

## 2020/05/19\_C++\_第10课\_虚函数、多态

笔记本: C++  
创建时间: 2020/5/19 星期二 15:03  
作者: ileemi  
标签: 多态, 虚表、虚表指针

---

- [虚函数](#)
- [成员函数指针](#)
  - [为上面的程序继续添加新的功能（为英雄增加出厂台词）](#)
- [虚函数内部实现、虚表、虚表指针](#)
- [多态](#)

# 虚函数

解决代码重用问题，使代码更加通用。当基类派生类有同名成员函数时，在想使用派生类的时候，程序可能存在问题，不能够精确的使用派生类的成员函数，代码所示如下：

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

// 基类
class CHero
{
public:
    // 英雄技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
            case 0:
                cout << " 平A " << endl;
                break;
            case 1:
                cout << " 回城 " << endl;
                break;
            case 2:
                cout << " 恢复 " << endl;
                break;
            default:
                break;
        }
    }
};
```

```

    }
}
};

// 派生类: 战士
class CWarrior : public CHero
{
public:
    // 战士技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:
            cout << "回城" << endl;
            break;
        case 2:
            cout << "回旋之刃" << endl;
            break;
        case 3:
            cout << "极刃风暴" << endl;
            break;
        case 4:
            cout << "不灭魔躯" << endl;
            break;
        default:
            break;
        }
    }
};

```

```

// 派生类: 法师
class CMage : public CHero
{
public:
    // 法师技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:

```

```

        cout << "回城" << endl;
        break;
    case 2:
        cout << "泪如泉涌" << endl;
        break;
    case 3:
        cout << "叹息水流" << endl;
        break;
    case 4:
        cout << "洛神降临" << endl;
        break;
    default:
        break;
    }
}
};

int main()
{
    //生成英雄
    cout << "请选择要初始化的英雄：(1-战士、2-法师)" << endl;
    CHero* pAryHeros[2] = { nullptr };

    //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
    //for (auto& pHero : pAryHeros)
    for (CHero*& pHero : pAryHeros)
    {
        int nIdx = 0; //输入对应的数值，输出对应类对象数据
        cin >> nIdx;
        switch (nIdx)
        {
        case 1:
            pHero = new CWarrior;
            break;
        case 2:
            pHero = new CMage;
            break;
        default:
            break;
        }
    }

    // 释放技能（输入对应的数值，输出对应的技能）
    while (true)
    {
        int nSkill = 0;
        cout << "输入0~4即可释放对应的技能：" << endl;

```

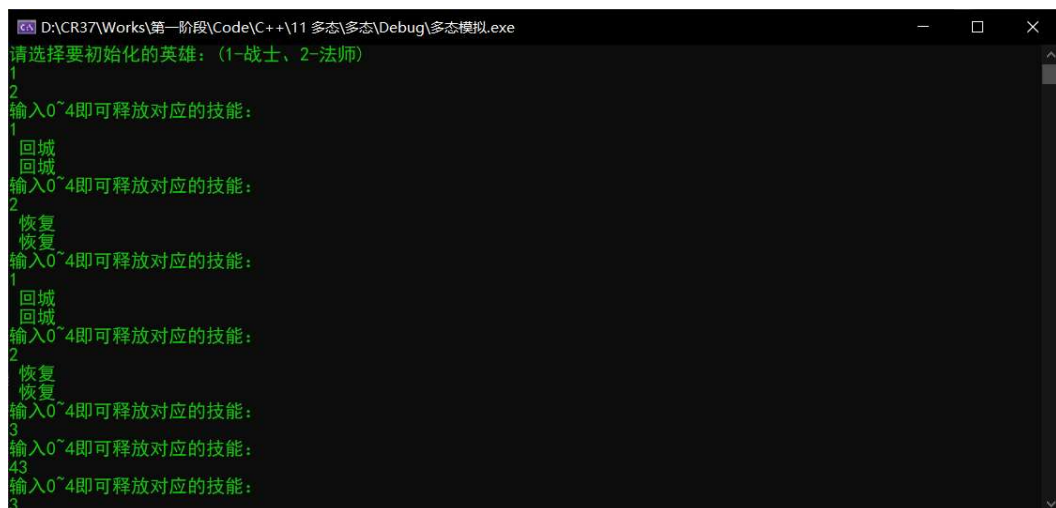
```

    cin >> nSkill;
    //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
    //for (auto& pHero : pAryHeros)
    for (CHero*& pHero: pAryHeros)
    {
        // 此时程序有Bug，没有按照预期的效果输出结果，没有进入子列，进入
        父类(CHero*)
        pHero->UseSkills(nSkill);
    }

