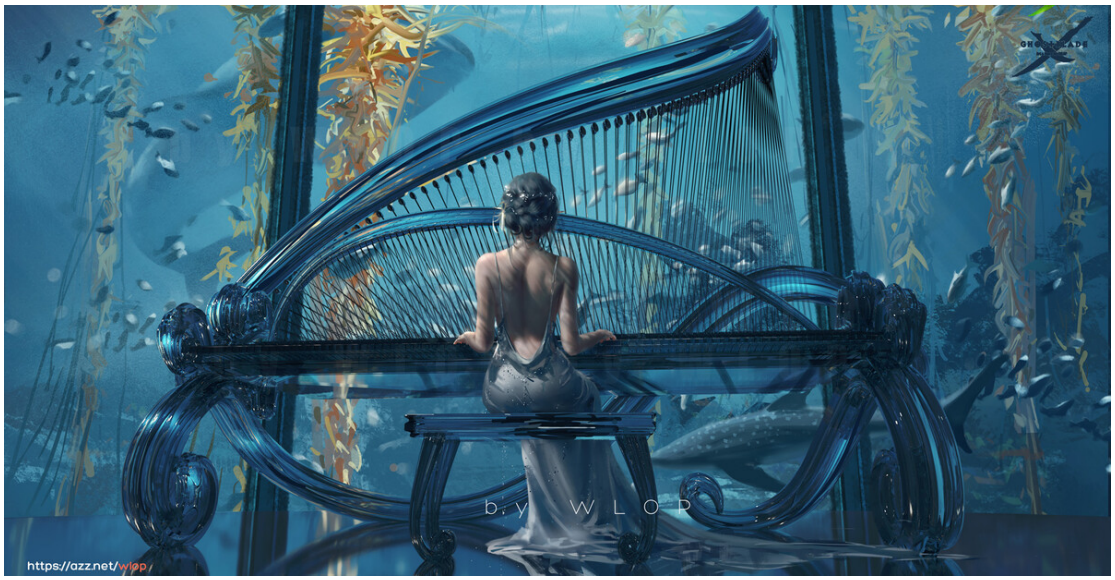


虚函数

虚函数

By 曹菡雯（计 03）、赵晨阳（计 06）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分难度不大，并没有太多扩展内容。主要是 L8-虚函数的课堂笔记整理，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于每一页 PPT 都有完整的解读，同时联系了前后课程的内容与课程作业，对于课件中的操作也有一定的扩展。

如果阅读时间不够充足，建议阅读课堂的扩展部分。

2.3 对象切片的理解

2.4 对象切片从内存空间角度的进一步讨论

3.3 虚函数表详解

最近一次更新: 2021 年 5 月 13 日，修改了在 2.3 节对于切片的错误定义。

一、向上类型转换

1.1 定义

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。

1.2 不允许非 public 继承的向上转换

```
#include <iostream>
using namespace std;
class Base {
private:
   int data{0};
public:
   int getData(){ return data;}
   void setData(int i){ data=i;}
};
class Derive1 : private Base {
public:
   using Base::getData;
};
int main() {
```

```
    Derive1 d1;
    cout<<d1.getData();
    Base& b = d1;       ///不允许私有继承的向上转换
    b.setData(10);   ///否则可以绕过 D1，调用基类的 setData 函数
    return 0 ;
}
Output：
error: cannot cast 'Derive1' to its private base class 'Base'
    Base& b = d1;       ///不允许私有继承的向上转换
```

如果 private 继承也可以向上转换，那么基类对象里的 public 成员也可以被派生类调用了。而 private 继承的原意就是让基类成员在派生类里变成 private 属性，所以 private 继承不能向上转换。

1.3 例子

```
#include <iostream>
using namespace std;
class Base {
public:
  void print() { cout << "Base::print()" << endl; }
};
class Derive : public Base {
public:
  void print() { cout << "Derive::print()" << endl; }
};
void fun(Base obj) { obj.print(); }
int main()
{
  Derive d;
  d.print();
  fun(d);      /// 本意：希望对 Drive::print 的调用
  return 0;
}
Output：
Derive::print()
Base::print()
```

派生类会覆盖掉基类的同名函数，未发生切片前，想要让派生类使用基类的 print，方法一是使用 using 关键字，方法二是启用命名空间。

二、对象切片

2.1 定义

当派生类的对象 b(不是指针或引用)被转换为基类的对象 a 时，派生类的对象被切片为对应基类的子对象。这句话的意义是 a 仅仅只能含有 b 中基类的部分，而 b 对象本身不受到影响。

切片过程不可逆，即便切片本身不改变指针的位置，改变了指针的类型，这一过程也不可逆。（见下文）

2.2 例子

例一、存储空间的变化

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

#pragma pack(4)//  按照四字节进行内存对齐

class Pet {
   public:  int att_i;//表示属性
   Pet(int x=0): att_i(x) {};
};
class Dog: public Pet {
   public:  int att_j;
   Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
void getSize(Pet p){
  cout << "Pet size:" << sizeof(p) << endl;
}
int main() {
  Pet p;
  cout << "Pet size:" << sizeof(p) << endl;
  Dog g;
  cout << "Dog size:" << sizeof(g) << endl;

  getSize(g);  /// 对象切片(传参)，p 丢失了 g 的数据，但是 g 不受影响。

  cout << "Dog size:" << sizeof(g) << endl;

  p = g;       /// 对象切片(赋值)，p 丢失了 g 的数据，但是 g 不受影响。

  cout << "Dog size:" << sizeof(g) << endl;
  return 0;
}
Pet size:4
Dog size:8
Pet size:4
Dog size:8
Dog size:8
```

例二、派生类数据无权访问

