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4    public: Cow(int f=0){grass=f;}
5      void Speak () { cout << "Moo.\n"; }
6      void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
7                   else cout << "I'm hungry\n";}
8    protected: int grass;
9  };
10 class SickCow : public Cow {
11   public: SickCow(int f=0,int m=0){grass=f; med=m;}
12     void Speak () { cout << "Ahem... Moo.\n"; }
13     void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
14                      else cout << "I'm dying\n";}
15   private: int med;
16 };
17 int main () {
18   Cow c1; SickCow c2(1); Cow *c3=&c2;
19   c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
20 }
```

Overcoming the limitations

New keyword: `virtual`

- Virtual function in the base class
- Function can be redefined in derived class
- Preserves calling properties

Overcoming the limitations

New keyword: `virtual`

- Virtual function in the base class
- Function can be redefined in derived class
- Preserves calling properties

Drawbacks:

- Binding: connecting function call to function body
- Early binding: compilation time
- Late binding: runtime, depending on the type, more expensive
- `virtual` implies late binding

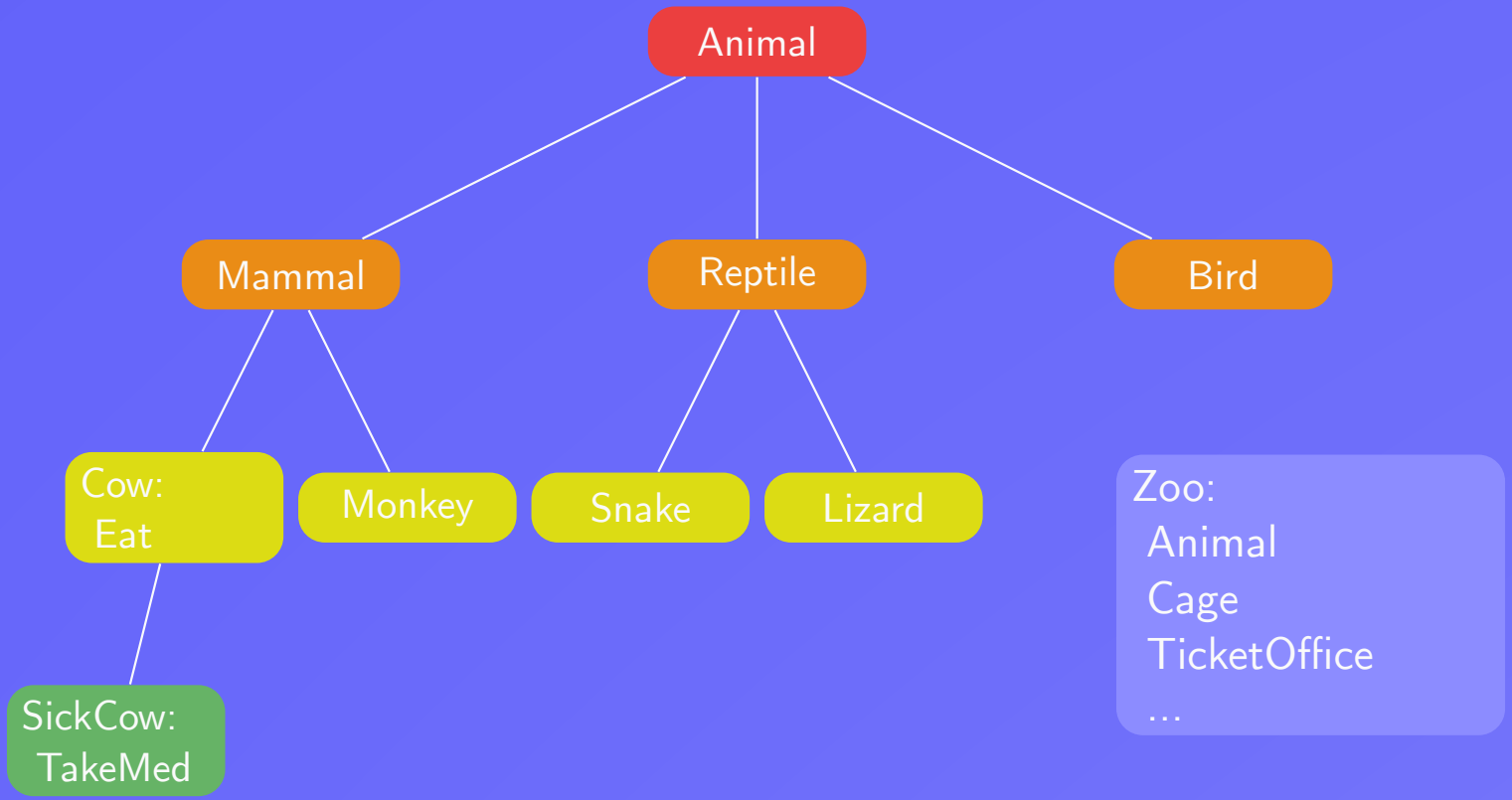
Fixing the cows

cows_4.cpp

