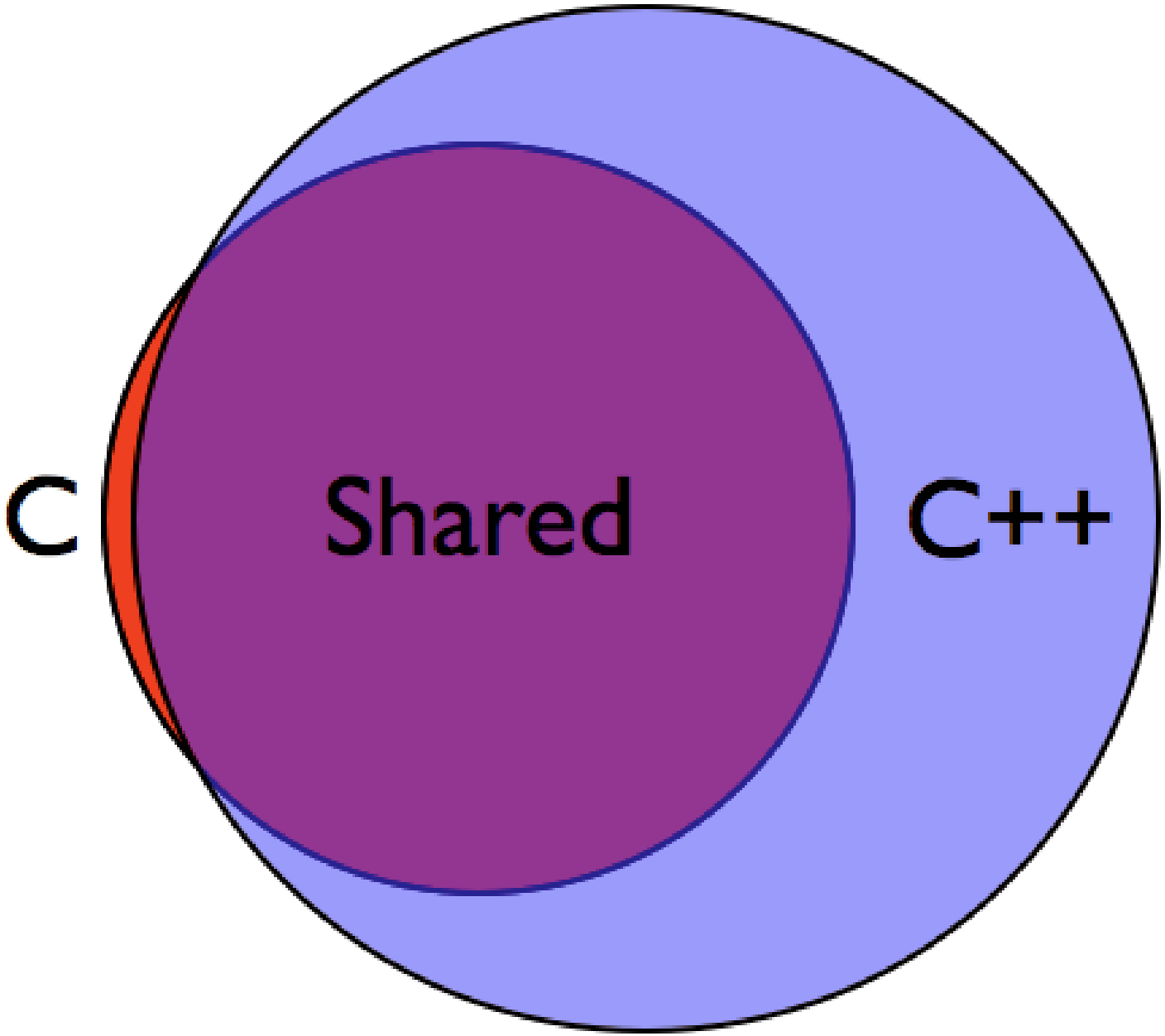
VG101 Final

FA2019 Chujie Ni

Table of Content



Relation between C/C++

- They shared a lot of syntax and features.
- C++ is more advanced than C. (High-level)

