组合与继承

1.组合(has-a)

两种方式实现组合：

- 已有类的对象作为新类的**公有数据成员**，这样通过允许直接访问子对象而"提供"旧类接口
- 已有类的对象作为新类的**私有数据成员**。新类可以**调整旧类的对外接口**，可以不使用旧类原有的接口（相当于对接口作了转换）

子对象构造时若需要参数，则应在当前类的构造函数的**初始化列表中进行**。若使用默认构造函数来构造子对象，则不用做任何处理。

对象组合示例

```
class C3 {
  int num;
  C1 sub_obj1;
  C2 sub_obj2;
public:
  C3() : num(0), sub_obj1(123) /// 构造函数初始化列表中构造子对象
        { cout << "C3()" << endl; }
  C3(int n) : num(n), sub_obj1(123)
        { cout << "C3(int)" << endl; }
  C3(int n, int k) : num(n), sub_obj1(k)
        { cout << "C3(int, int)" << endl; }
  ~C3() { cout << "~C3()" << endl; }
};
```

2. 继承（is-a）

如果类A具有类B全部的属性和服务，而且具有自己**特有的**某些属性或服务，则称A为B的**派生类**，B为A的**基类**。例如，"兔子"，"猫"是"动物"的**派生类**，继承"动物"类。

常见的继承方式：public, private

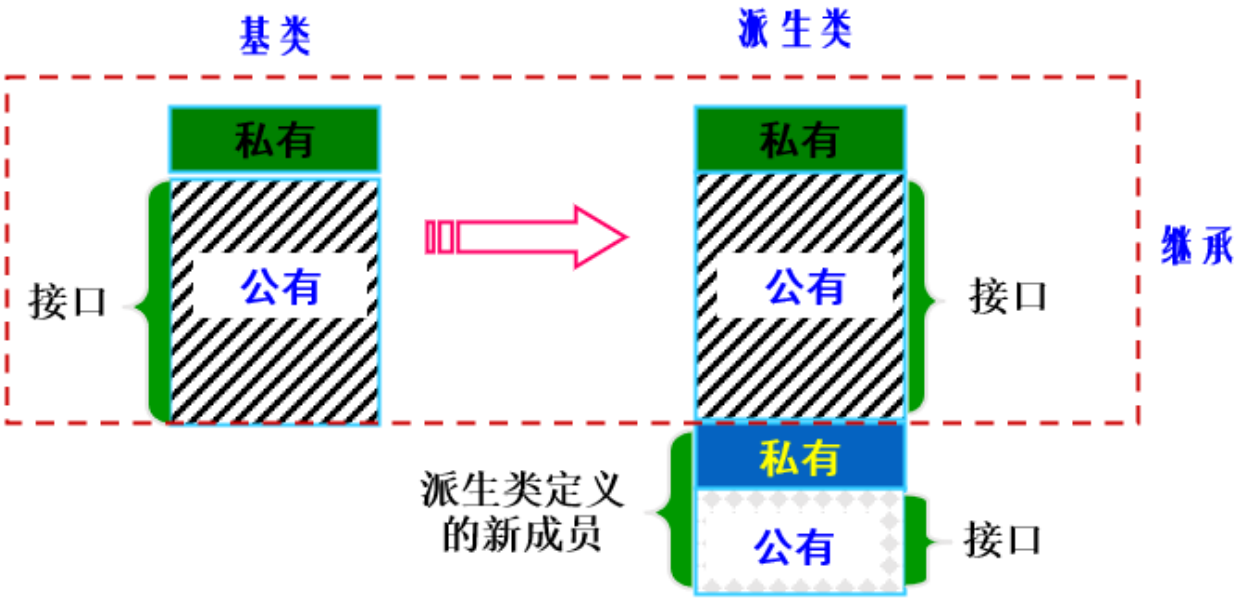
- `class Derived : [private] Base { .. };` 缺省继承方式为private继承。
- `class Derived : public Base { ... };` public 继承