```
1  #include <iostream>
2  using namespace std;
3  class Cow {
4    public: Cow(int f=0){grass=f;}
5      virtual void Speak () { cout << "Moo.\n"; }
6      void Eat() { if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
7                   else cout << "I'm hungry\n";}
8    protected: int grass;
9  };
10 class SickCow : public Cow {
11   public: SickCow(int f=0,int m=0){grass=f; med=m;}
12     void Speak () { cout << "Ahem... Moo.\n"; }
13     void TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
14                      else cout << "I'm dying\n";}
15   private: int med;
16 };
17 int main () {
18   Cow c1; SickCow c2(1); Cow *c3=&c2;
19   c1.Speak();c1.Eat();c2.Speak();c2.TakeMed();c3->Speak();//c3->TakeMed;
20 }
```

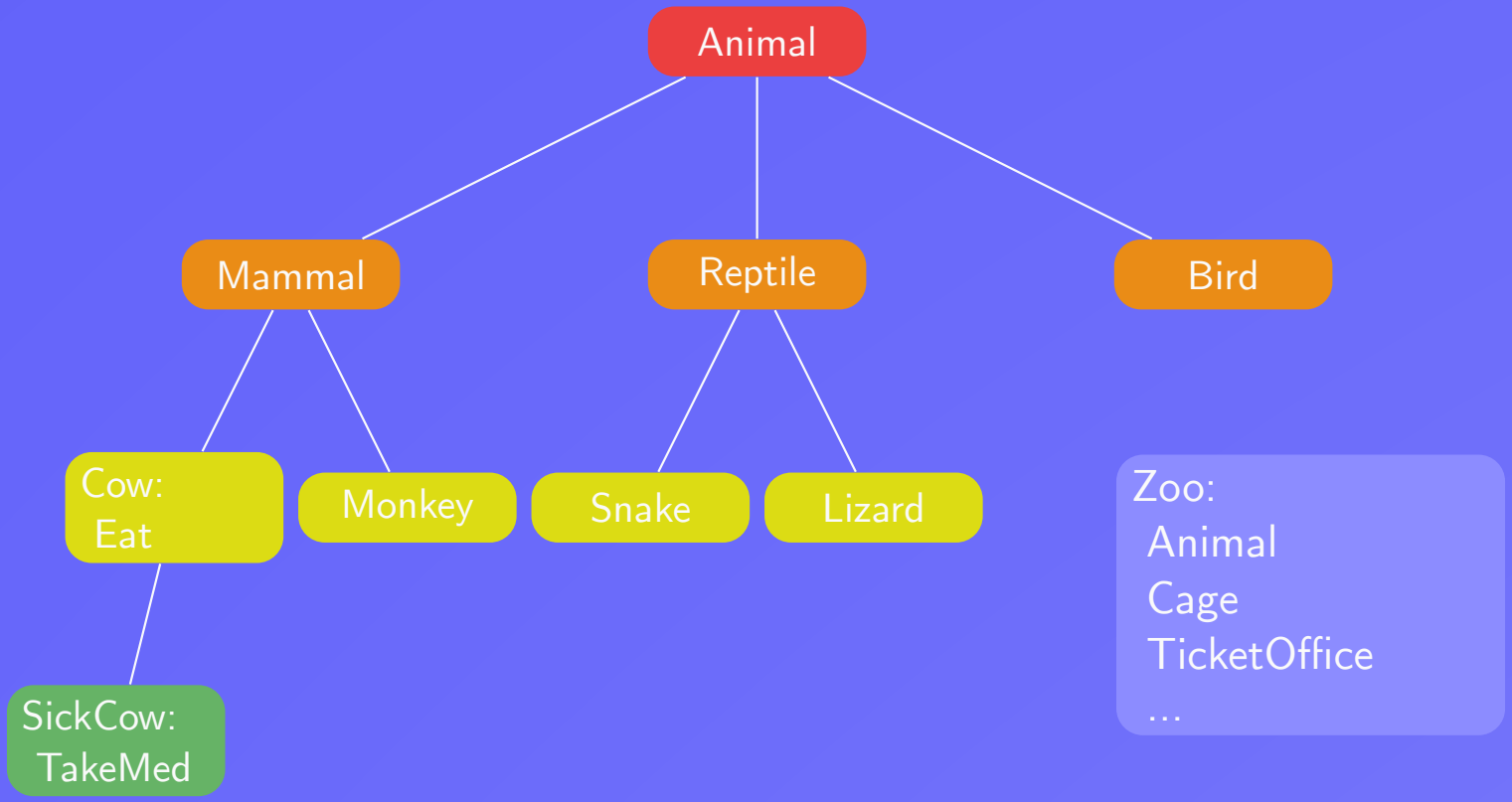
Extending the idea

Applying the same idea to generalize the diagram:



Extending the idea

Applying the same idea to generalize the diagram:



Benefits:

- Feed all the animals at once
- Animals speak their own language when asked to speak

Pure virtual methods

Pushing it further:

- Write a totally abstract class “at the top”
- This class has virtual member functions without any definition
- The method definition is replaced by `=0`

Example.

