多态

来看多态产生的一个场景：

```
class Animal {
public:
    Animal(const string &name) : __name(name) {}
    void run() {
        cout << "I don't know how to run" << endl;
    }
protected:
    string __name;
};
class Cat : public Animal {
public:
    Cat() : Animal("Cat") {}
    void run() {
        cout << "I can run with four legs" << endl;
    }
};
```

```
Cat a;
Animal &b = a;
Animal *c = &a;
a.run();
b.run();
c->run();
return 0;
```

可以看到无论是a、b、c，他们指向的是同一个猫类的对象，只不过被不同的标识符调用了。

- 当我们产生“让这只猫跑”的语义时：a.run()
- 当我们说的是“让这只动物跑”：b.run()，这时候会发现程序并不知道怎么做。（调用Animal类中的run方法，但是还未被定义）

可以看到这种实现方式的可笑之处/不合理之处：仅仅是换了种说法，结果就报错，（即使这仍然是一只猫对象）。

产生这种现象的原因：**普通的成员方法是跟着类走的。**当我们在调用普通的成员方法的时候，具体调用的是哪一个类中的方法看的是**前面标识符的类型。**比如a是Cat类型，b和c都是Animal类型。

虚函数

那么为了解决这种逻辑不合理的现象，C++提出了**虚函数**的概念。

- **普通成员函数：**跟着类走。
- **虚函数：**跟着对象走。

```
a.run();
b.run();
c.run();
// 甭管abc三个是什么类，我只知道他们都指向一个Cat猫对象，他们调用的都是我猫对象的run方法。
```

往深了讲：

- 普通的成员方法：**编译期的状态。（代码没有运行起来你也知道它是什么）**
- 虚函数：**运行时期的状态。（只有当代码运行起来了，你才能知道他是什么）**见下面这段程序。

```cpp
class Animal {
public:
    virtual void run() {
        cout << "I don't know how to run" << endl;
    }
};
//定义了一个猫类
class Cat : public Animal {
public:
    void run() override {
        cout << "I can run with four legs" << endl;
    }
};
//定义了一个人类
class Human : public Animal {
public:
    void run() override {
        cout << "I can run with two legs" << endl;
    }
};
//定义了一个鸟类
class Bird : public Animal {
public:
    void run() override {
        cout << "I can fly" << endl;
    }
};

int main() {
    srand(time(0));
    Animal *arr[10];
    for (int i = 0; i < 10; i++) {
        switch (rand() % 3) {
            case 0: arr[i] = new Cat(); break;
            case 1: arr[i] = new Human(); break;
            case 2: arr[i] = new Bird(); break;
        }
    }
    for (int i = 0; i < 10; i++) {
        arr[i]->run();
    }
    return 0;
}
// 当定义了虚函数地时候，你就无法确切地知道每次输出地结果到底是猫、人、鸟怎么样一个顺序。

// 反之如果没有定义虚函数地话，每次结果很确切地知道就是Animal地run()方法。
```

虚函数的定义：**在成员方法的前面加上virtual关键字**，当前的成员方法就变成了虚函数。

==**一个重要性质**：==如果在父类中，某个成员方法（包括析构函数）是虚函数，那么在**子类中该方法自动变为虚函数。**

virtual关键字

- **语义信息：**子类的这个方法可能会跟父类的有所不同。
- **成员方法调用时**：
  virtual关键字的方法跟着【对象】
  非virtual关键字的方法跟着【类】
- **限制：**不能用来修饰【类方法-static】
  > 因为本身逻辑上就不跟着对象走。

```
//父类:
virtual void run() {}

//按照编码规范，子类中重写run()的方法得要加上override关键字，但是该关键字不做任何功能性的保证（说白了去掉没

//子类:
void run() override {}
```

```
//override关键字的作用:
1. 防止写错函数名
2. 判断父类中的方法是不是虚函数
```

> **override关键字所体现的C++的设计哲学：**让我们大多数时候的运行时错误变成编译时错误；让更加严格的编码规范，让错误暴露在编译阶段。C++入门的特征：你能不能理解C++的设计哲学，**举个最简单的例子**：
>
> **public、private、protected三个关键字是作用在编译阶段**。那么当我们在编译代码的时候我们错误地访问了某个成员属性地时候，编译器就会给我们报错。所谓的访问控制权限，控制的是代码的准确性。