Three Features of C++

- **Encapsulation**: class
- **Inheritance**: derived class (subtype)
- **Polymorphisom**: virtual function, overload & override

Some Basic Concepts

- **class**: The defination of a type.
  - class Circle { ... };
  - class is like a product blueprint, the things we produced using this blueprint are `instance`.
- **derived class(subclass or subtype)**:
  - A class inherited from other class
  - The class it inherited from is called base class (superclass or supertype)
  - Don't call it father/mother class

Some Basic Concepts

- **instance**:
  - Things we produced with class defination.
  - Circle a, b;
  - A class can have a lot of instances.
- **object**:
  - Has broader meaning than `instance`, but somehow equivalent.
  - A more general concept.
  - A more philosophical concept.

- **interface**:
  - A series of class/methods for other programmers to rewrite or inherit from or use.
  - In C++ most time it refers to `Abstract class`.
  - `Abstract class` are classes with `virtual` functions, and **cannot have an instance**.
- **implementation**:
  - Detailed information of the programme
  - Usually written in `.cpp` file and is separated from defination.
  - **Usually, we want to hide the implementation.**

- **method**:
  - Functions belong to a `class`.
  - Define the behavior of an object.
  - Also called `member function`.
- **attribute**:
  - Variables belong to a `class`.
  - Data/property of an object.
  - Also called `member variable`.

- **overload**:
  - C++ allows you to write functions of **same name** but different `parameters number/parameters type`.
- **override**:
  - Achieved by `virtual`.
  - In a derived class, rewrite the virtual function inherited from the base class is called `function overriding`

Understanding of "other programmers"

Key Concept: Different Kinds of Programming Roles

Programmers tend to think about the people who will run their applications as *users*. Similarly a class designer designs and implements a class for *users* of that class. In this case, the user is a programmer, not the ultimate user of the application.

When we refer to a *user*, the context makes it clear which kind of user is meant. If we speak of *user code* or the *user* of the `Sales_data` class, we mean a programmer who is using a class. If we speak of the *user* of the bookstore application, we mean the manager of the store who is running the application.

Note

C++ programmers tend to speak of *users* interchangeably as users of the application or users of a class.

In simple applications, the user of a class and the designer of the class might be one and the same person. Even in such cases, it is useful to keep the roles distinct. When we design the interface of a class, we should think about how easy it will be to use the class. When we use the class, we shouldn't think about how the class works.

Authors of successful applications do a good job of understanding and implementing the needs of the application's users. Similarly, good class designers pay close attention to the needs of the programmers who will use the class. A well-designed class has an interface that is intuitive and easy to use and has an implementation that is efficient enough for its intended use.

C++ Primer, 5th edition, Chapter 7.1

Table of Content

OOP (Object-Oriented Programming)

- A philosophical concept~
- Everything is an object
- The TV example
- Methods and Attributes

Order of defining a class

- Define the methods
- Define the attributes

Why?

- Because we define a class to solve specific problems. We must think of the behavior of the object first. Then we select proper attributes to implement these methods.
- Otherwise, we can easily omit necessary attributes or define some extra useless ones.

class in C++

Circle.h (interface)

```
class Circle {
     /* user methods (and attributes)*/
       public:
         void move(float dx, float dy);
         void zoom(float scale);
         float area();
     /* implementation attributes (and methods) */
       private:
         float x, y, r;
}
```

Declarations are written in .h files.

class in C++

Circle.cpp (detailed implementation)

```
#include "Circle.h"
void Circle::move(float dx, float dy) { ... }  \\ specific implementation
void Circle::move(float scale) { ... }
float Circle::area() { ... }
```

Separated Files: GOOD!

More convenient to construct larger program; more detailed information can be hidden; easier for teamwork on a program; focus on the problem rather than details...