    // 解决方法1:
    /*
    // 存在问题，代码通用性不高
    int nType = pHero->GetType();
    switch(nType)
    {
    case TYPE_WARRIOR:
        ((CWarrior*)pHero)->UseSkills(nSkill);
        break;
    case TYPE_MAGE:
        ((CMage*)pHero)->UseSkills(nSkill);
        break;
    }
    */
}

return 0;
}

```



```

D:\CR37\Works\第一阶段\Code\C++\11 多态\多态\Debug\多态模拟.exe
请选择要初始化的英雄：(1-战士、2-法师)
1
2
输入0~4即可释放对应的技能：
1
回城
回城
输入0~4即可释放对应的技能：
2
恢复
恢复
输入0~4即可释放对应的技能：
1
回城
回城
输入0~4即可释放对应的技能：
2
恢复
恢复
输入0~4即可释放对应的技能：
3
回城
回城
输入0~4即可释放对应的技能：
43
回城
回城
输入0~4即可释放对应的技能：
3
回城
回城

```

## 成员函数指针

成员函数指针(默认传递\_\_thiscall)不可以赋值(转换)非成员函数指针

子类对象重写了父类的成员函数

允许将子类对象的指针赋值给父类类型的对象

**成员函数的调用方式（使用对象、使用指针）：**

代码示例：

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

class A
{
public:
    //成员函数的调用约定默认为__thiscall
    void Test()
    {
        cout << "A::Test()" << endl;
    }
};

//1、2
//函数指针的定义方式
//typedef void(*PFN_TEST)(); //一般的函数指针

//3
//typedef void(__thiscall *PFN_TEST)();

//4
//typedef void(__thiscall A:: *PFN_TEST)();

//5 __thiscall可以去掉，编译器在编译时，会自动判断其调用约定
typedef void(A :: *PFN_TEST)();

int main()
{
    //1
    //PFN_TEST pfn = A::Test;
    //error C3867: "A::Test": 非标准语法; 请使用 "&" 来创建指向成员的指针

    //2
    //PFN_TEST pfn = &A::Test;
    //error C2440: "初始化": 无法从 "void (__thiscall A::*)(void)" 转换为 "PFN_TEST"

    //3
    //main函数外: typedef void(__thiscall * PFN_TEST)();
    //error C2440: "初始化": 无法从 "void(__thiscall A::*)(void)" 转换为
```

为 “PFN\_TEST”

```
//4 成员函数定义处应添加类名表明其作用域
//typedef void(__thiscall A::* PFN_TEST) ();

PFN_TEST pfn = &A::Test;

//成员函数的调用方式（使用对象、使用指针）
//使用对象怎样调用 pfn?
A a;
//a.pfn();
//a.*pfn(); //error C2064: 项不会计算为接受 0 个参数的函数

//固定写法（标准写法）：
(a.*pfn)(); //输出：A::Test()

//使用指针，pA怎样调 pfn?
A* pA = &a;
(pA->*pfn)();
return 0;
}
```

成员函数存在的么蛾子情况，代码示例：

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

class A
{
public:
    //成员函数的调用约定默认为__thiscall
    void Test()
    {
        cout << "A::Test()" << endl;
    }
};

//可否将一个基类的成员函数赋值给一个派生类的成员函数指针
class B:public A
{
public:
    //成员函数的调用约定默认为__thiscall
    void Test1()
    {
        cout << "A::Test()" << endl;
    }
};
```

```

//1、2
//函数指针的定义方式
//typedef void(*PFN_TEST)(); //一般的函数指针

//3
//typedef void(__thiscall *PFN_TEST)();

//4
//typedef void(__thiscall A:: *PFN_TEST)();

//5 __thiscall可以去掉，编译器在编译时，会自动判断其调用约定
typedef void(A :: *PFN_TEST)();
typedef void(B :: *PFN_TEST1)();