基类中的数据成员，通过继承成为派生类对象的一部分，需要在构造派生类对象的过程中调用基类构造函数来正确初始化。

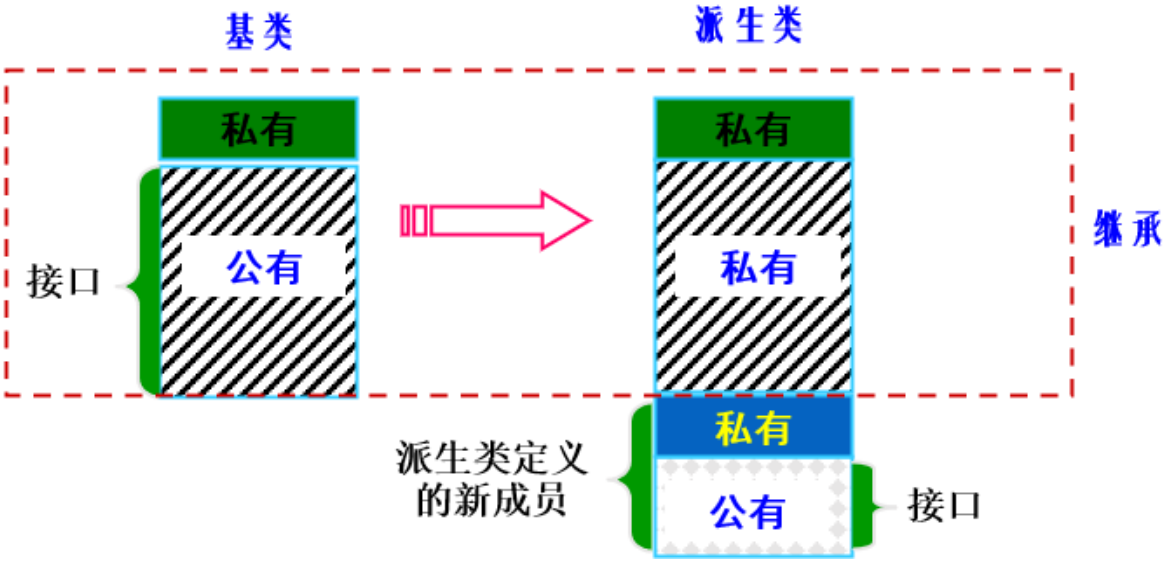
- 若没有显式调用，则编译器会自动生成一个对基类的**默认构造函数**的调用。
- 若想要显式调用，则只能在派生类构造函数的**初始化成员列表**中进行，既可以调用基类中不带参数的默认构造函数，也可以调用合适的带参数的其他构造函数。

```
class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; } /// 默认构造函数
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() {} /// 无显式调用基类构造函数，则调用基类默认构造函数
    Derive(int i) : Base(i) {} /// 显式调用基类构造函数，在初始化成员列表中进行
};
```

- public继承：基类中公有成员**仍能在派生类中保持公有。**原接口可沿用。最常用。



- private继承：通常不使用



成员的访问权限：

- 基类中的私有成员，不允许在派生类成员函数中访问，也不允许派生类的对象访问它们。真正体现“基类私有”，对派生类也不开放其权限！
- 基类中的公有成员：允许在派生类成员函数中被访问。若是使用public继承方式，则成为派生类公有成员，可以被派生类的对象访问；是使用private/protected继承方式，则成为派生类私有/保护成员，不能被派生类的对象访问。若想让某成员能被派生类的对象访问，可在派生类public部分用关键字using声明它的名字。
- 基类中的保护成员：与基类中的私有成员的不同在于：保护成员允许在派生类成员函数中被访问。

3. 重写隐藏和重载

重载(overload)：

- 目的：提供同名函数的不同实现，属于静态多态。
- 函数名必须相同，函数参数必须不同，作用域相同（如位于同一个类中）。

重写隐藏(redefining)：

- 目的：在派生类中重新定义基类函数，实现派生类的特殊功能。
- **屏蔽**了基类的所有其它同名函数。
- 函数名必须相同，函数参数可以不同

重写隐藏发生时，基类中该成员函数的其他重载函数都将**被屏蔽掉**，不能提供给派生类对象使用

可以在派生类中通过using 类名::成员函数名; 在派生类中“恢复”指定的基类成员函数（即去掉屏蔽），使之重新可用

函数重写隐藏示例

```
#include <iostream>
using namespace std;
class T {};
class B {
public:
  void f() { cout << "B::f()\n"; }
  void f(int i) { cout << "B::f(" << i << ")\n"; } /// 重载
  void f(double d) { cout << "B::f(" << d << ")\n"; } ///重载
  void f(T) { cout << "B::f(T)\n"; } ///重载
};
class D1 : public B {
public:
  void f(int i) { cout << "D1::f(" << i << ")\n"; } ///重写隐藏
};
int main() {
  D1 d;
  d.f(10);
  d.f(4.9);        /// 编译警告。执行自动类型转换。
  //  d.f();       /// 被屏蔽，编译错误
  //  d.f(T());    /// 被屏蔽，编译错误
  return 0;
}
```

4.9 → 4

运行结果

```
D1::f(10)
D1::f(4)
```

恢复基类成员函数示例

```
#include <iostream>
using namespace std;
class T {};
class B {
public:
  void f() { cout << "B::f()\n"; }
  void f(int i) { cout << "B::f(" << i << ")\n"; }
  void f(double d) { cout << "B::f(" << d << ")\n"; }
  void f(T) { cout << "B::f(T)\n"; }
};
class D1 : public B {
public:
  using B::f;
  void f(int i) { cout << "D1::f(" << i << ")\n"; }
};
int main() {
  D1 d;
  d.f(10);
  d.f(4.9);
  d.f();
  d.f(T());
  return 0;
}
```

使用using 基类名::函数名;恢复基类函数

运行结果

```
D1::f(10)
B::f(4.9)
B::f()
B::f(T)
```

4. 向上类型转换

派生类对象/引用/指针转换成**基类**对象/引用/指针，称为向上类型转换。只对public继承有效，在继承图上是上升的；对private、protected继承无效。向上类型转换（派生类到基类）可以由编译器自动完成，是一种隐式类型转换。