```
1  class Animal {
2    public:
3      virtual void Speak() = 0;
4  }
```

Animals

animals.h

```
1  class Animal {
2    public:
3      virtual void Speak() = 0;
4      virtual void Eat() = 0;
5  };
6  class Cow : public Animal {
7    public:
8      Cow(int f=0); virtual void Speak(); void Eat();
9    protected: int grass;
10 };
11 class SickCow : public Cow {
12   public:
13     SickCow(int f=0,int m=0); void Speak(); void TakeMed();
14   private: int med;
15 };
16 class Monkey : public Animal {
17   public:
18     Monkey(int f=0); void Speak(); void Eat();
19   protected: int banana;
20 };
```

Animals

animals.cpp

```
1  #include <iostream>
2  #include "animals.h"
3  using namespace std;
4  Cow::Cow(int f) {grass=f;}
5  void Cow::Speak() { cout << "Moo.\n"; }
6  void Cow::Eat(){
7    if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
8    else cout << "I'm hungry\n";
9  }
10 SickCow::SickCow(int f,int m) {grass=f; med=m;}
11 void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
12 void SickCow::TakeMed() {
13   if(med > 0) { med--; cout << "I feel better\n";}
14   else cout << "I'm dying\n";
15 }
16 Monkey::Monkey(int f) {banana=f;}
17 void Monkey::Speak() { cout << "Hoo hoo hoo hoo\n";}
18 void Monkey::Eat() {
19   if(banana > 0) {banana--; cout << "Give me another banana!\n";}
20   else cout << "Who took my banana?\n";
21 }
```

Zoo

zoo.h

```
1  #include <iostream>
2  #include <string>
3  #include "animals.h"
4  using namespace std;
5  class Employee {
6    public:
7      void setName(string n); string getName();
8    private:
9      string name;
10 };
11 class Tamer : public Employee {
12   public: void Feed(Animal *a);
13 };
14 class Zoo {
15   public:
16     Zoo(int s);
17     ~Zoo();
18     int getSize(); Tamer* getTamer(); Animal *getAnimal(int i);
19   private:
20     int size; Animal **a; Tamer *g;
21 };
```

Zoo

zoo.cpp

```
1  #include <iostream>
2  #include "zoo.h"
3  void Employee::setName(string n) { name=n; }
4  string Employee::getName() { return name; }
5  void Tamer::Feed(Animal *a) {a->Speak(); a->Eat();}
6  Zoo::Zoo(int s) {
7    size=s; a=new Animal*[size]; g=new Tamer;
8    for(int i=0; i<size; i++) {
9      switch(i%4) {
10       case 0: a[i]=new Cow; break; case 1: a[i]=new SickCow; break;
11       case 2: a[i]=new Monkey;break; case 3: a[i]=new Monkey(1);break;
12     }
13   }
14 }
15 Zoo::~Zoo() {
16   for(int i=0; i<size; i++) delete a[i];
17   delete[] a; delete g;
18 }
19 int Zoo::getSize() { return size; };
20 Tamer* Zoo::getTamer() { return g; }
21 Animal *Zoo::getAnimal(int i) {return a[i];}
```

Benefits of polymorphism

zoo_main.cpp