//一般的函数指针
typedef void(*PFN)();

int main()
{

    // 问题1
    //可否将一个基类的成员函数赋值给一个派生类的成员函数指针？
    //安全的，派生类指针pfnBTest由派生类来调用，派生类对象可以调用基类的
    成员函数，没有问题
    PFN_TEST1 pfnBTest = &A::Test; //可以

    // 问题2
    //可否将一个派生类的成员函数赋值给一个基类的成员函数指针？
    //不安全的，pfnATest为基类，基类的函数指针由基类对象来调用，Test1为派
    生类的成员函数

    //PFN_TEST pfnATest = &B::Test1; //不可以

    // error C2440: “初始化”：无法从“void (__thiscall B::*)(void)”转
    换为“PFN_TEST”
    //message : 从基类型到派生类型的强制转换需要 dynamic_cast 或
    static_cast

    /*
    |-----|
    |   基类   |
    |-----|----> 派生类成员函数可以访问的内存，
    |   派生类 | 这个时候，如果基类的对象调用派生类的成员函数存在访问越
    界
    |-----|
    */
}

```

```

//派生类的成员函数赋值给一个基类的成员函数指针可以强转
//PFN_TEST pfnATest = &B::Test1; //不可以
PFN_TEST pfnATest = (PFN_TEST)&B::Test1; //可以强转

// 问题3 可否将成员函数指针赋值给一般的函数指针?

//typedef void(*PFN)(); //一般的函数指针
//PFN pfn = pfnATest;
// error C2440: “初始化”: 无法从“PFN_TEST”转换为“PFN”

//尝试将其强转
//PFN pfn = (PFN_TEST)pfnATest;
// error C2440: “初始化”: 无法从“PFN_TEST”转换为“PFN”

//尝试转void * -->不可行
//PFN pfn = (void *)pfnATest;
//error C2440: “类型强制转换”: 无法从“PFN_TEST”转换为“void *”

//可使用下面的方式, 将pfnATest在内存中的地址进行强转取内容, 在强转
PFN pfn = (PFN)(int *)&pfnATest; //了解即可, 不推荐使用

/*
存在危害:
成员函数指针pfnATest默认参为 __thiscall 默认传递this 指针,
一般的函数指针不会传递this指针
*/
return 0;
}

```

**使用成员函数指针修改上面程序存在的问题:**

代码示例如下:

```

#include <iostream>
using namespace std;

class CHero;
//定义成员函数指针
typedef void(CHero :: *PFN_HERO_USESKILLS)(int);

class CHero
{
public:
    CHero()

```



```

{
    m_pfnHeroUseSkills = &CHero::UseSkills;
    /*
        error C2440: “=” : 无法从 “void (__thiscall CHero::* )(int)” 转换
        为 “PFN_HERO_USESKILLS”
        声明不匹配，在定义成员函数指针的地方增加对应的参数
    */
}

//英雄技能
void UseSkills(int nSkill)
{
    switch (nSkill)
    {
    case 0:
        cout << "平A" << endl;
        break;
    case 1:
        cout << "回城" << endl;
        break;
    case 2:
        cout << "恢复" << endl;
        break;
    default:
        break;
    }
}

//在类内定义一个成员函数指针
public:
    PFN_HERO_USESKILLS m_pfnHeroUseSkills;
    //什么时候对成员函数指针进行赋值？赋值操作可以在构造时进行
};

//派生类：战士
class CWarrior : public CHero
{
public:
    //战士技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:
            cout << "回城" << endl;
            break;
        }
    }
}

```

```

        case 2:
            cout << "回旋之刃" << endl;
            break;
        case 3:
            cout << "极刃风暴" << endl;
            break;
        case 4:
            cout << "不灭魔躯" << endl;
            break;
        default:
            break;
    }
}

public:
    CWarrior()
    {
        m_pfnHeroUseSkills = (PFN_HERO_USESKILLS)&CWarrior::UseSkills;
        /*
            error C2440: "=" : 无法从 "void (__thiscall CWarrior::*)(int)" 转
            换为 "PFN_HERO_USESKILLS"
            需要强转
        */
    }
};

