可以不可以

静态函数可以声明为虚函数吗?

> **静态函数不可以声明为虚函数，同时也不能被const 和 volatile关键字修饰**,
>
> static成员函数不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义
>
> 虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，静态成员函数没有this指针，所以无法访问vptr。

构造函数可以为虚函数吗?

> **构造函数不可以声明为虚函数**。同时除了inline|explicit之外，构造函数不允许使用其它任何关键字。
>
> 为什么构造函数不可以为虚函数?
>
> 尽管虚函数表vtable是在编译阶段就已经建立的，但指向虚函数表的指针vptr是在运行阶段实例化对象时才产生的。 如果类含有虚函数，编译器会在构造函数中添加代码来创建vptr。问题来了，如果构造函数是虚的，那么它需要vptr来访问vtable，可这个时候vptr还没产生。 因此，构造函数不可以为虚函数。
>
> 我们之所以使用虚函数，是因为需要在信息不全的情况下进行多态运行。而构造函数是用来初始化实例的，实例的类型必须是明确的。 因此，构造函数没有必要被声明为虚函数。

析构函数可以为虚函数吗?

> **析构函数可以声明为虚函数**。如果我们需要删除一个指向派生类的基类指针时，应该把析构函数声明为虚函数。 事实上，只要一个类有可能会被其它类所继承， 就应该声明虚析构函数(哪怕该析构函数不执行任何操作)。

虚函数可以为私有函数吗?

> **基类指针指向继承类对象，则调用继承类对象的函数**； int main()必须声明为Base类的友元，否则编译失败。 编译器报错： ptr无法访问私有函数。 当然，把基类声明为public， 继承类为private，该问题就不存在了。

虚函数可以被内联吗?

> **通常类成员函数都会被编译器考虑是否进行内联。 但通过基类指针或者引用调用的虚函数必定不能被内联。 当然，实体对象调用虚函数或者静态调用时可以被内联，虚析构函数的静态调用也一定会被内联展开。**

> 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。 inline virtual 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 Base::who()），这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

*关于dynamic_cast

【格式】：dynamic_cast < type-id > (expression)

> 该运算符把expression转换成type-id类型的对象。Type-id可以是类的指针、类的引用或者void*。如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

【作用】：将一个基类对象指针（或引用）cast到继承类指针，dynamic_cast会根据基类指针是否真正指向继承类指针来做相应处理， 即会作出一定的判断。

> 若对指针进行dynamic_cast，失败返回null，成功返回正常cast后的对象指针； 若对引用进行dynamic_cast，失败抛出一个异常，成功返回正常cast后的对象引用。

【注意】：

1、dynamic_cast在将父类cast到子类时，父类必须要有虚函数，否则编译器会报错。

2、 dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

> 在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的； 在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全。

下面通过一个简单的例子来说明dynamic_cast的作用：

```
#include <iostream>
#include <assert.h>
using namespace std;
// 我是父类
class Tfather
{
public:
    virtual void f() { cout << "father's f()" << endl; }
};

// 我是子类
class Tson : public Tfather
{
public:
    void f() { cout << "son's f()" << endl; }
    int data; // 我是子类独有成员
};

int main()
```

```
{
    Tfather father;
    Tson son;
    son.data = 123;

    Tfather *pf;
    Tson *ps;

    /* 上行转换：没有问题，多态有效 */
    ps = &son;
    pf = dynamic_cast<Tfather *>(ps);
    pf->f();

    /* 下行转换（pf实际指向子类对象）：没有问题 */
    pf = &son;
    ps = dynamic_cast<Tson *>(pf);
    ps->f();
    cout << ps->data << endl;       // 访问子类独有成员有效

    /* 下行转换（pf实际指向父类对象）：含有不安全操作，dynamic_cast发挥作用返回NULL */
    pf = &father;
    ps = dynamic_cast<Tson *>(pf);
    assert(ps != NULL);             // 违背断言，阻止以下不安全操作
    ps->f();
    cout << ps->data << endl;       // 不安全操作，对象实例根本没有data成员

    /* 下行转换（pf实际指向父类对象）：含有不安全操作，static_cast无视 */
    pf = &father;
    ps = static_cast<Tson *>(pf);
    assert(ps != NULL);
    ps->f();
    cout << ps->data << endl;       // 不安全操作，对象实例根本没有data成员

    system("pause");
}
```

> user: 我没看懂,,,,被自己菜哭了

//摘自C++那些事。