```
#include <iostream>
using namespace std;
#pragma pack(4)

class Pet {
public:  int att_i;//表示属性
    Pet(int x=0): att_i(x) {}
};
class Dog: public Pet {
public:  int att_j;
    Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
int main() {
  Pet p(1);
  cout << p.att_i << endl;
  Dog g(2,3);
  cout << g.att_i << " " << g.att_j << endl;

  p = g;        /// 对象切片，只赋值基类数据

  cout << p.att_i << endl;

  //cout << p.att_j << endl;  // 没有该参数，编译错误

  return 0;
}
Output：
1
2 3
2
```

例三、派生类方法无权访问

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

class Pet {
public:
  void name(){ cout << "Pet::name()" << endl; }
};
class Dog: public Pet {
public:
  void name(){ cout << "Dog::name()" << endl; }
};
```

```
void getName(Pet p){
    p.name();
}
int main() {
  Dog g;
  g.name();
  getName(g);  /// 对象切片（传参），调用基类的 name 函数
  Pet p = g;
  p.name();  /// 对象切片（赋值），调用基类的 name 函数
  return 0;
}
Output：
Dog::name()
Pet::name()
Pet::name()
```

2.3 对象切片的理解

情况一：指针型向上类型转换

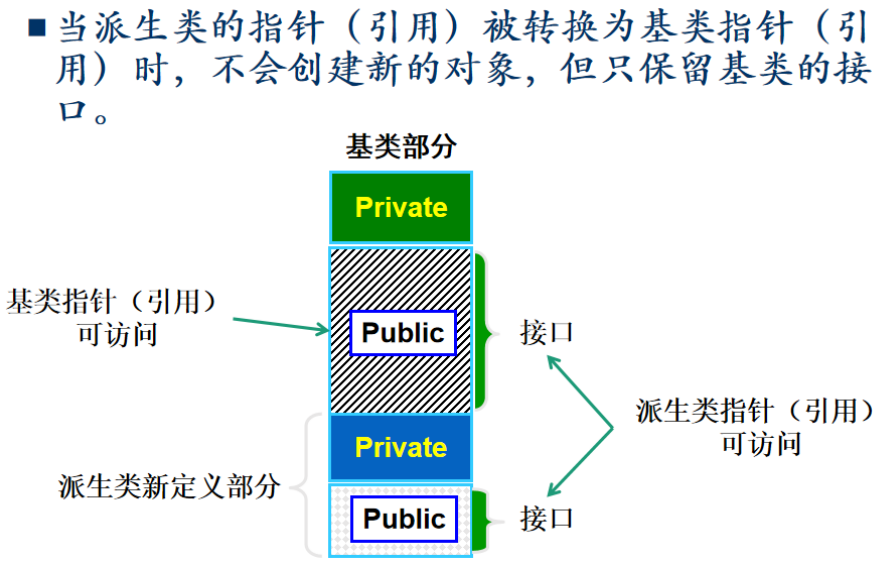
```
#include <iostream>
using namespace std;
#pragma pack(4)
class Pet {
 public:  int att_i;//表示属性
 Pet(int x=0): att_i(x) {};
};
class Dog: public Pet {
 public:  int att_j;
 Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
int main() {
 Dog* g = new Dog(1, 2);
 Pet* p = g;
 cout << g << ' ' << p << endl;
 int* a = (int*)g;
 cout << a << ' ' << a + 1 << endl;
 cout << *a << ' ' << *(a + 1) << endl;
 cout << p->att_i << endl;
 cout << g->att_i <<" "<<g->att_j << endl;
 return 0;
}
Output：
```

```
0x1c94eb0 0x1c94eb0
0x1c94eb0 0x1c94eb4
1 2
1
1 2
```

在主函数中，我们使用了基类指针 p 指向派生类对象 g。将 p 和 g 指向内存空间的地址打印出来，同时将 g 指向内存空间上的两个 int 地址打印出来，我们发现 g 和 p 都指向同一个地址（这个地址就是 att_i）。

实际上，数据在内存中是连续紧密排列的。先排列基类数据 att_i，再排列派生类数据 att_j。而我们用基类指针指向派生类对象时，基类和派生类指针都指向同一个内存地址，即基类数据的地址。但是访问权限不同，基类指针只能访问基类的部分，派生类能访问所有部分。情况一类型的数据丢失，并非派生类数据销毁了，而是被转换生成的指针没有访问派生类数据的权限。

不过，当多重继承时，比如 Class A 同时 Public 继承 B，C。(按照类的定义顺序从左到右)在内存中则先排列 B，之后 C，最后 A。还是使用基类指针指向派生类对象的话，如果 A 类指针 a 和 C 类指针 c 不重合。a 指向头部，c 指向 C 的部分，访问权限也不同。