```
1  #include <iostream>
2  #include "zoo.h"
3  int main () {
4    Zoo z(10); z.getTamer()->setName("Mike");
5    cout << "Hi " << z.getTamer()->getName()
6      << ", please feed the animals.\n";
7    for(int i=0; i<z.getSize(); i++) {
8      cout << endl;
9      z.getTamer()->Feed(z.getAnimal(i));
10   }
11 }
```

Remark.

How many lines of code are necessary to achieve the same result without inheritance and polymorphism?

Benefits of polymorphism

Understanding the code:

- Explain the benefits of polymorphism
- Why is the `Zoo` destructor not empty?
- Is it possible to instantiate an `Animal`?
- Adapt the previous classes and main function to add:
  - Cages that can be locked and unlocked
  - A vet and more guards
  - A boss, who gives orders while other employees do the real work (feed, give medication, open cages...)
  - Visitors who can watch the animals, get a fine if they feed the animals...
  - If an animal escapes there is an emergency announcement and the zoo closes

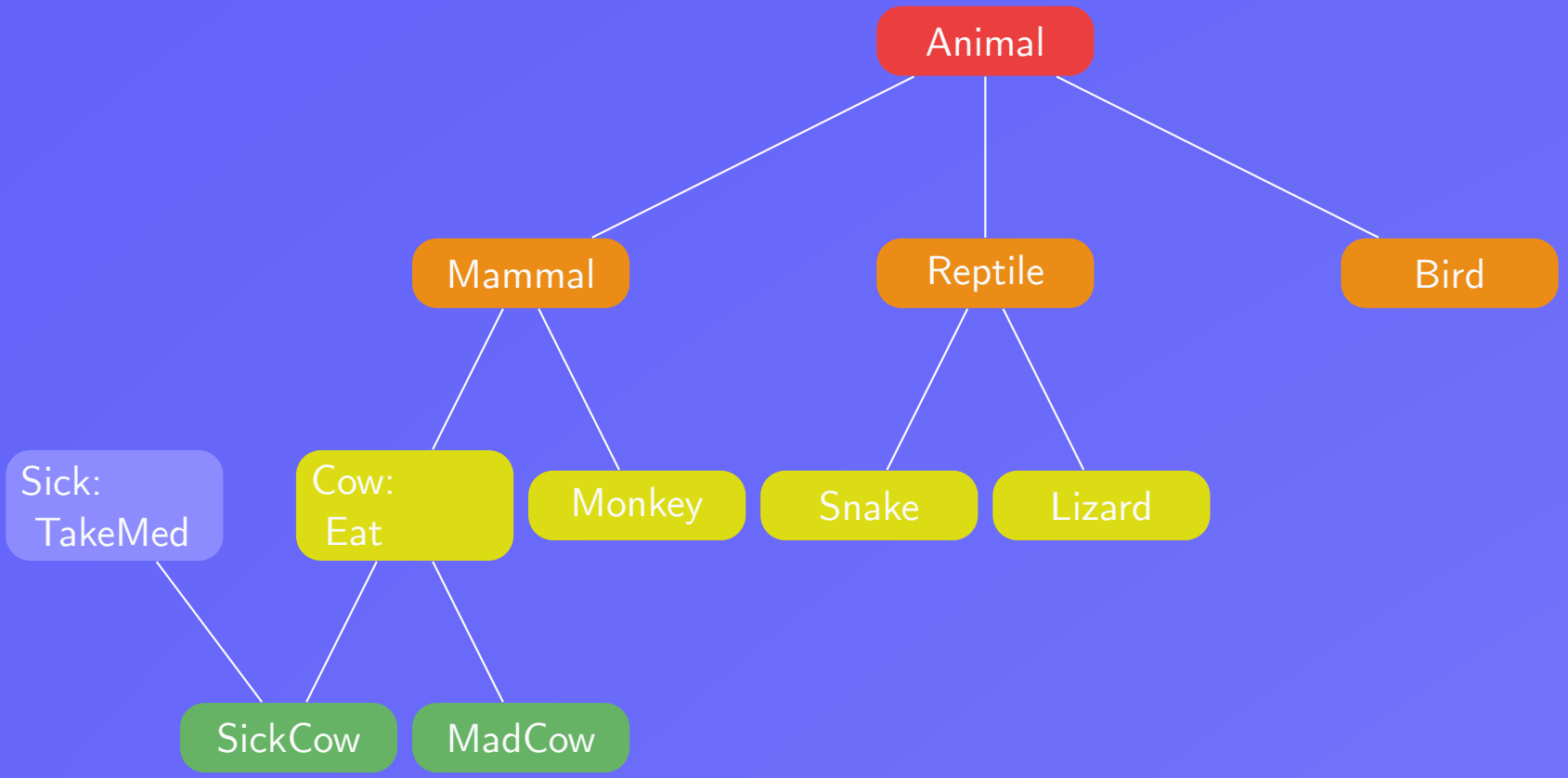
Multiple inheritance

With multiple inheritance, a class can inherit from several classes

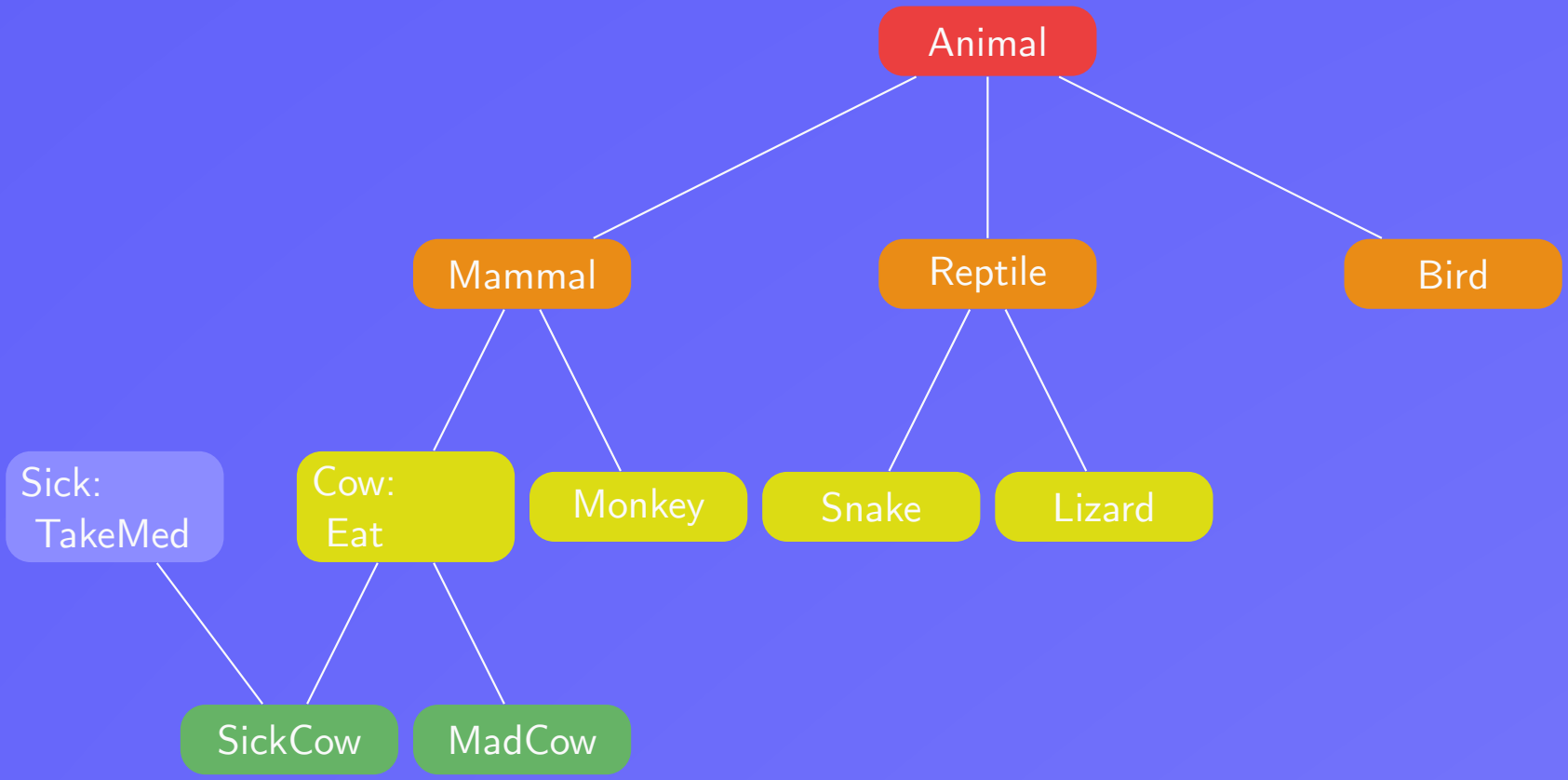
Example.
Any sick animal should be put under medication:

- Not only cows can be sick
- Create a generic "sick class" that can be used by any animal
- A sick cow *is* a cow and *is* sick
- A sick cow inherits from sick and from cow

Multiple inheritance



Multiple inheritance