Constructor & Destructor

Constructor & Destructor are called *automatically*.

```
class Circle {
     /* user methods (and attributes)*/
public:
    Circle();           // A default constructor
    Circle(float r);    // A constructor with one parameter
    ~Circle();          // Only one destructor, no parameters allowed.
    // You cannot call destructor manually
private:
    float x, y;
    float r;
}
```

When are they called?

```
int main() {
    Circle c1;      // no parameters provided, default constrcutor called
    Circle c2(5);   // Circle(float r); called
    return 0;
}
// After the program exits, all objects are destroyed,
// and their destructor are called automatically.
```

Attention: Destructors will be called automatically don't mean that memory allocated will be collected automatically.

Constructor

- Attributes of a class will not be initilized automatically when we create an instance.
- A special **method** with exact the same name with the class. Used to initialize the object.
- A class can have **multiple constructor** with different parameters (function overloading).

Destructor

Every memory allocated **in a class** should be deleted in the destructor.

```
class Circle {
public:
    Circle() { this->r = new float; };
    // Also other member functions may allocate for new memory.
    // You need to delete them all in the destructor.
    ~Circle() { delete this->r };
private:
    float *r;}
```

While memory allocated at other places in your program still should be deleted at that place manually.

What if no Destructor?

Namespace

Circle.cpp

```
#include "Circle.h"
void Circle::move(float dx, float dy) { ... }  \\ specific implementation
void Circle::move(float scale) { ... }
float Circle::area() { ... }
```

A special operator "::"

"::" means **belongs to**. Each class has its own namespace and we use "::" to show this method belongs to this namespace.

Namespace

You cand define your own namespace. It is a way to avoid name conflict:

```
#include <iostream>

namespace vg101 { int score = 0; } // You can also add functions

namespace vv256 { int score = -100; } // Even classes. Actually everything is fine.

int main() {
    std::cout << vg101::score << std::endl;
    std::cout << vv256::score << std::endl;
    return 0;
}
```

Namespace

And to import the variables and methods in a namespace at one time:

```
#include <iostream>
using namespace std;  // Import all content in the "std" namespace

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Now we can use all things in the namespace `std` without adding `std::`

Table of Content

New/Delete

It's easy to manage memory in C++:

```
int *a = new int;        // Allocate a new int
int *b = new int(10);    // Allocate a new int, and initialize it as 10
// The () is actually constructor
int *c = new int[10];    // Allocate an array of int.
```

And delete it before exiting:

```
delete a;
delete b;
delete[] c;      // Use delete[] to delete an array
```

Table of Content

Standard and File I/O

```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    int a;
    cin >> a;
    cout << a;
}
```

- `cin` and `cout` are actually two objects defined by C++ (in iostream)
- `<<` and `>>` are two operators overloaded for `cin` and `cout`.

fio_c.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
void FileIO(){
  string s;
  ifstream a("1.txt"); ofstream b("2.txt",ios::app);
  if (a.is_open() && b.is_open()) {
    while(getline(a,s)) {b << s << endl; cout << s;}
    b.close(); a.close();
  }
  else cerr << "Unable to open the file(s)\n";
}
int main () {FileIO();return 0;}
```

Open Mode of Files

Member types and constants

Type	Explanation
openmode	stream open mode type The following constants are also defined:

Constant	Explanation
app	seek to the end of stream before each write
binary	open in binary mode
in	open for reading
out	open for writing
trunc	discard the contents of the stream when opening
ate	seek to the end of stream immediately after open

Table of Content

Encapsulation

- Mainly achieved with `class`
- class has three access modifiers:
  - public
  - private (default)
  - protected
- By default, the access level is `private`.

Encapsulation

Suppose now we have a class called `Human`.

```
// Human.h
class Human {
private:
    int age = 0;
public:
    void grow() { this->age++; };
};
```

Here `this` is a pointer to the current instance. It can be ignored. BUT explicit is alway better...

We cannot access private members directly. Private members are only available in its own class.

```
// main.cpp
#incluce "Human.h"   // other necessary header files emitted

int main() {
    Human ncj;           // Declare a new instance
    cout << ncj.age;     // Not OK.
    ncj.age = 19;        // Not OK
    ncj.grow();          // OK!
}
```

The only way here to increase a Human's `age` is to use the `grow()` function, which protect this attribute.