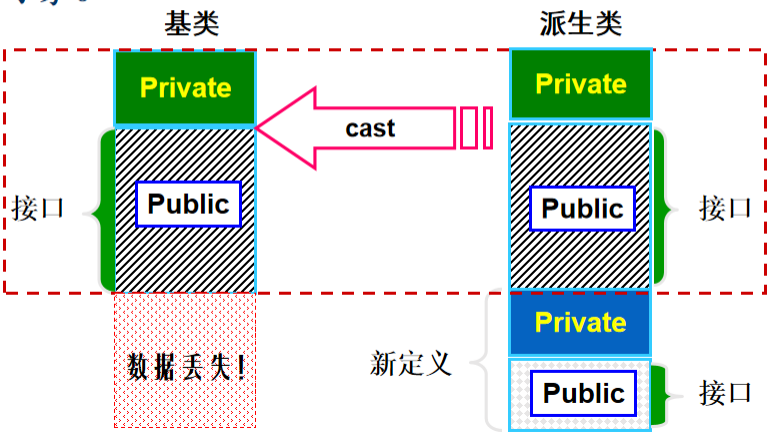
情况二：拷贝型向上类型转换

```
#include <iostream>
using namespace std;
#pragma pack(4)
class Pet {
 public:  int att_i;//表示属性
 Pet(int x=0): att_i(x) {};
};
class Dog: public Pet {
 public:  int att_j;
 Dog(int x=0, int y=0): Pet(x), att_j(y) {}
```

```
};
int main() {
  Dog d(10,12);
  Pet t;
  cout<<"locate of d: "<<&d<<endl;
  cout<<d.att_i<<" "<<d.att_j<<endl;
  cout<<"locate of t: "<<&t<<endl;
  cout<<t.att_i<<endl;
  t=d;
  cout<<t.att_i<<endl;
  cout<<"locate of t: "<<&t<<endl;
  return 0;
}
Output：
locate of d: 0x7ffefd9e81a0
10 12
locate of t: 0x7ffefd9e8198
0
10
locate of t: 0x7ffefd9e8198
```

可以发现，在定义了 Pet t 后，就已经为 t 分配了内存空间，和 d 并不是同一块空间。接下来的 t=d 类似于一个部分拷贝赋值语句，并没有改变 t 和 d 的地址。比起指针型的切片简单许多。

- 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。



2.4 进一步讨论

编译器为 glot 平台

```
#include <iostream>
using namespace std;
#pragma pack(4)
```

```
class Pet {
 public:  int att_i;//表示属性
 Pet(int x=0): att_i(x) {};
};
class Dog: public Pet {
 public:  int att_j;
 Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
void getSize(Pet p){
    cout<<"locate of p: "<<&p<<endl;
  cout << "Pet size:" << sizeof(p) << endl;
}
int main() {
 Dog* g = new Dog(1, 2);
 cout<<"locate of g: "<<g<<endl;
 Pet* j=g;
  cout<<"locate of j: "<<j<<endl;
 getSize(*g);
 cout<<g->att_i<<endl;
  delete j;
  cout<<g->att_i;
 return 0;
}
Output：
locate of g: 0x18beeb0
locate of j: 0x18beeb0
locate of p: 0x7ffe369b8848
Pet size:4
1
6334（实际上是非法访问，不同编译器行为不同）
```

这里对于拷贝型向上类型转换和指针型向上类型转换进行分析。思考一个问题，拷贝型向上类型转换是否会在参数被析构后破坏原有的派生类数据？

答案是不会的。

上述例子通过传递参数参数产生的切片实质是拷贝传参，相当于用派生类对象在另外一块地址上拷贝了一个基类对象。这一对象指针指针指向的内存和派生类指针指向的内存当然不是同一块，故而析构的时候不会有问题。

而对于指针型向上类型转换，在上文已经讨论过这一切片的本质——两个指向地址相同的指针，但是访问权限不同。

如同上文输出所示，情况一类型的切片生成的基类指针和派生类指针指向同一块空间。将基类指针所指空间释放后，破坏了派生类数据。

又例如：

```
#include <iostream>
using namespace std;
#pragma pack(4)

class Pet {
public:  int att_i;
    Pet(int x=0): att_i(x) {}
};
class Dog: public Pet {
public:  int att_j;
    Dog(int x=0, int y=0): Pet(x), att_j(y) {}
};
int main() {
  Dog g(2,3);
  cout << g.att_i << " " << g.att_j << endl;
  Pet& p = g;       /// 引用向上转换
  cout << p.att_i << endl;
  p.att_i = 1;       /// 修改基类存在的数据
  cout << p.att_i << endl;
  cout << g.att_i << " " << g.att_j << endl; /// 影响派生类
  return 0;
}
Output:
2 3
2
1
1 3
```

课堂上的例子只强调了基类引用会影响派生类数据，实际上应结合地址和内存空间来理解。

```
#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(传参)，编译器早绑定，无对象切片产生

  Instrument &inst = flute;  /// 引用的向上类型转换(赋值)

  inst.play();
  return 0;
}
Output：
Instrument::play
Instrument::play
```

同样的，Ins 类的引用的权限不足以访问 Wind 的 public 接口。

最后，指针的基本知识：指针的值是指指针所指向的地址。而指针本身是一种数据结构，也需要内存空间存放，指针的地址就是存放指针的内存空间的地址。比如 g 是 dog 类型的指针

```
cout<<"locate of g: "<<g<<endl;
```

这是打印 g 指向的内存空间的地址。

```
cout<<"locate of g: "<<&g<<endl;
```

这是存储 g 的内存空间的地址。

而对于变量，例如 Pet p，&取地址符即可输出地址。

```
   cout<<"locate of p: "<<&p<<endl;
```

三、虚函数

3.1 函数调用绑定

把函数体与函数调用相联系称为捆绑(binding)。即将函数体的具体实现代码，与调用的函数名绑定，执行到调用代码时进入直接进入捆绑好的函数体内部。

当捆绑在程序运行之前（由编译器和连接器）完成，也即是运行之前已经决定了函数究竟调用哪个函数，称为早捆绑(early binding)。

例如：

```
#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(传参)，编译器早绑定，无对象切片产生

  Instrument &inst = flute;  /// 引用的向上类型转换(赋值)

  inst.play();
  return 0;
}
Output：
Instrument::play
Instrument::play
```