```
1  class SickCow : public Cow, public Sick {
2     ...
3  }
```

More cows

animals_m.h

```
1  class Animal {
2    public:
3      virtual void Speak() = 0; virtual void Eat() = 0;
4  };
5  class Sick {
6    public:  void TakeMed();
7    protected: int med;
8  };
9  class Cow : public Animal {
10   public: Cow(int f=0); virtual void Speak(); void Eat();
11   protected: int grass;
12 };
13 class SickCow : public Cow, public Sick {
14   public: SickCow(int f=0,int m=0); void Speak();
15 };
16 class MadCow : public Cow {
17   public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
18   protected: int pills;
19 };
```

More cows

animals_m.cpp

```
1  #include <iostream>
2  #include "animals_m.h"
3  using namespace std;
4  void Sick::TakeMed(){
5    if(med > 0) { med--; cout << "I feel better\n";}
6    else cout << "I'm dying\n";
7  }
8  Cow::Cow(int f) {grass=f;}
9  void Cow::Speak() { cout << "Moo.\n"; }
10 void Cow::Eat(){
11   if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
12   else cout << "I'm hungry\n";
13 }
14 SickCow::SickCow(int f,int m) {grass=f; med=m;}
15 void SickCow::Speak() { cout << "Ahem... Moo.\n"; }
16 MadCow::MadCow(int f, int p) {grass=f; pills=p;}
17 void MadCow::Speak() { cout << "Woof\n";}
18 void MadCow::TakePills() {
19   if(pills > 0) {pills--; cout << "Moof, that's better\n";}
20   else cout << "Woof woof woof!\n";
21 }
```

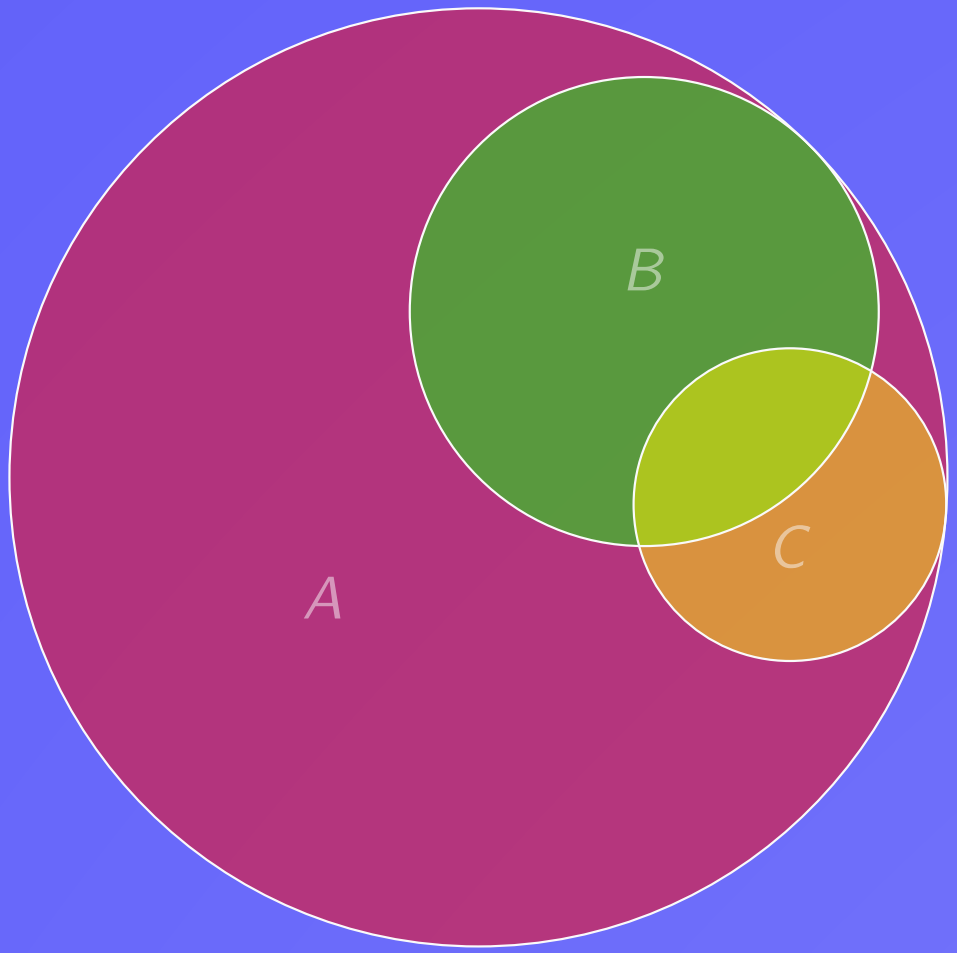
More cows

animals_main_m.cpp