//派生类：法师
class CMage : public CHero
{
public:
    //法师技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:
            cout << "回城" << endl;
            break;
        case 2:
            cout << "泪如泉涌" << endl;
            break;
        case 3:
            cout << "叹息水流" << endl;
            break;
        case 4:

```

```

        cout << "洛神降临" << endl;
        break;
    default:
        break;
    }
}

public:
    CMage()
    {
        m_pfnHeroUseSkills = (PFN_HERO_USESKILLS)&CMage::UseSkills;
        /*
        error C2440: "=" : 无法从 "void (__thiscall CMage::*)(int)" 转换
        为 "PFN_HERO_USESKILLS"
        需要强转
        */
    }
};

int main()
{
    //生成英雄
    cout << "请选择要初始化的英雄：(1-战士、2-法师)" << endl;
    CHero* pAryHeros[2] = { nullptr };

    //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
    //for (auto& pHero : pAryHeros)
    for (CHero*& pHero : pAryHeros)
    {
        int nIdx = 0;          //输入对应的数值，输出对应类对象数据
        cin >> nIdx;
        switch (nIdx)
        {
        case 1:
            pHero = new CWarrior;
            break;
        case 2:
            pHero = new CMage;
            break;
        default:
            break;
        }
    }

    //释放技能（输入对应的数值，输出对应的技能）
    while (true)
    {

```

```

int nSkill = 0;
cout << "输入0~4即可释放对应的技能：" << endl;
cin >> nSkill;
//pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
//for (auto& pHero : pAryHeros)
for (CHero*& pHero: pAryHeros)
{
    //此时程序有Bug，没有按照预期的效果输出结果，没有进入子列，进入父
    类(CHero*)
    //pHero->UseSkills(nSkill);

    // 使用成员函数指针进行访问
    //(pHero->*m_pfnHeroUseSkills)(nSkill);
    /*
    编译不通过m_pfnHeroUseSkills不是成员函数指针，是对象的数据成员，
    应该将其取出来(pHero->m_pfnHeroUseSkills)
    */

    //(对象的指针->*成员函数指针)();
    //此时这段代码就通用了

    //符合多态
    //多态的定义：不关心子类对象的类型，调用属于其自己的成员函数
    (pHero->*(pHero->m_pfnHeroUseSkills))(nSkill);

#if 0
    class CFoo
    {
    public:
        PFN_HERO_USESKILLS m_pfn;
    };
    CFoo foo;
    (pHero->*(foo.m_pfn))(nSkill);
#endif // 0
}

//解决方法1：存在代码不通用
#if 0
//存在问题，代码通用性不高
int nType = pHero->GetType();
switch(nType)
{
case TYPE_WARRIOR:
    ((CWarrior*)pHero)->UseSkills(nSkill);
    break;
case TYPE_MAGE:
    ((CMage*)pHero)->UseSkills(nSkill);

```

```

        break;
    }
#endif //

}

return 0;
}

```

程序运行效果图：

```

D:\CR37\Works\第一阶段\Code\C++\11 多态\多态\Debug\多态模拟.exe
请选择要初始化的英雄：(1-战士、2-法师)
1
2
输入0~4即可释放对应的技能：
0
平A
平A
输入0~4即可释放对应的技能：
1
回城
回城
输入0~4即可释放对应的技能：
2
回旋之刃
泪如泉涌
输入0~4即可释放对应的技能：
3
极刃风暴
叹息水流
输入0~4即可释放对应的技能：
4
不灭魔躯
洛神降临
输入0~4即可释放对应的技能：
5

```

为上面的程序继续添加新的功能（为英雄增加出厂台词）

```

#include <iostream>
using namespace std;

class CHero;
//定义成员函数指针
typedef void(CHero :: *PFN_HERO_USESKILLS)(int);
typedef void(CHero :: *PFN_HERO_SPEAK)();
//当添加一个新功能，需要添加一个成员函数指针，操作不够灵活

union HERO_FUNCS
{
    PFN_HERO_SPEAK m_pfnHeroSpeak;
    PFN_HERO_USESKILLS m_pfnHeroUseSkills;
};

class CHero
{
public:
    CHero()
    {
        m_AryHeroFuncs[0].m_pfnHeroSpeak = &CHero::Speak;
    }

```