此处 inst 是 flute 的引用，由于发生切片，编译器将 tune 中的函数调用 i.play()与 Instrument::play()绑定。

当捆绑根据对象的实际类型(上例中即子类 Wind 而非 Instrument)，发生在程序运行时，称为晚捆绑(late binding)，又称动态捆绑或运行时捆绑。

也即是要求在运行时能确定对象的实际类型，并绑定正确的函数。

晚捆绑只对类中的虚函数起作用，使用 virtual 关键字声明虚函数。

3.2 虚函数定义

对于被派生类重新定义的成员函数，若它在基类中被声明为虚函数（如下所示），则通过基类指针或引用调用该成员函数时，编译器将根据所指（或引用）对象的实际类型决定是调用基类中的函数，还是调用派生类重写的函数。

```
class Base {
 public:
  virtual ReturnType FuncName(argument); //虚函数
  ...
};
```

若某成员函数在基类中声明为虚函数，当派生类重写覆盖它时(同名，同参数函数)，无论是否声明为虚函数，该成员函数都仍然是虚函数。

（区分重写覆盖和重写隐藏：重写隐藏（redefining）是在派生类中重新定义同名、参数可能不同的函数，并屏蔽了基类的所有同名函数；重写覆盖则是

一种特殊的重写隐藏，要求不仅同名，参数也要相同，是一个针对虚函数的概念）

例一：

```
#include <iostream>
using namespace std;
class Instrument {
public:
  virtual void play() { cout << "Instrument::play" << endl; }
};
class Wind : public Instrument {
public:
  void play() { cout << "Wind::play" << endl; }
    /// 重写覆盖(稍后：重写隐藏和重写覆盖的区别)
};
void tune(Instrument& ins) {
 ins.play(); /// 由于 Instrument::play 是虚函数，编译时不再直接绑定，运行时根据
 ins 的实际类型调用。
}
int main() {
  Wind flute;
  tune(flute); /// 向上类型转换
  return 0;
}
Output：
Wind::play
```

注意到上方 void tune(Instrument& ins) ;的参数是引用（或指针），才成功起到了晚绑定效果。如果参数是 void tune(Instrument ins)，那么发生拷贝型切片，依然无法晚绑定。

3.3 虚函数表

3.3.1 概述

为了实现 C++的多态，C++使用了一种动态绑定的技术。这个技术的核心是虚函数表（简称虚表）。3.3 节介绍虚函数表是如何实现动态绑定的。

3.3.2 类的虚表

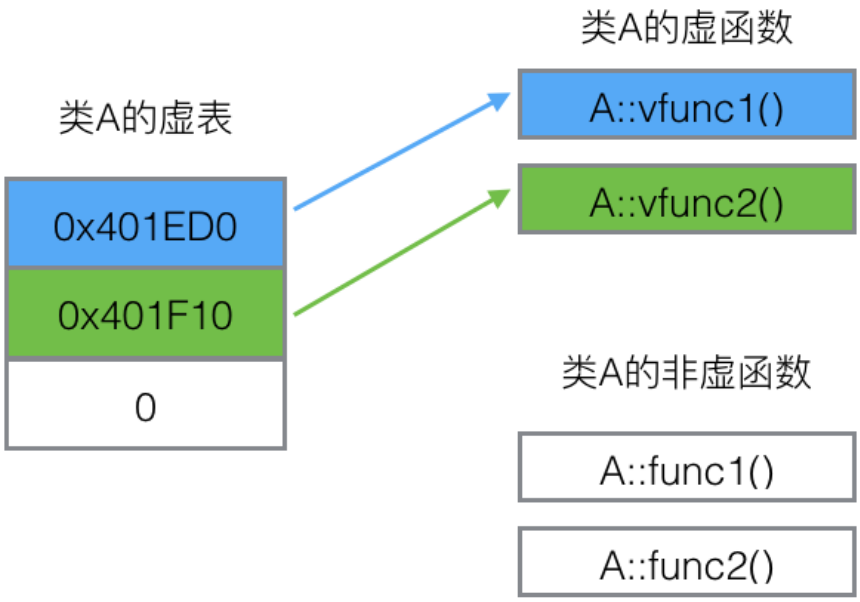
每个包含了虚函数的类都包含一个属于整个类的虚表。（只要有一个虚函数就行）

当一个类（A）继承另一个类（B）时，类 A 会继承类 B 的函数的调用权。所以如果一个基类包含了虚函数，那么其继承类也可调用这些虚函数，换句话说，一个类继承了包含虚函数的基类，那么这个类也拥有自己的虚表。

例如：

```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
```

类 A 包含虚函数 vfunc1，vfunc2，由于类 A 包含虚函数，故类 A 拥有一个虚表。



虚表是一个指针数组，其元素是虚函数的指针，每个元素对应一个虚函数的函数指针。需要指出的是，普通的函数即非虚函数，其调用并不需要经过虚表，所以虚表的元素并不包括普通函数的函数指针。