```cpp
#include <iostream>
#include "animals_m.h"
using namespace std;
int main () {
  SickCow c1(1,1);
  c1.Speak(); c1.Eat(); c1.TakeMed();
  c1.Eat(); c1.TakeMed();
  cout << endl;
  MadCow c2(1,1);
  c2.Speak(); c2.Eat(); c2.TakePills();
  c2.Eat(); c2.TakePills();
}
```

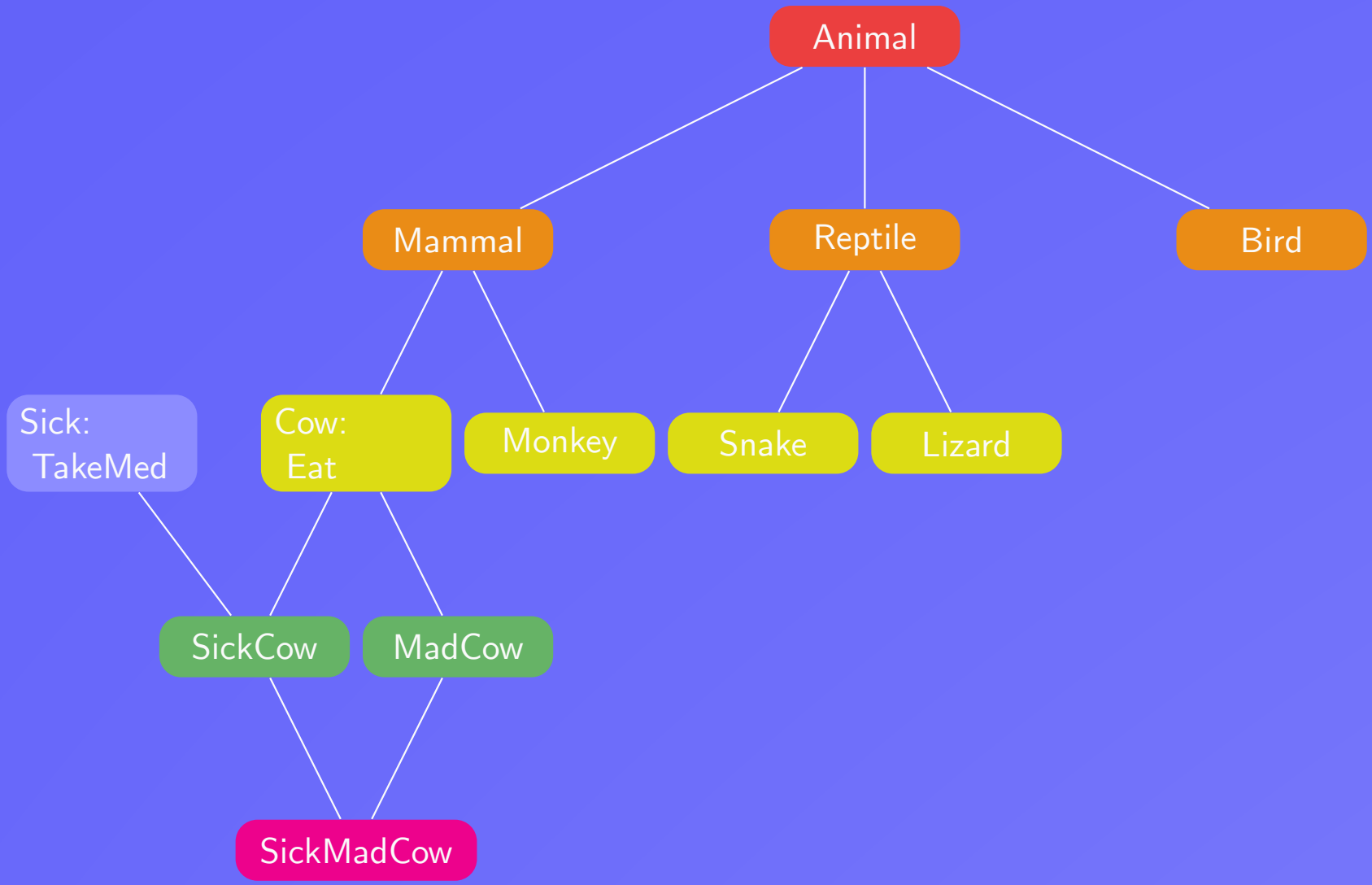
The diamond problem

Multiple inheritance can be tricky:



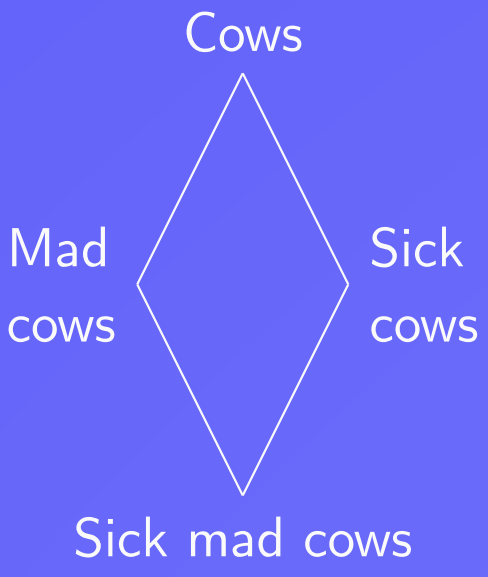
- A: Cows
- B: Sick cows
- C: Mad cows
- Sick mad cows are in $B \cap C$

The diamond problem

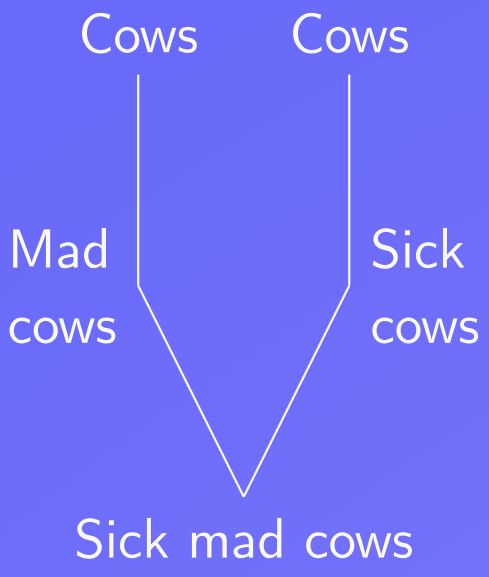


The diamond problem

Human perspective

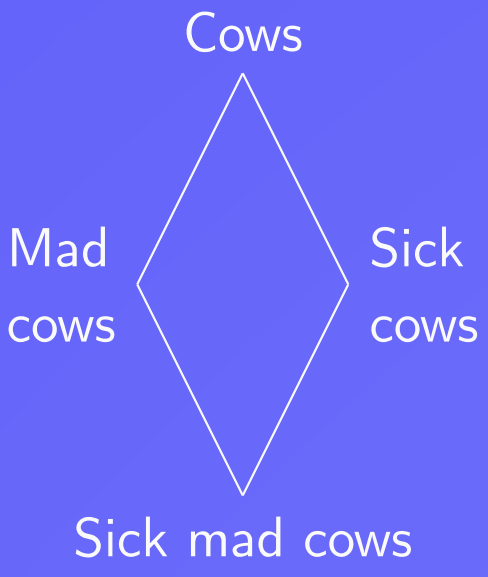


Computer perspective

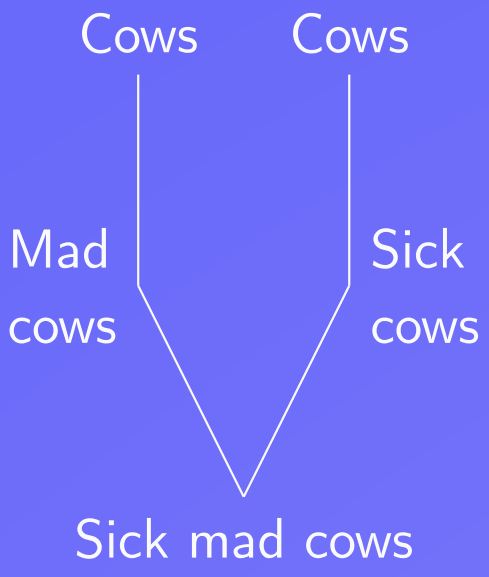


The diamond problem

Human perspective



Computer perspective



Major issues:

- Is `Eat` inherited from `Cow` through `SickCow` or `MadCow`?
- What happens if the variable `grass` is updated?

The diamond problem

Solutions to overcome the problem:

- Best: create a hierarchy without diamond problem
- Declare the derived classes as virtual