我们既然说override关键字是用来保证代码的准确性的。这又是为什么呢？

```
//假设在父类中由这样一个成员方法foo():
virtual void foo() {
    cout << "foo function" << endl;
}

//但是在子类中重写该方法的时候，由于程序员的粗心，把foo写成了f00
void f00() {
    cout << "f00 function" << endl;
}

//可见他已经犯了一个低级错误，但还没意识到，这个时候他要是写上了override关键字:
void f00() override {
    cout << "f00 function" << endl;
}
//编译器就会爆出没有f00这个函数的错误，即使有这玩意儿也不是虚函数啊。因此就能避免这种低级错误。
```

在继承的情况下，动态地申请和释放对象的时候究竟会出现什么问题？查看下面这段代码：

```
class Animal {
public:
    virtual void run() {
        cout << "I don't know how to run" << endl;
    }
    ~Animal() {
        cout << "Animal destructor" << endl;
    }
};
class Cat : public Animal {
public:
    void run() override {
        cout << "I can run with four legs" << endl;
    }
    ~Cat() {
        cout << "Cat destructor" << endl;
    }
};
class Human : public Animal {
public:
    void run() override {
        cout << "I can run with two legs" << endl;
    }
    ~Human() {
        cout << "Human destructor" << endl;
    }
};
class Bird : public Animal {
public:
    void run() override {
        cout << "I can fly" << endl;
    }
    ~Bird() {
        cout << "Bird destructor" << endl;
    }
};

int main() {
    srand(time(0));
    Animal *p;
    switch (rand() % 3) {
        case 0: p = new Cat();   break;
        case 1: p = new Human(); break;
        case 2: p = new Bird();  break;
    }
    p->run();
```

```
    delete p;
    return 0;
}
```

```
I can run with four legs
Animal destructor
// 我们知道这次的运行结果构造的顺序是先构造父类Animal，再构造子类Cat。
// 那么析构的时候原本应该先析构Cat类，再析构Animal类。
// 可是此时猫类的析构函数却并没有调用。
// 错误描述：没有办法正确地析构子类的相关空间。
```

> **无法正确析构子类相关空间的原因：**
>
> 再delete p的时候，实际上实在调用p的析构函数。p是Animal类的，Animal类中此时的析构函数是**普通的成员方法**，所以它就只会根据p的类型去判断调用哪一个析构函数。而一旦**在基类Animal前面加上virtual**以后就会变成虚函数，**所有派生类的析构函数也都会自动地变成虚函数。**这样就可以正确地析构子类对象了。
>
> 所以：所有基类的虚函数都得是虚函数。

纯虚函数

- **语义：**子类肯定会有这个方法，而父类只能说【抱歉】
- **应用场景：**定义接口

> 如果说虚函数还是有定义的，而**纯虚函数那就是任何定义都没有的。**声明形式如下所示：

```
class Animal {
public:
    virtual void run() = 0;
    //代表我并不需要知道Animal是怎么样跑的。
}
```

纯虚函数在逻辑上代表：在父类中，我们不知道这个方法是怎么实现的，或者说在父类中实现该方法是没有任何意义的。

一旦在父类中定义了纯虚函数，在**子类中是一定要去实现的。否则相应的子类是无法产生对象**的。比如如下的代码中Cat类没有去实现`run()`法：

```
class Cat : public Animal {
public:
    /*
    void run() override {
        cout << "I can run with four legs" << endl;
    }
    */
};
// 就会产生抽象类是没有办法产生对象的错误，
```

- 所谓**抽象类**，就是拥有**纯虚函数**的类：

> **抽象类无法产生对象**，一般作用就是用作接口定义的。**接口**是为了统一某类数据对外的调用形式的。
>
> 对于接口的理解：就好比充电线，我插在键盘上实现的是键盘的功能，插在耳机上实现的就是耳机的功能。具体实现什么功能不是接口定义的。
>
> 在Animal中我们定义了个虚函数，而我们说虚函数是跟着对象走的。又因为Animal类中包含纯虚函数，无法产生对象。所以如果Animal的指针要是指向了一个对象，这个对象一定是Animal类的派生类所产生的对象，也就是说，Animal类的派生类可以产生对象，即这个派生类必定是实现了所有的纯虚函数。至于怎么实现的可以是任意的。 我们唯一可以直接确定的是：这个函数接口形式是长这个样子。所以我们才管这种类叫”接口类“。