虚表内的条目，即虚函数指针的赋值发生在编译器的编译阶段，也就是说在代码的编译阶段，虚表就完成了构造。

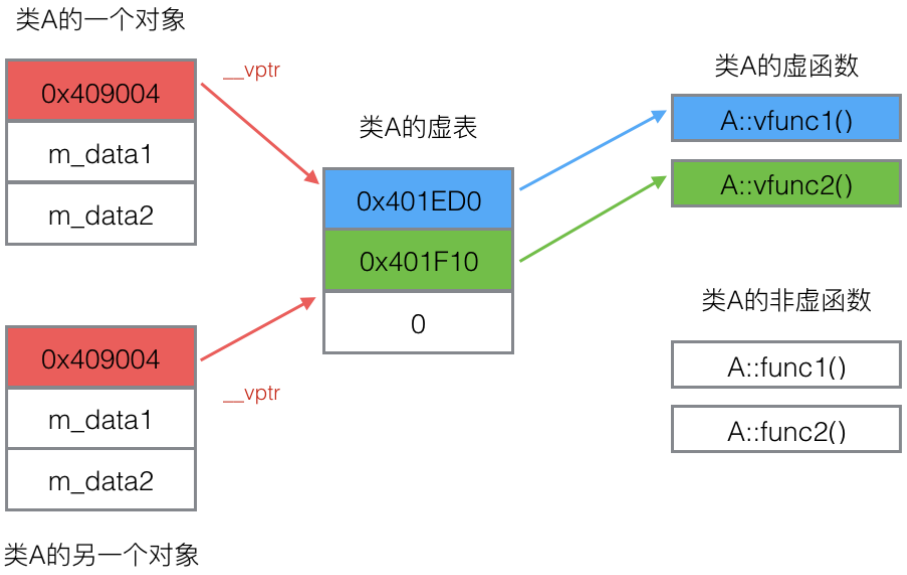
3.3.3 虚表指针

虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可，同一个类的所有对象都使用同一个虚表。（但另一方面，不同的类的虚表不同，即使它们有继承关系）

每一个虚函数类对象内部包含一个指向所属类的虚表的指针，来指向自己

所使用的虚表。为了让每个包含虚表的类的对象都拥有一个虚表指针，编译器在类中隐式添加了一个指针*vptr，用来指向虚表。（类似于*this。）这样，当类的对象在创建时便拥有了这个指针，且这个指针的值会自动被设置为指向类的虚表。

进一步而言，对象的虚表指针用来指向自己所属类的虚表，虚表中的函数指针会指向其继承的最近的一个类的虚函数与自身重写覆盖的虚函数。

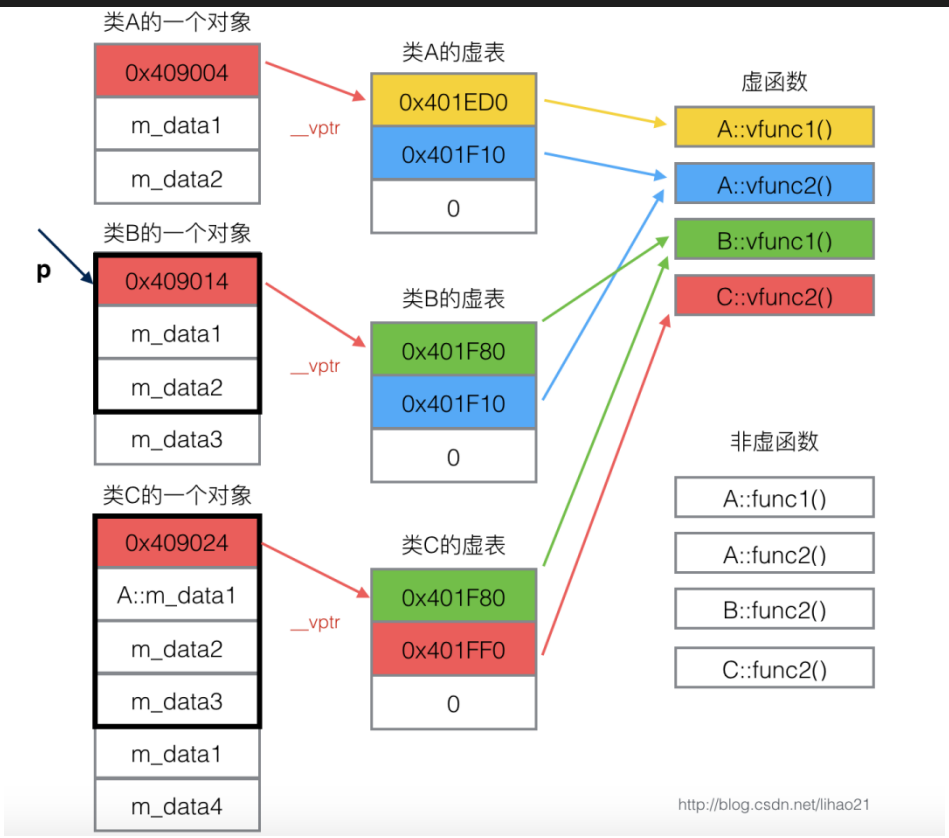


前文指出，一个继承类的基类如果包含虚函数，那个这个继承类也有拥有自己的不同的虚表。这个继承类的对象也包含一个虚表指针，用来指向它的虚表。

3.3.4 动态绑定

```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
class B : public A {
public:
    virtual void vfunc1();
    void func1();
private:
    int m_data3;
};
class C: public B {
public:
    virtual void vfunc2();
    void func2();
```

```
private:
    int m_data1, m_data4;
};
```



类 A 是基类，类 B 继承类 A，类 C 又继承类 B。由于这三个类都有虚函数，故编译器为每个类都创建了一个虚表，即类 A 的虚表（A vtbl），类 B 的虚表（B vtbl），类 C 的虚表（C vtbl）。类 A，类 B，类 C 的对象都拥有一个虚表指针，*__vptr，用来指向自己所属类的虚表。