What if we really want to know the value?

```
// Human.h
class Human {
private:
    int age = 0;
public:
    void grow() { this->age++; };
    int getAge() { return this->age; };
    // Only return value, not pointer or reference.
};
```

By now, we can get the value of `age` from outside the class, but we still cannot modify it directly: this protect it from unexpected modification.

Instance

Each instance has its own attributes, but they share methods.

```
int main() {
    Human ncj, manuel;      // Declare two different instances.
    ncj.grow();
    manuel.grow();
    manuel.grow();
}
```

Then manuel's age is 2 and ncj's age is 1.
They have their own attributes and won't affect each other.

They are independent objects !!

Table of Content

Inheritance

- The capability of a class to derive properties and characteristics from another class is called **Inheritance**.
- Allows you to write less code. (avoid redundant/similar code)

Inheritance

- Usually we use public inheritance.
- The derived class cannot access the private member in inherited(super) class.

```
class SickCow : public Cow {
    ...
}
```

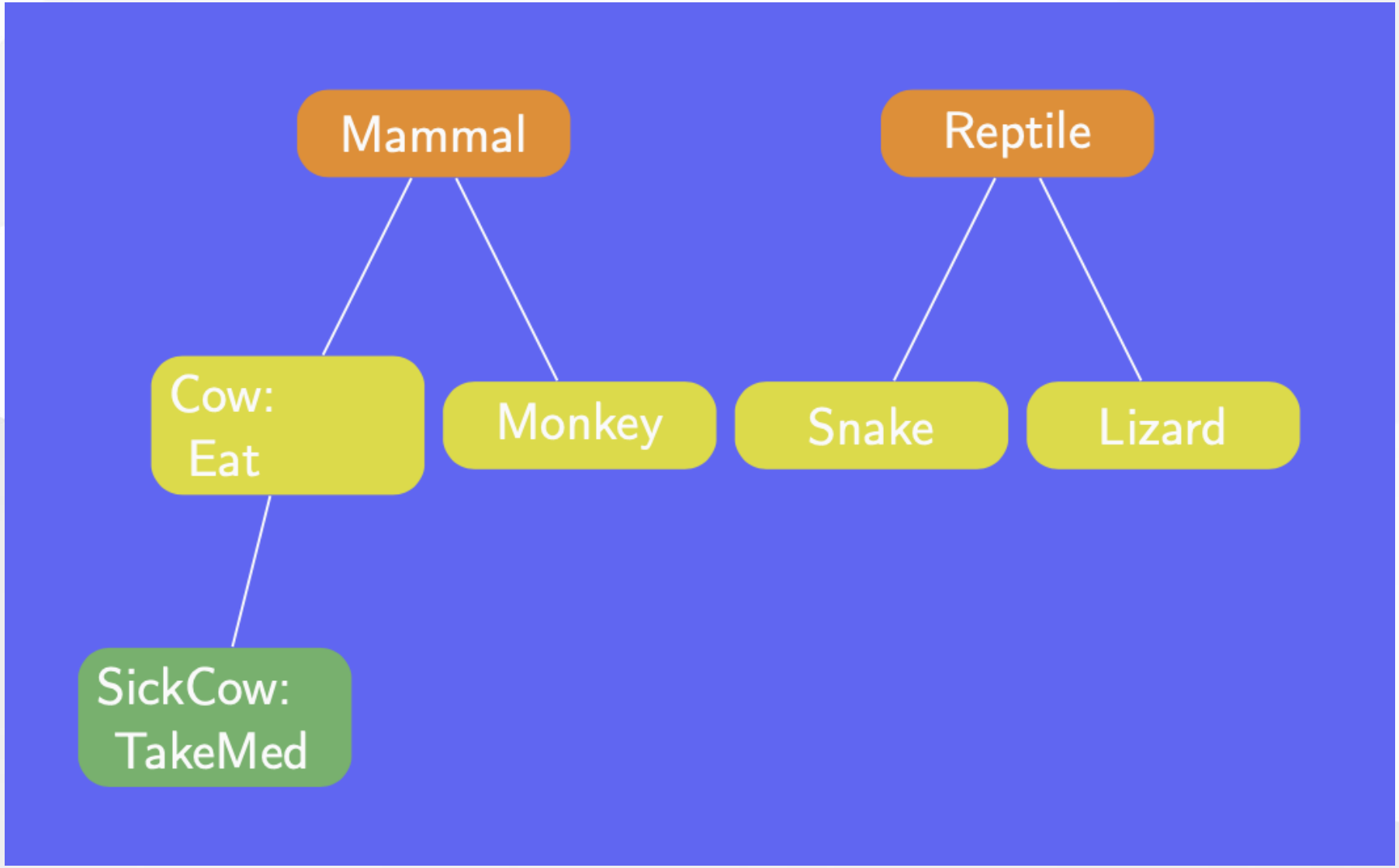
Remember the basic syntax of inheriting a class!!