```

        m_AryHeroFuncs[1].m_pfnHeroUseSkills = &CHero::UseSkills;
    }
    //英雄技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
            case 0:
                cout << "平A" << endl;
                break;
            case 1:
                cout << "回城" << endl;
                break;
            case 2:
                cout << "恢复" << endl;
                break;
            default:
                break;
        }
    }
    //在类内定义一个成员函数指针

    //增加一个说台词的功能
    void Speak()
    {
        cout << "CHero" << endl;
    }
public:
    //属于类，不属于对象，这样写每个子类都会包含一份，可将其这是为静态
    HERO_FUNCS m_AryHeroFuncs[2];

};
//派生类：战士
class CWarrior : public CHero
{
public:
    //战士技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
            case 0:
                cout << "平A" << endl;
                break;
            case 1:
                cout << "回城" << endl;
                break;

```

```

        case 2:
            cout << "回旋之刃" << endl;
            break;
        case 3:
            cout << "极刃风暴" << endl;
            break;
        case 4:
            cout << "不灭魔躯" << endl;
            break;
        default:
            break;
    }
}

//增加一个说台词的功能
void Speak()
{
    cout << "一个人可以被毁灭，但绝不可以被打败" << endl;
}

public:
    CWarrior()
    {
        m_AryHeroFuncs[0].m_pfnHeroSpeak = (PFN_HERO_SPEAK)&CWarrior::Speak;
        m_AryHeroFuncs[1].m_pfnHeroUseSkills =
(PFN_HERO_USESKILLS)&CWarrior::UseSkills;
    }
};

//派生类：法师
class CMage : public CHero
{
public:
    //法师技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:
            cout << "回城" << endl;
            break;
        case 2:
            cout << "泪如泉涌" << endl;
            break;
        case 3:

```

```

        cout << "叹息水流" << endl;
        break;
    case 4:
        cout << "洛神降临" << endl;
        break;
    default:
        break;
    }
}

//增加一个说台词的功能
void Speak()
{
    cout << "春哪春，得和你两留连，春去如何遣？" << endl;
}

public:
    CMage()
    {
        m_AryHeroFuncs[0].m_pfnHeroSpeak = (PFN_HERO_SPEAK)&CMage::Speak;
        m_AryHeroFuncs[1].m_pfnHeroUseSkills =
(PFN_HERO_USESKILLS)&CMage::UseSkills;
    }
};

int main()
{
    //生成英雄
    cout << "请选择要初始化的英雄：(1-战士、2-法师)" << endl;
    CHero* pAryHeros[2] = { nullptr };

    //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
    //for (auto& pHero : pAryHeros)
    for (CHero*& pHero : pAryHeros)
    {
        int nIdx = 0;          //输入对应的数值，输出对应类对象数据
        cin >> nIdx;
        switch (nIdx)
        {
        case 1:
            pHero = new CWarrior;
            break;
        case 2:
            pHero = new CMage;
            break;
        default:
            break;
        }
    }
}

```



```

    }

    //释放技能（输入对应的数值，输出对应的技能）
    while (true)
    {
        int nSkill = 0;
        cout << "输入0~4即可释放对应的技能：" << endl;
        cin >> nSkill;
        //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
        //for (auto& pHero : pAryHeros)
        for (CHero*& pHero : pAryHeros)
        {
            //符合多态
            //多态的定义：不关心子类对象的类型，调用属于其自己的成员函数
            (pHero->*(pHero->m_AryHeroFuncs[0].m_pfnHeroSpeak)) ();
            (pHero->*(pHero->m_AryHeroFuncs[1].m_pfnHeroUseSkills)) (nSkill);
            cout << endl;
        }
    }

    return 0;
}

```

**进一步优化（每个类都有一个属于自己的静态成员数组，存储了自己的成员函数的地址）：**

```

#include <iostream>
using namespace std;

//定义成员函数指针
typedef void(CHero::* PFN_HERO_USESKILLS) (int);
typedef void(CHero::* PFN_HERO_SPEAK) ();
//当添加一个新功能，需要添加一个成员函数指针，操作不够灵活

union HERO_FUNCS
{
    PFN_HERO_SPEAK m_pfnHeroSpeak;
    PFN_HERO_USESKILLS m_pfnHeroUseSkills;
};

class CHero
{
public:
    CHero()
    {
        //m_AryHeroFuncs[0].m_pfnHeroSpeak = &CHero::Speak;
        //m_AryHeroFuncs[1].m_pfnHeroUseSkills = &CHero::UseSkills;
    }
}

```