注意到前文所述：对象的虚表指针用来指向自己所属类的虚表，虚表中的函数指针会指向其继承的最近的一个类的虚函数与自身重写覆盖的虚函数。

类 A 包括两个虚函数，故 A vtbl 包含两个指针，分别指向 A::vfunc1()和 A::vfunc2()。

类 B 继承于类 A，故类 B 可以调用类 A 的函数，但由于类 B 重写了 B::vfunc1()函数，故 B vtbl 的两个指针分别指向 B::vfunc1()和 A::vfunc2()。

类 C 继承于类 B，故类 C 可以调用类 B 的函数，但由于类 C 重写了 C::vfunc2()函数，故 C vtbl 的两个指针分别指向 B::vfunc1()（指向继承的最近的一个类的函数）和 C::vfunc2()。

总结而言，类 X 的虚函数表的函数指针指向继承链上所有没有发生重写覆盖的虚函数和被 X 类重写覆盖的虚函数。

而每一个类 X 的对象都有一个独属于自身的虚表指针，来指向 X 类的虚表。注意到，对象 y 的虚表指针位于基类部分，指向 y 的内存空间的头部。结合到前文叙述的“继承类的内存结构”，无论用哪一个基类的指针去指向一个派生类对象，基类指针都能有权访问派生类对象内存空间的头部，都有权访问派生类对象的虚函数指针，进而能够完全访问派生类的虚函数表。

这也可以解释，为什么只有指针型的切片能够实现晚绑定。因为拷贝型的切片无法访问原对象的内存空间头部，而是将这片内存空间上的数据在另一片内存空间拷贝了一番。

例如：

```
class A {
public:
    virtual void vfunc1();
    virtual void vfunc2();
    void func1();
    void func2();
private:
    int m_data1, m_data2;
};
class B : public A {
public:
    virtual void vfunc1();
    void func1();
private:
    int m_data3;
};
class C: public B {
public:
    virtual void vfunc2();
    void func2();
private:
    int m_data1, m_data4;
};
int main()
{
    B bObject;
    A *p = & bObject;
    p->vfunc1();
}
```

程序在执行 p->vfunc1()时，检测到 p 是个指针（已经无需在意是什么类的指针了），且调用的函数是虚函数。 首先，根据虚表指针 p->__vptr 来访问对象 bObject 对应的虚表。虽然指针 p 是基类 A*类型，但是*__vptr 也是基类的一部分，所以可以通过 p->__vptr 可以访问到对象对应的虚表。

然后，在虚表中查找所调用的函数对应的条目。由于虚表在编译阶段就已完成构造，所以可以根据所调用的函数定位到虚表中的对应条目。p->vfunc1()的调用对应 B vtbl 的第一项，即是 B::vfunc1 对应的条目。

最后，根据虚表中找到的函数指针，调用函数。从图 3 可以看到，B vtbl 的第一项指向 B::vfunc1()，所以 p->vfunc1()实质会调用 B::vfunc1()函数。

又比如：

```
int main()
{
    A aObject;
    A *p = &aObject;
    p->vfunc1();
}
```

p 指向类 A 的对象，当 aObject 在创建时，它的虚表指针__vptr 已设置为指向 A vtbl，这样 p->__vptr 就指向 A vtbl。vfunc1 在 A vtbl 对应在条目指向了 A::vfunc1()函数，所以 p->vfunc1()实质会调用 A::vfunc1()函数。

3.3.5 动态绑定的条件

通过指针来调用函数

指针 upcast 向上类型转型

调用的是虚函数

如果一个函数调用符合以上三个条件，编译器就会把该函数调用编译成动态绑定，其函数的调用过程走的是上述通过虚表的机制。

3.3.6 虚指针大小

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

#pragma pack(4) //按照 4 字节进行内存对齐

class NoVirtual{ //没有虚函数
  int a;
public:
  void f1() const {}
  int f2() const {return 1;}
};
class OneVirtual{ //一个虚函数
  int a;
public:
  virtual void f1() const {}
  int f2() const {return 1;}
};
class TwoVirtual{//两个虚函数
  int a;
public:
  virtual void f1() const {}
  virtual int f2() const {return 1;}
};
int main(){

    cout<<"int: "<<sizeof(int)<<endl;
```

```
    cout<<"NoVirtual: "<<sizeof(NoVirtual)<<endl;

    cout<<"void* : "<<sizeof(void*)<<endl;

    cout<<"OneVirtual: "<<sizeof(OneVirtual)<<endl;

    cout<<"TwoVirtual: "<<sizeof(TwoVirtual)<<endl;
return 0;
}
Output：
int: 4
NoVirtual: 4
void* : 8
OneVirtual: 12
TwoVirtual: 12
```

可以观察到，虚指针的大小为 8 字节，且虚指针只是指向了虚表，虚函数的个数只是影响虚表的大小，不影响虚指针的大小。

3.3.7 总结

通过使用这些虚函数表，即使使用的是基类的指针来调用函数，也可以达到正确调用运行中实际对象的虚函数。

把经过虚表调用虚函数的过程称为动态绑定，其表现出来的现象称为运行时多态。动态绑定区别于传统的函数调用，传统的函数调用我们称之为静态绑定，即函数的调用在编译阶段就可以确定下来。封装，继承，多态是面向对象设计的三个特征，而多态是面向对象设计的关键。C++通过虚函数表，实现了虚函数与对象的动态绑定，从而构建了 C++面向对象程序设计的基石。

3.4 虚函数与构造、析构

3.4.1 虚函数与构造函数

当创建一个包含有虚函数的对象时，必须初始化它的 VPTR 以指向相应的 VTABLE。设置 VPTR 的工作由构造函数完成。编译器在构造函数的开头秘密的插入能初始化 VPTR 的代码。

构造函数不能也不必是虚函数。

不能：如果构造函数是虚函数，则创建对象时需要先知道 VPTR，而在构造函数调用前，VPTR 未初始化。

不必：构造函数的作用是提供类中成员初始化，调用时明确指定要创建对象的类型，没有必要是虚函数。

例一