Inheritance

- Base class and derived class must have similar **features** and **behaviors**.
- Base class is more general/abstract than derived class.
- It's not a relation of position or inclusion.
  - Not OK: `Classroom` -> `Student`
  - OK: `Student` -> `JI Student` -> `TA`, `VG101 Student`
  - OK: `Room` -> `Classroom`, `Discussion Room`, `Washroom`
- Substitution rule: Code written to correctly use a base class is still correct if it uses a derived class.

Hierarchy Diagram

A diagram used to show the inheritance relations of classes in a program.


Mammal
Reptile
Cow:
Eat
Monkey
Snake
Lizard
SickCow:
TakeMed

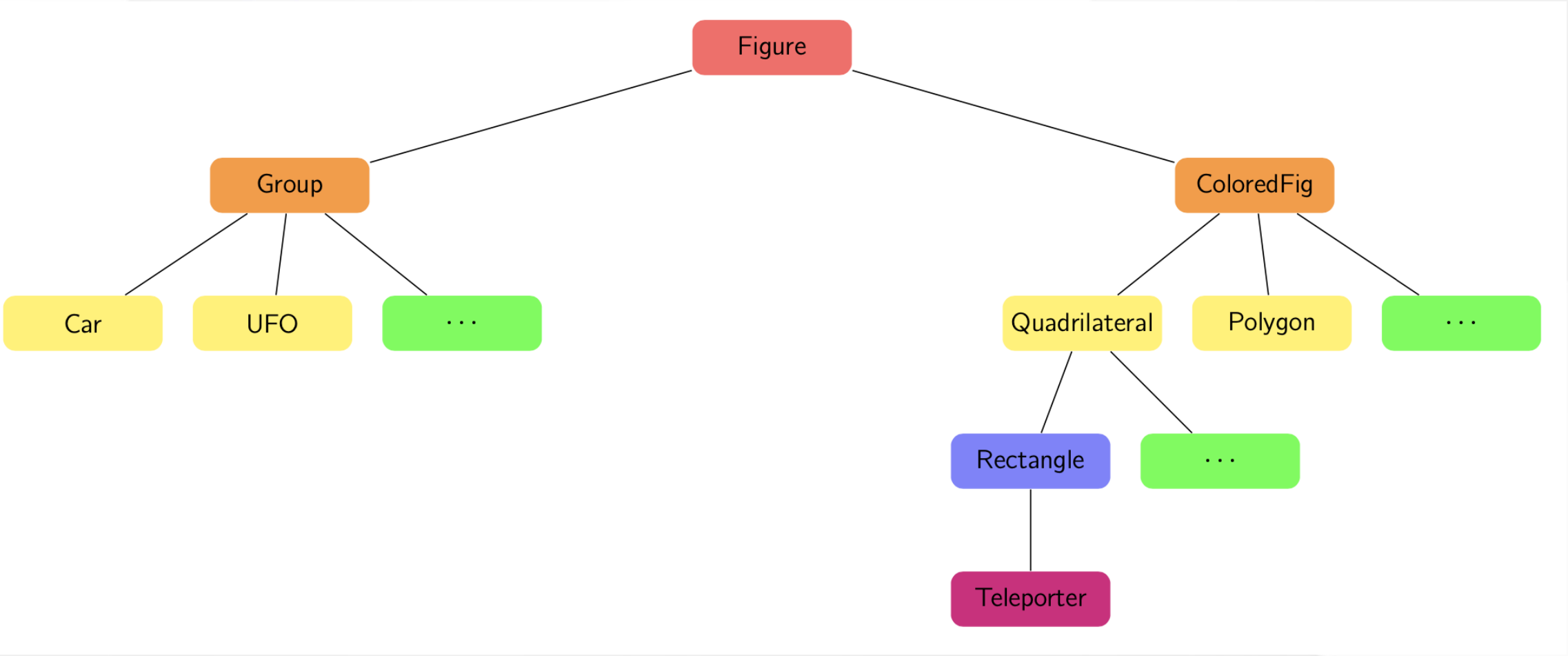
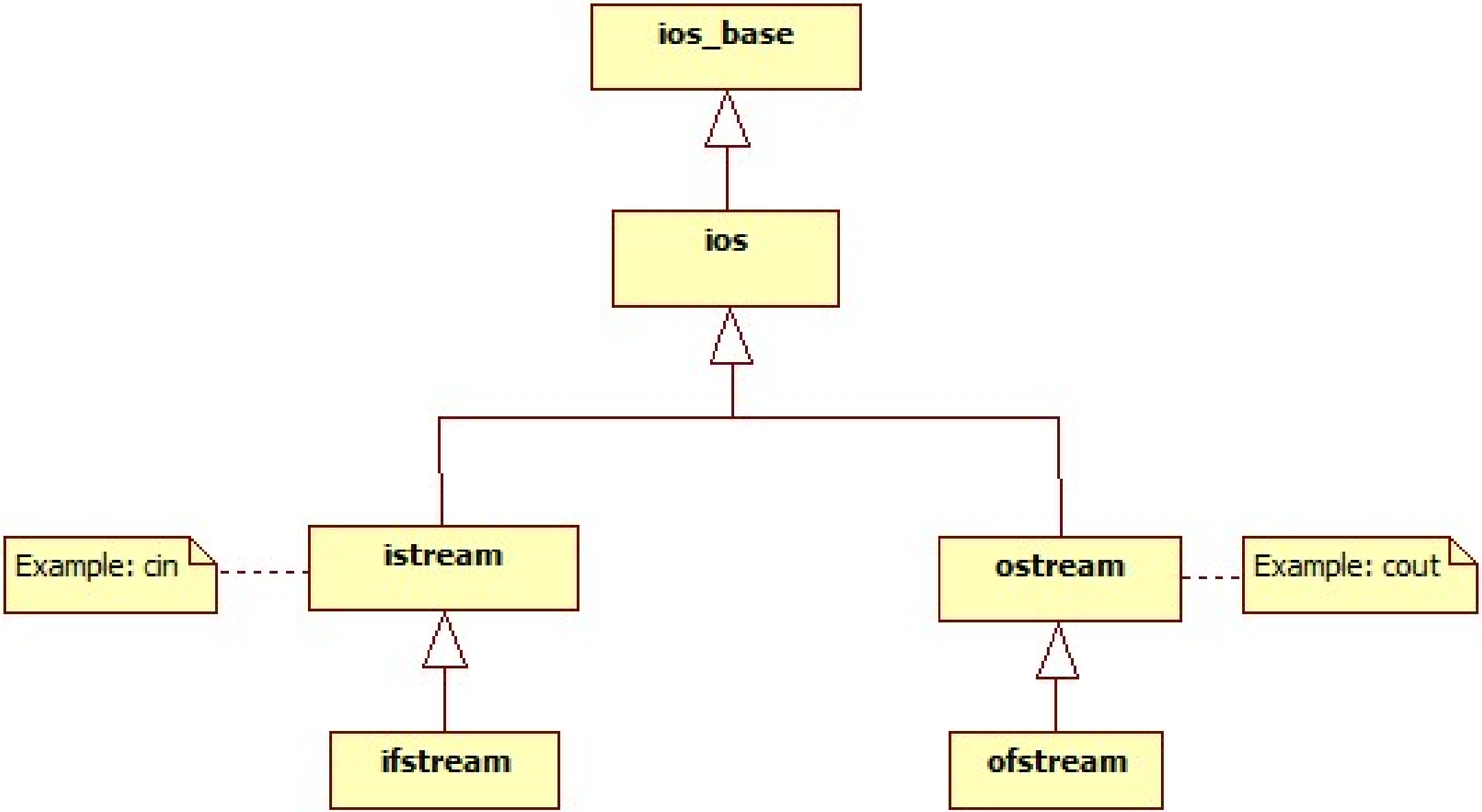