再往深了讲，为什么虚函数是一个运行时的状态？

虚函数表

> 运行时的状态统一都有一个特征：肯定需要占用运行时的一片存储空间，而虚函数占用的存储空间是一类叫做虚函数表的地方。

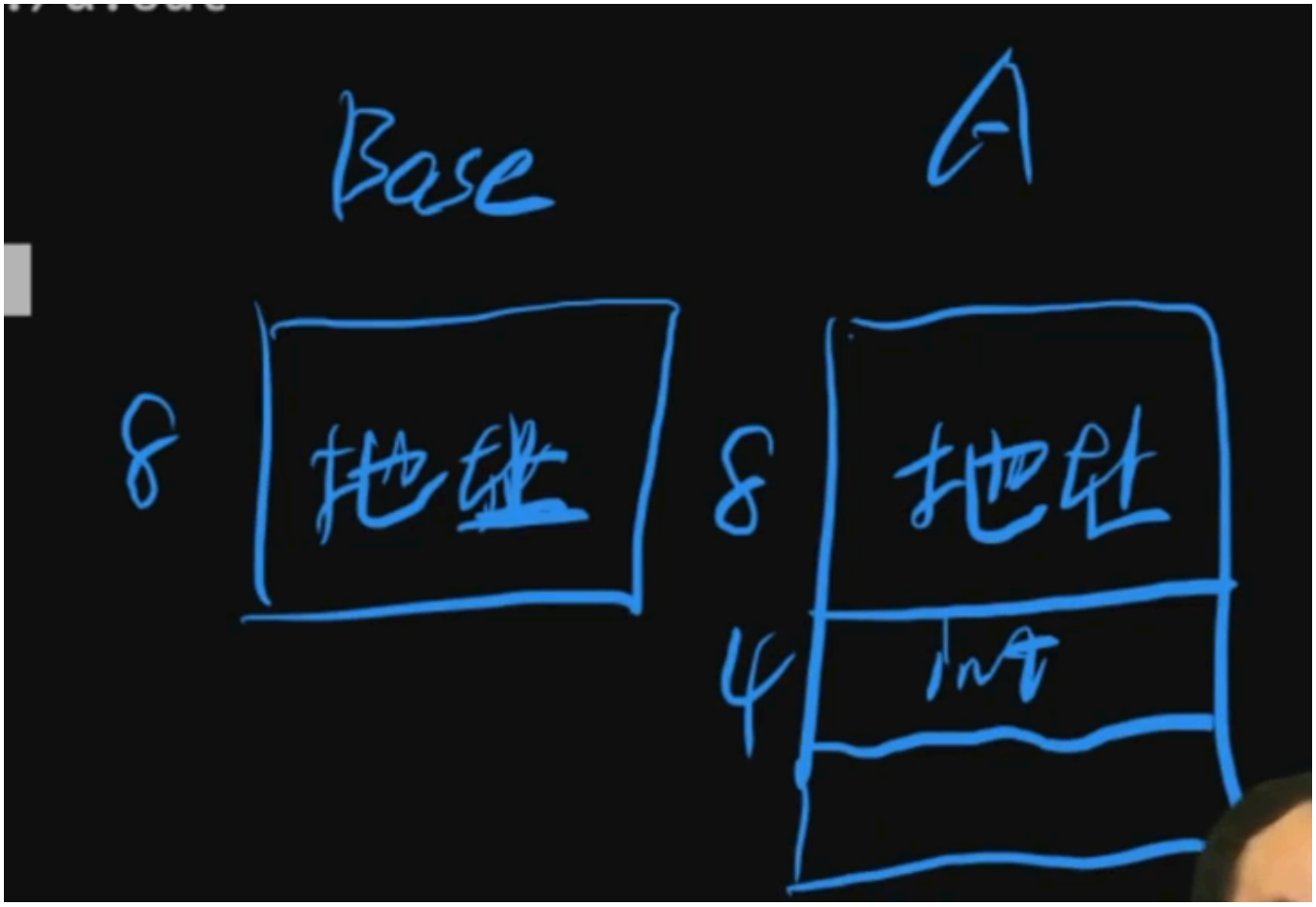
我们知道内存中**任何对象**肯定占用了**一片存储区**，有了虚函数以后的存储对象存储区的**头8个字节**会存储一个**地址**，这个地址指向**虚函数表（vtable）**，vtable里头装着每一个**虚函数**。

> 为什么虚函数可以跟着对象走？因为对象的头8个字节就记录着虚函数表的地址，一旦当我们调用的是某个虚函数的话，他会准确地找到虚函数在虚函数表中的位置，然后去准确地执行。