```
#include <iostream>
using namespace std;
class Base {
```

```
public:
virtual void foo(){cout<<"Base::foo"<<endl;}
Base(){foo();}  ///在构造函数中调用虚函数 foo

void bar(){foo();}; ///在普通函数中调用虚函数 foo

};
class Derived : public Base {
public:
  int _num;
  void foo(){cout<<"Derived::foo "<<_num<<endl;}
    Derived(int j):Base(),_num(j){}
};
int main() {
    Derived d(0);
    Base &b = d;
    b.bar();
    b.foo();
    return 0;
}
Output：
Base::foo
Derived::foo 0
Derived::foo 0
```

注意到，构造函数里调用的是 Base::foo()，因为初始化列表是按照定义的先后顺序来初始化的，与初始化列表写法先后顺序无关。而我们已知，初始化是先初始化基类部分，再初始化派生类部分。故而 Derived 的初始化列表中调用 Base 来进行构造时，一定会先调用 Base 的本地版本的 foo，因为_num 还未被初始化。不过，如果将_num 去掉又会如何？

```
#include <iostream>
using namespace std;
class Base {
public:
virtual void foo(){cout<<"Base::foo"<<endl;}
Base(){foo();}  ///在构造函数中调用虚函数 foo

void bar(){foo();}; ///在普通函数中调用虚函数 foo

};
class Derived : public Base {
public:
  void foo(){cout<<"Derived::foo"<<endl;}
    Derived(int j):Base(){}
};
```

```
int main() {
    Derived d(0);
    return 0;
}
Output：
Base::foo
```

依然只是调用了本地版本的 foo 函数，因为在构造函数中，基类中调用虚函数如果允许多态特性的话，就会使用派生类中还没有构造的部分，这有风险。故而 C++11 禁止了在未完成构造之前启用多态特性。

退回到例一。在构造 d 之后，b.bar()调用之后实际上为 b.foo()，完成虚函数匹配调用，故而调用了 Derived::foo();

总结：

在构造函数中调用一个虚函数，被调用的只是这个函数的本地版本(即当前类的版本)，即虚机制在构造函数中不工作。

构造函数语句的执行顺序：(与构造函数初始化列表顺序无关)

基类初始化

对象成员初始化

构造函数体

3.4.2 虚函数与析构函数

析构函数能是虚的，且常常是虚的。虚析构函数仍需定义函数体。并且基类的析构函数是虚的，那么派生类的析构函数也是虚的。（这和重写覆盖要求同名同参数并不一样）

虚析构函数的用途：当删除基类对象指针时，编译器将根据指针所指对象的实际类型，调用相应的析构函数。

若基类析构不是虚函数，则删除基类指针所指派生类对象时，编译器仅自动调用基类的析构函数，而不会考虑实际对象是不是基类的对象。这可能会导致内存泄漏。

在析构函数中调用一个虚函数，被调用的只是这个函数的本地版本，即虚机制在析构函数中不工作。理由和构造函数类似，因为派生类先被析构。如果在基类的析构里允许启用多态性，可能就会访问到已经被派生类析构函数释放的内存，存在风险。

可以进一步这么理解：派生类的析构函数其实是只析构派生类多出的部分。更具体的细节是虚指针指向的子类虚函数表已经被释放了，会产生内存不安全，所以不允许析构函数启用多态性。（这个更具体的细节助教老师说有待商榷）

```
#include <iostream>
using namespace std;
class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};
```

```
class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};
class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};
int main() {
  Base1* bp = new Derived1;

  delete bp; /// 只调用了基类的析构函数

  Base2* b2p = new Derived2;

  delete b2p; /// 派生类虚析构函数调用完后调用基类的虚析构函数

  return 0;
}
Output：
~Base1()
~Derived2()
~Base2()
```

有的时候我们可能会说，析构函数中不能调用虚函数，这句话是对的。这句话实际上是指，析构函数和构造函数中都没有多态性，但是还是可以完成调用本地版本。

总之，重要原则：总是将基类的析构函数设置为虚析构函数。

3.5 overload, override, redefining

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;或同名全局函数）

重写隐藏(redefining)：
目的：在派生类中重新定义非虚基类函数，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。

重写覆盖(override):

目的: 对基类的虚函数进行重定义，函数名和参数完全相同，返回值一般相同。（如果派生类定义新函数与基类虚函数的函数名相同而参数不同，派生类定义的函数会 redfining 基类虚函数，发生隐藏而不是覆盖。）

覆盖的由来: 基类虚函数表中的虚函数指针会被派生类中重新定义的中同名同参数的虚函数的指针覆盖掉，以此来实现动态绑定。

总结：**非虚同名函数在同一个类中发生重载，在不同类（基类和派生类）中发生隐藏**。

虚同名函数，参数相同则在派生类中发生重写覆盖，参数不同则在派生类中发生重写隐藏。

重载：同级调用的优先匹配问题。

重写隐藏：派生类优先调用派生类同名函数。

重写覆盖: 无论是基类还是派生类的指针，指向派生类对象则只能调用派生类同名函数。（详细的总结见 L7 • 组合与继承的 3.3 节）

例一