```
1 class Cow {...};
2 class SickCow : public virtual Cow {...};
3 class MadCow : public virtual Cow {...};
4 class SickMadCow : public SickCow, public MadCow {...};
```

Calling `Eat` *or updating* `grass` *does not generate any problem*

The diamond problem

Solutions to overcome the problem:

- Best: create a hierarchy without diamond problem
- Declare the derived classes as virtual

```
1 class Cow {...};
2 class SickCow : public virtual Cow {...};
3 class MadCow : public virtual Cow {...};
4 class SickMadCow : public SickCow, public MadCow {...};
```

Calling `Eat` *or updating* `grass` *does not generate any problem*

Never design a hierarchy diagram exhibiting a diamond problem

Sick mad cows

animals_d.h

```
1  class Animal {
2    public: virtual void Speak() = 0; virtual void Eat() = 0;
3  };
4  class Sick {
5    public:  void TakeMed();
6    protected: int med;
7  };
8  class Cow : public Animal {
9    public: Cow(int f=0); virtual void Speak(); void Eat();
10   protected: int grass;
11 };
12 class SickCow : public virtual Cow, public Sick {
13   public: SickCow(int f=0,int m=0); void Speak();
14 };
15 class MadCow : public virtual Cow {
16   public: MadCow(int f=0,int p=0); void Speak(); void TakePills();
17   protected: int pills;
18 };
19 class SickMadCow : public SickCow, public MadCow {
20   public: SickMadCow(int f=0, int m=0, int p=0); void Speak();
21 };
```

Sick mad cows

animals_d.cpp

```cpp
#include <iostream>
#include "animals_d.h"
using namespace std;
void Sick::TakeMed() { if(med > 0) { med--; cout << "I feel better\n";}
  else cout << "I'm dying\n";
}
Cow::Cow(int f) {grass=f;}
void Cow::Speak() { cout << "Moo.\n"; }
void Cow::Eat(){ if(grass > 0) { grass-- ; cout << "Thanks I'm full\n";}
  else cout << "I'm hungry\n";
}
SickCow::SickCow(int f,int m) {grass=f; med=m;}
void SickCow::Speak() { cout << "Ahem... Moo\n"; }
MadCow::MadCow(int f, int p) {grass=f; pills=p;}
void MadCow::Speak() { cout << "Woof\n";}
void MadCow::TakePills() {
  if(pills > 0) {pills--; cout << "Moof, that's better\n";}
  else cout << "Woof woof woof!\n";
}
SickMadCow::SickMadCow(int f, int m, int p) {grass=f; med=m; pills=p;}
void SickMadCow::Speak() {cout << "Ahem... Woof\n";}
```

Sick mad cows

animals_main_d.cpp

```cpp
#include <iostream>
#include "animals_d.h"
using namespace std;
int main () {
  SickCow c1(1,1);
  c1.Speak(); c1.Eat(); c1.TakeMed();
  c1.Eat(); c1.TakeMed();
  cout << endl;
  MadCow c2(1,1);
  c2.Speak(); c2.Eat(); c2.TakePills();
  c2.Eat(); c2.TakePills();
  cout << endl;
  SickMadCow c3(1,1,1);
  c3.Speak(); c3.Eat(); c3.TakePills(); c3.TakeMed();
  c3.Eat(); c3.TakePills(); c3.TakeMed();
  SickMadCow  c4(1,1,0); Cow *c5=&c4;
  c4.Speak(); c4.Eat(); c4.TakePills(); c4.TakeMed();
  c5->Speak(); c5->Eat(); //c5->TakePills(); c5->TakeMed();
}
```

Sick mad cows

Understanding the code:

- How is polymorphism used?
- Describe the diamond problem
- How was the problem overcome?
- Draw a hierarchy diagram without the diamond problem
- What is happening if line 18 (10.57) is uncommented? Why?

Project development

Process to organise a project:

1. Define what is needed or expected
2. Express everything in terms of objects
3. Define the relationships among the objects
4. Abstract new classes
5. Draw the hierarchy diagram
6. If there is any diamond, adjust the diagram
7. For each object define the methods
8. For each object define the attributes
9. Write the classes

Key points

- What is object oriented programming?
- In what order should the attributes and methods be defined?
- What are private and public?
- Why using inheritance?
- What is polymorphism?
- What is the best way to solve the diamond problem?

Thank you!