1. 虚函数表给类的大小带来的变化

```
#include<iostream>
using namespace std;
#define BEGINS(x) namespace x {
#define ENDS(x) }
BEGINS(test1)

class Base {
public:
    virtual ~Base() {}
};
class A : public Base {
public:
    int x;
};
class B {
public:
    int x;
};

int main() {
    cout << "sizeof(Base) = " << sizeof(Base) << endl;
    cout << "sizeof(A)    = " << sizeof(A) << endl;
    cout << "sizeof(B)    = " << sizeof(B) << endl;
    return 0;
}
ENDS(test1)


int main() {
    test1::main();
    return 0;
}
```

```
sizeof(Base) = 8
sizeof(A)    = 16
sizeof(B)    = 4
```



A类中最大的数据是8个字节，所以A类中就会按照**8个字节来默认对齐**

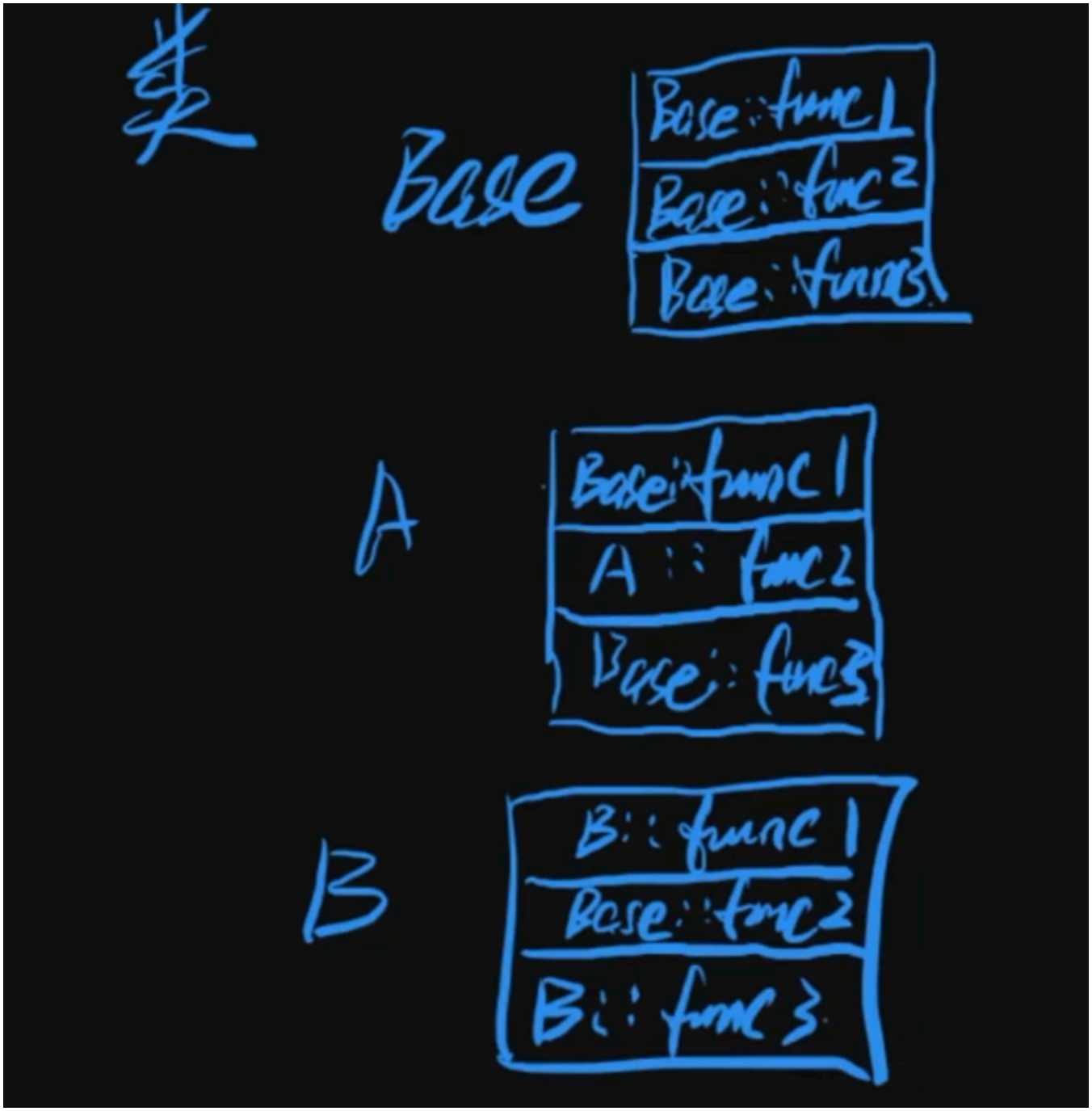
2. 画出一下代码中三个类的虚函数表：

```cpp
class Base {
public:
    virtual void func1() {
        cout << "Base func1" << endl;
    }
    virtual void func2() {
        cout << "Base func2" << endl;
    }
    virtual void func3() {
        cout << "Base func3" << endl;
    }
    virtual ~Base() {}
};
class A : public Base {
public:
    void func2() override {
        cout << "A func2" << endl;
    }
    int x;
};
class B : public Base {
public:
    void func1() override {
        cout << "B func1" << endl;
    }
    void func3() override {
        cout << "B func3" << endl;
    }
    int x;
};
int main() {
    A a;
    B b;
    a.func1();
    a.func2();
    a.func3();
    b.func1();
    b.func2();
    b.func3();
    return 0;
}
/*
    Base func1
    A func2
    Base func3
    B func1
```

```
    Base func2
    B func3
*/
```