凡是接受基类对象/引用/指针的地方（如函数参数），都可以使用派生类对象/引用/指针，编译器会自动将派生类对象转换为基类对象以便使用。

示例

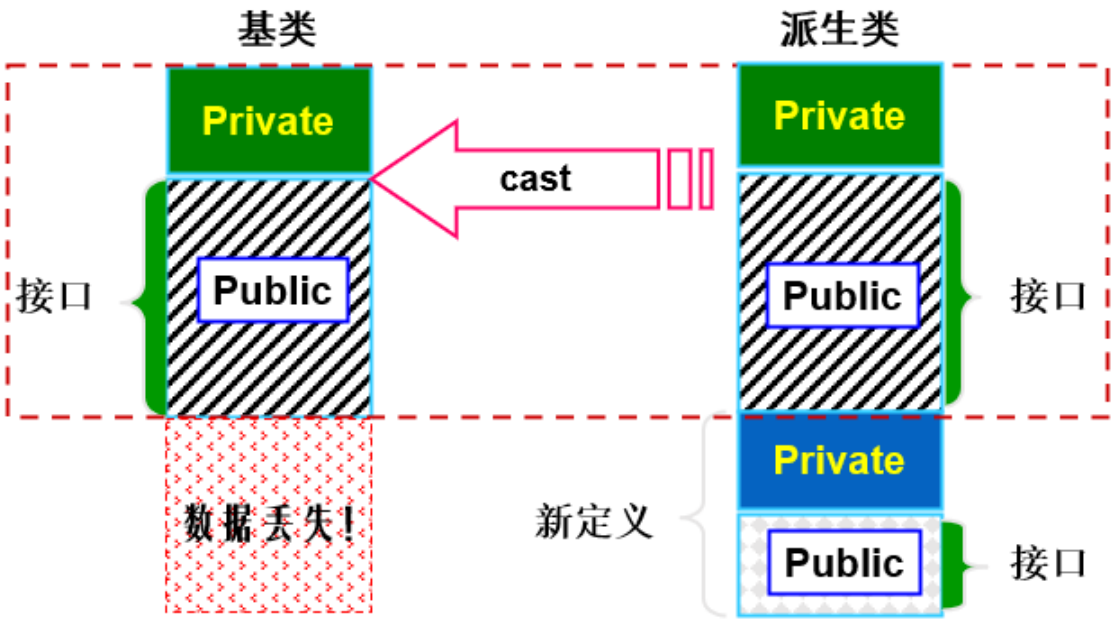
```
#include <iostream>
using namespace std;

class Instrument {
public:
  void play() { cout << "Instrument::play" << endl; }
};

class Wind : public Instrument {
public:
  // Redefine interface function:
  void play() { cout << "Wind::play" << endl; }
};

void tune(Instrument& i) {
  i.play();
}

int main() {
  Wind flute;
  tune(flute); /// 引用的向上类型转换
}
```

运行结果

```
Instrument::play
```

5. 对象切片

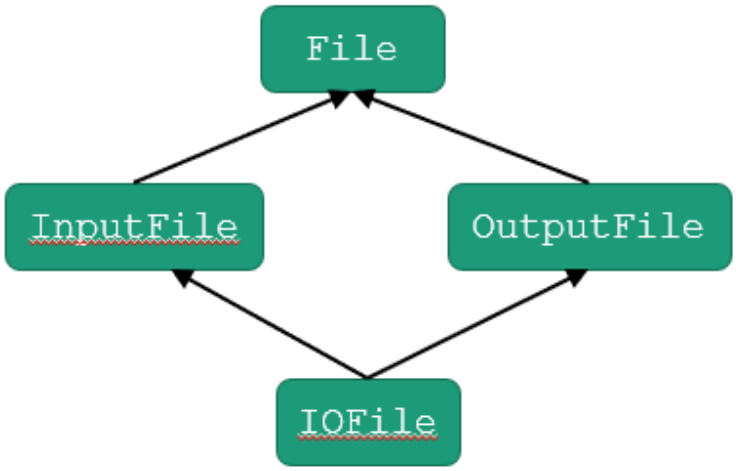
当派生类的**对象**(不是指针或引用)被转换为基类的对象时，派生类的对象被切片为对应基类的子对象。



因此，设计原则：只对引用和指针进行向上转换，避免对象切片

6. 多重继承

```
class File{};
class InputFile: public File{};
class OutputFile: public File{};
class IOFile: public InputFile, public OutputFile{};
```



数据存储

- 如果派生类D继承的两个基类A,B，是同一基类Base的不同继承，则A,B中继承自Base的数据成员会在D有两份独立的副本，可能带来**数据冗余**。

二义性

- 如果派生类D继承的两个基类A,B，有同名成员a，则访问D中a时，编译器无法判断要访问的哪一个基类成员。