Figure
Group
ColoredFig
Car
UFO
· · ·
Quadrilateral
Polygon
· · ·
Rectangle
· · ·
Teleporter


ios_base
ios
istream
ifstream
ostream
ofstream
Example: cin
Example: cout

Pratice

Draw the hierarchy diagram of SJTU campus, at least three base classes should be chosen. e.g. `Building`

*Pointers

```
Cow* p;          // A pointer of base class
SickCow c2;      // An instance of a derived class
p=&c2;           // Let p point to c2
```

C++ allows you to declare a pointer of the base class, and point it to instances of derived classes (of any level).

But not vice versa!!! (A pointer of derived class cannot point to a base class).

C++ doesn't do a type-casting. Instead, some details of `Sickcow` are just *hidden* when we are using this pointer `p`.

*Pointers

```
Cow* p;          // A pointer of base class
SickCow c2;      // An instance of a derived class
p=&c2;           // Let p point to c2
```

- **Apparent type**: the declared type of the pointer. (Cow)
- **Actual type**: the real type of the pointer. (PosIntSet)

In default situation, C++ chooses the method to run based on its **apparent type**.

Multiple inheritance

- With multiple inheritance, a class can inherit from several classes.
- Allow you to assemble the property of several classes onto one.
- More tricky and more risky.

The Diamond Problem

- We can always avoid diamond inheritance.
- Or we can use `virtual` inheritance (actually we seldom use this).

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

Table of Content

Polymorphism

- General understanding: An object/function can show different characteristics/behaviors under different conditions.
- Specific meaning in C++:
  - Function overloading: functions with same name but different parameters (number/type)
  - Operator overloading: a same operator can apply different operation to different objects
  - Function overriding: a derived class rewrite the `virtual` function in the base class.

Operator Overloading

We can view an operator as a function:

```
int a=1, b=2;
a = a + b;
```

And `+` is to some extent equivalent to:

```
int add(int x, int y) { return x + y; }
a = add(a, b);
```

For different objects, C++ will call match the corresponding operator.

Operator Overloading

```
Vec operator+ (Vec v) {
    return Vec(x + v.getX(), y + v.getY());
}   // Note: here v.x and v.y are also ok since it's still inside the same class.
```

What is the first operand?

- The first operand is **this** class, and the second is the parameter `v` of this function.

Does the order of operands matter?

- In this situation it doesn's matter since the operation is symmetric.

Overriding

- Achieved by `virtual` function and pointers.
- `virtual` functions with `= 0` at the end are called pure virtual function.
- A class with pure virtual function cannot have instances and is called `abstract class`. Any derived class of it must override all pure virtual functions.

*Overriding

Recall the concept of **Apparent type** and **Actual type**.

```
Cow* p;          // A pointer of base class
SickCow c2;      // An instance of a derived class
p=&c2;           // Let p point to c2
p->Speak();
```

By default, the last line will call `Cow`'s `Speak()` function.
And if we declare the `Speak()` function in `Cow` as `virtual` and rewrite it in `SickCow`: the compiler will search for its **actual type**.

Note: **virtualness** is also inherited.