三个类的虚函数表的画法：

> 注意是三个类而不是对象，相同类的对象指向的虚函数表是**同一张**。



3. 如何用特殊的方式调用到虚函数表的第一个虚函数？

首先理解虚函数表是一个存储着函数地址的数组。假设函数类型是`func`类型。那么数组每一位存储的类型就是`func *`类型，数组的首地址就是`func **`，所以可以用如下的C语言原生指针的形式调用：

```
typedef void (*func)();
int main() {
    A a;
    a.func(123);
    ((func **)(&a))[0][1]();
    return 0;
}
```

但会出现一个问题：C语言原生指针的形式没法正确传参：

```
//上面的输出结果为:
0x7ffcdace1960 : class A : 123
0x7b : class A : 0
// 0x7b 换算成10进制就是123，也就是说123没能传给x，而是传给了this指针
```

这就涉及到另外一个知识点：**this指针**

C++的成员方法，看上去只有一个参数，实际上是有两个参数的`func(this, x)`，第一个参数是一个隐藏参数，this指针。

`a.func(123);`其实可以被翻译成(在底层的调用形式)`func(&a, 123);`

意识到这一点之后，原生指针在设计的时候应该像这样：

```
typedef void (*func)(void *, int );
int main() {
        ((func **)(&a))[0][0](&a, 123);
}
// 0x7ffe8ae1ce60 : class A : 123
// 0x7ffe8ae1ce60 : class A : 123
```

我们甚至可以通过获取a对象的虚函数表的虚函数方法，传入b对象的地址给this指针，于是调用b对象的方法：

```
A a, b;
a.x = 1000;
b.x = 10000;
a.func(123);
((func **)(&a))[0][0](&a, 123);
((func **)(&a))[0][0](&b, 123);
/*
    0x7ffe961c2f60 : class A : 123
    this->x : class A : 1000
    0x7ffe961c2f70 : class A : 123
    this->x : class A : 10000
*/
```