```
#include <iostream>
using namespace std;
class Base{
public:
   virtual void foo(){cout<<"Base::foo()"<<endl;}
   virtual void foo(int ){cout<<"Base::foo(int )"<<endl;}///重载
   void bar(){};
};

class Derived1 : public Base {
public:
   void foo(int){cout<<"Derived1::foo(int )"<<endl;} /// 是重写覆盖
};
class Derived2 : public Base {
public:
   void foo(float ) {cout<<"Derived2::foo(float )"<<endl;}
   /// 同名不同参，不是重写覆盖，是重写隐藏

};
int main() {
      Derived1 d1;
      Derived2 d2;
      Base* p1 = &d1;
      Base* p2 = &d2;
      Dervived2 &tag=d2;
      Derived2 *pointer=&d2;
```

```
    d1.Base::foo();
    d2.Base::foo();
    p1->foo();
    p2->foo();
    d1.foo(3); ///派生类定义同名同参数虚函数，发生重写覆盖。我们通常意义
上的重写覆盖是希望基类指针调用派生类函数，此处直接用派生类对象，那必然
调用派生类函数
    d2.foo(3.0);  ///调用的是派生类 foo(float )
    p1->foo(3);  ///重写覆盖，虚函数表中是派生类的 foo(int )
    p2->foo(3.0);
    d2.foo(3.0);
    tag.foo(3.0);
    pointer->foo(3.0);
  return 0;
  }
output:
Base::foo()
Base::foo()
Base::foo()
Base::foo()
Derived1::foo(int )
Derived2::foo(float )
Derived1::foo(int )
Base::foo(int )
Derived2::foo(float )
Derived2::foo(float )
Derived2::foo(float )
```

首先，d1.foo();与 d2.foo();都不合法，由于派生类都定义了带参数的 foo，基类 foo()对实例不可见，故而这两句会报错，但是我们可以用 Base::。

第二，p1->foo();时，派生类没有同名同参数覆盖掉基类虚函数，虚函数表继承基类的 foo()虚函数，故而调用了基类的 foo();

第三， p2->foo(3.0); 这里没有发生重写覆盖，发生了重写隐藏。在权限足够的时候，比如从派生类对象或者派生类指针可以直接访问优先级最高的同名函数。但是用基类指针的访问权限仅限于基类部分，无法访问定义在派生类部分的优先级最高的函数，故而只能访问基类定义的同名函数中匹配的一个。

例二

关于下列代码的说法正确的是

```
#include <iostream>
using namespace std;
class Base{
public:
    void foo(float){}          //(1)
    virtual void foo(){}       //(2)
    virtual void foo(int){}    //(3)
};
```

```
class Derived : public Base {
public:
    void foo() {}              //(4)
    virtual void foo(float){} //(5)
};
int main(){
    Derived d;
    d.foo(1);                  //(6)
    return 0;
}
```

A (5)处是重写覆盖

B (4)处是重写覆盖

C (1)是对(2)和(3)的重载

D (6)无编译错误并调用(5)

对于 A，注意重写覆盖的条件：在派生类中定义基类虚函数的同名同参函数。此处(1)不是虚函数，所以不是重写覆盖，是重写隐藏。

对于 D，首先，(5)不是重写覆盖，是重写隐藏，因而直接调用(5)。第二，基类函数对于派生类对象不可见，派生类对象或派生类指针引用哪怕 warning 也会先调用自身的函数。如果自身的函数没有一个可以调用的，哪怕基类函数有可以调用的，派生类对象宁愿 error 也不会调用基类函数。

3.6 override 关键字

重写覆盖要满足的条件很多，很容易写错，可以使用 override 关键字辅助检查。override 关键字明确地告诉编译器一个函数是对基类中一个虚函数的重写覆盖，编译器将对重写覆盖要满足的条件进行检查，正确的重写覆盖才能通过编译。如果没有 override 关键字，但是满足了重写覆盖的各项条件，也能实现重写覆盖。它只是编译器的一个检查，正确实现 override 时，对编译结果没有影响。

例子

```
#include <iostream>
using namespace std;
class Base{
public:
   virtual void foo(){cout<<"Base::foo()"<<endl;}
   virtual void foo(int ){cout<<"Base::foo(int )"<<endl;} ///重载
   void bar(){};
};
class Derived1 : public Base {
public:
   void foo(int ) {cout<<"Derived1::foo(int )"<<endl;} /// 是重写覆盖
};
```

```
class Derived2 : public Base {
public:
   void foo(float ) {cout<<"Derived2::foo(float )"<<endl;} /// 误把参数写错了，不是
重写覆盖，是重写隐藏,但是仍然通过了编译
};
class Derived3 : public Base {
public:
   void foo(int ) override {cout<<"Derived3::foo(int )"<<endl;/// 重写覆盖正确，与
Derived1 等价
   void foo(float ) override {}; /// 参数不同，不是重写覆盖，编译错误
   void bar() override {}; /// bar 非虚函数，编译错误
};
```

3.7 final 关键字

在虚函数声明或定义中使用 final 关键字后，确保函数为虚且不可被派生类重写。可在继承关系链的“中途“”进行设定，禁止后续派生类对指定虚函数重写，但是派生类仍可调用该虚函数。

在类定义中使用时，final 指定此类不可被继承。

Final 和 overide 在修饰函数时，都只能修饰虚函数。

例子

```
class Base{
  virtual void foo(){};
};
class A: public Base {
  void foo() final {}; /// 重写覆盖，且是最终覆盖
  void bar() final {}; /// bar 非虚函数，编译错误
};
class B final : public A{
  void foo() override {}; /// A::foo 已是最终覆盖，编译错误
};
class C : public B{ /// B 不能被继承，编译错误
};
```