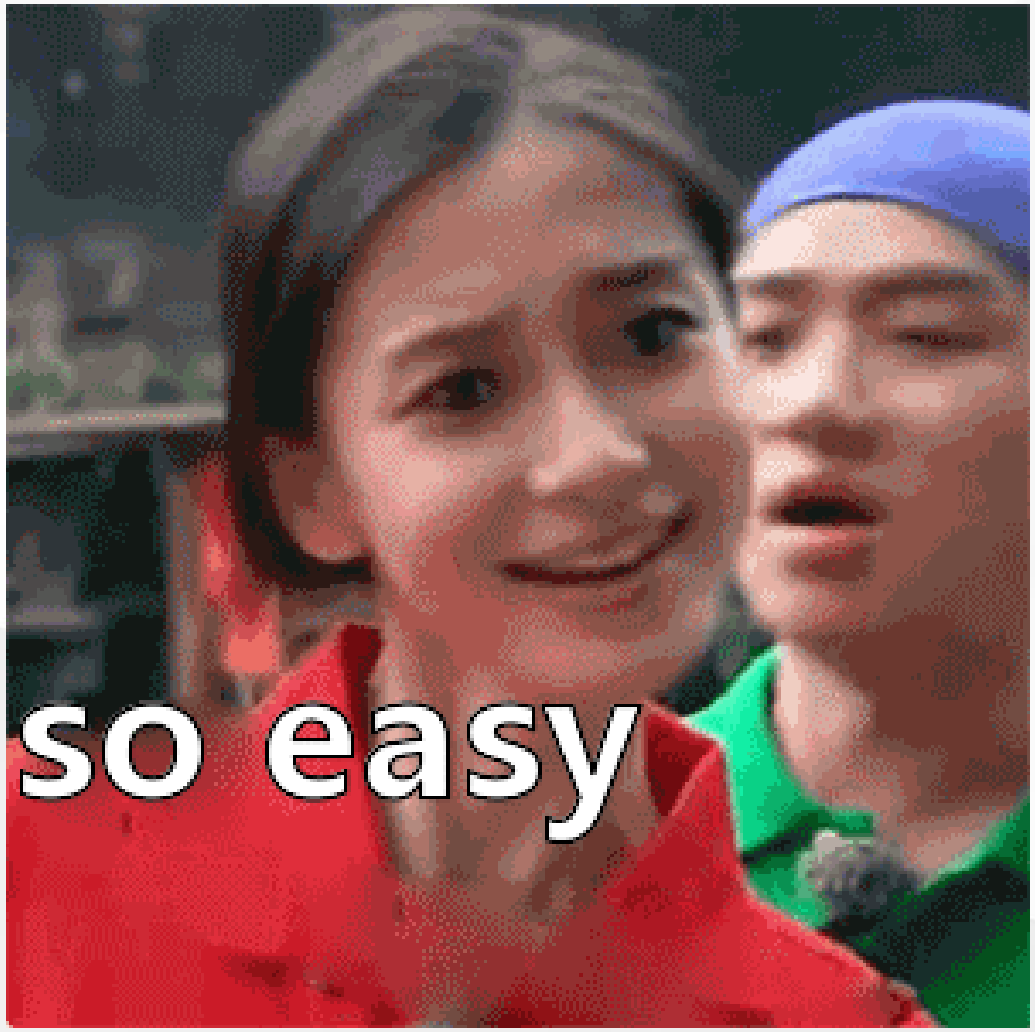
*Correct Way of utilizing overriding

Suppose we want to compose and draw a car:

```
Circle wheel1, wheel2;
Rect body;
Trapezium top;

wheel1.draw();
wheel2.draw();
body.draw();
top.draw();
```

Question: What if this vehicle(group) has hundreds of shapes?


so easy

*Correct Way of utilizing overriding

```
Circle c[1000];
Rect r[1000];
Trapezium t[1000];
Quadrilateral q[1000];

for (int i = 0; i < n; i++) c[i].draw();
for ...
for ...
```

Well, much better...

But... it's still not called polymorphism...

*Correct Way of utilizing overriding

```
Figure* fig[1000];

for (int i = 0; i < n; i++) fig[i]->draw();
```

This is the purpose of using inheritance and `virtual`: write less!!

Even better: use a `vector` container to store the pointers.

Some other knowledge..

- inline function
- reference in C++
- ...

Finally...