4. 返回值优化

再来回顾一个知识点：当初学习返回值优化的时候，是不是很惊讶**编译器为啥能把a对象和temp对象当成一个对象？**

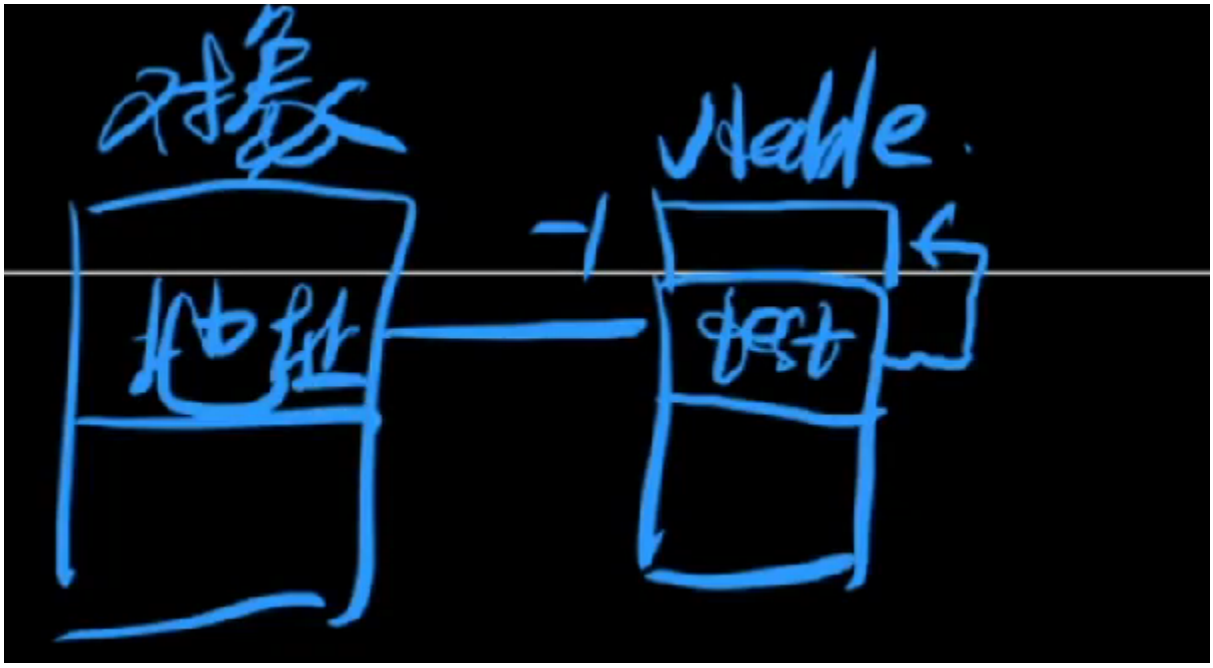
```
A func() {
    A temp(3);
    cout << "object temp: " << &temp << endl;
    return temp;
}

int main() {
    A a = func();
    cout << "object a: " << &a << endl;
    return 0;
}
```

在调用构造函数的时候，构造函数本质上也有个隐藏的this指针，所以编译器只需要把a对象的地址传给`A temp(3)`的this指针就行了。保证了我们对temp对象的所有操作最终都会同步到a对象上。同时在func函数里面取temp对象的地址也会发现是a对象的地址。

动态类型转换

*dynamic_cast<A >(p) 关键字：将父类指针转换为子类指针地这样一种类型转换方法：如果转化之后为空指针nullptr，则p指针不是A *类型，反之如果不为空，则为A *类型。*
*但是dynamic_cast只有在*多态（polymorphic）**情况下才能用，为什么？*

在多态的情况下，每一个对象都额外地记录着一个虚函数表，所以dynamic_cast是通过虚函数表来判断当前对象到底属于什么类型的。之所以能作出这样的判断是基于每个类的虚函数表是唯一的。其实，在虚函数表的**-1位存着类型信息**：



那么怎么让一个Base类变成多态呢？加一个虚函数，怎么加呢？总不可能加一个无用的虚函数吧，那样的话可太丑了。其实按照编码规范，基类天生就有一个虚函数——**析构函数。**，这样一来基类就有一张虚函数表了。就可以用dynamic_cast了。

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

#define BEGINS(x) namespace x {
#define ENDS(x) }
BEGINS(test1)
class Base {
public:
    virtual ~Base() {}
};

class A : public Base {};
class B : public Base {};
class C : public Base {};

int main() {
    srand(time(0));
    Base *p;
    switch (rand() % 4) {
        case 0: p = new A(); break;
        case 1: p = new B(); break;
        case 2: p = new C(); break;
        case 3: p = nullptr; break;
    }
    A a;
    if (p) ((void **)(p))[0] = ((void **)(&a))[0];
    // 把p绑定为类型A.
    if (dynamic_cast<A *>(p) != nullptr) {
        cout << "p pointer A class Object" << endl;
    } else if (dynamic_cast<B *>(p) != nullptr) {
        cout << "p pointer B class Object" << endl;
    } else if (dynamic_cast<C *>(p) != nullptr) {
        cout << "p pointer C class Object" << endl;
    } else {
        cout << "p is nullptr" << endl;
    }
    return 0;
}
ENDS(test1)
int main() {
    test1::main();
    return 0;
}
```

按理来说，不同类型的对象，虚函数表的首地址是不一样的。那么按理说我就可以通过虚函数表的首地址判读类型信息了。但是为什么不能这么做？为什么还要提出一个-1位置存储类型信息？

本质原因：-1位置和虚函数表首地址所能提供信息的能力不同。

比如说C类型继承自A类型，一个地址是没有办法包含这种继承关系的。而-1位置的类型信息，在它的存储结构中可以足够复杂到维护这种**继承关系**。

那么类型信息为什么要放上面-1位置呢？

其实就是说，取数据的时候比较方便，虚函数表是用来选择到相关的虚函数的，每个虚函数地址都是固定的8个字节，而类型信息没有固定的大小，况且本质上也是两类信息。