```

        m_pfnHeroFuncs = m_AryHeroFuncs;
    }
    //英雄技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
            case 0:
                cout << "平A" << endl;
                break;
            case 1:
                cout << "回城" << endl;
                break;
            case 2:
                cout << "恢复" << endl;
                break;
            default:
                break;
        }
    }
    //在类内定义一个成员函数指针

    //增加一个说台词的功能
    void Speak()
    {
        cout << "CHero" << endl;
    }
public:
    //属于类，不属于对象，这样写每个子类都会包含一份，可将其这是为静态
    //HERO_FUNCS m_AryHeroFuncs[2];
    HERO_FUNCS* m_pfnHeroFuncs;
    static HERO_FUNCS m_AryHeroFuncs[2];
};

//每个类都有一个属于自己的静态成员数组，存储了自己的成员函数的地址
HERO_FUNCS CHero::m_AryHeroFuncs[2] =
{
    &CHero::Speak,
    (PFN_HERO_SPEAK)&CHero::UseSkills
};

//派生类：战士
class CWarrior : public CHero
{
public:
    CWarrior()
    {

```

```

        //m_AryHeroFuncs[0].m_pfnHeroSpeak =
        (PFN_HERO_SPEAK)&CWarrior::Speak;

        //m_AryHeroFuncs[1].m_pfnHeroUseSkills =
        (PFN_HERO_USESKILLS)&CWarrior::UseSkills;

        m_pfnHeroFuncs = m_AryWarriorFuncs;
    }

public:
    //战士技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
            case 0:
                cout << "平A" << endl;
                break;
            case 1:
                cout << "回城" << endl;
                break;
            case 2:
                cout << "回旋之刃" << endl;
                break;
            case 3:
                cout << "极刃风暴" << endl;
                break;
            case 4:
                cout << "不灭魔躯" << endl;
                break;
            default:
                break;
        }
    }

    //增加一个说台词的功能
    void Speak()
    {
        cout << "一个人可以被毁灭，但绝不可以被打败" << endl;
    }

    static HERO_FUNCS m_AryWarriorFuncs[2];
};

//每个类都有一个属于自己的静态成员数组，存储了自己的成员函数的地址
HERO_FUNCS CWarrior::m_AryWarriorFuncs[2] =
{
    (PFN_HERO_SPEAK)&CWarrior::Speak,

    (PFN_HERO_SPEAK) (PFN_HERO_USESKILLS)&CWarrior::UseSkills
};

```

```

//派生类：法师
class CMage : public CHero
{
public:
    CMage()
    {
        //m_AryHeroFuncs[0].m_pfnHeroSpeak = (PFN_HERO_SPEAK)&CMage::Speak;
        //m_AryHeroFuncs[1].m_pfnHeroUseSkills =
        (PFN_HERO_USESKILLS)&CMage::UseSkills;
        m_pfnHeroFuncs = m_AryMageFuncs;

    }
public:
    //法师技能
    void UseSkills(int nSkill)
    {
        switch (nSkill)
        {
        case 0:
            cout << "平A" << endl;
            break;
        case 1:
            cout << "回城" << endl;
            break;
        case 2:
            cout << "泪如泉涌" << endl;
            break;
        case 3:
            cout << "叹息水流" << endl;
            break;
        case 4:
            cout << "洛神降临" << endl;
            break;
        default:
            break;
        }
    }

    //增加一个说台词的功能
    void Speak()
    {
        cout << "春哪春，得和你两留连，春去如何遣？" << endl;
    }

    static HERO_FUNCS m_AryMageFuncs[2];
};

//每个类都有一个属于自己的静态成员数组，存储了自己的成员函数的地址

```

```

HERO_FUNCS CMage::m_AryMageFuncs[2] =
{
    (PFN_HERO_SPEAK)&CMage::Speak,

    (PFN_HERO_SPEAK) (PFN_HERO_USESKILLS)&CMage::UseSkills
};

int main()
{
    //生成英雄
    cout << "请选择要初始化的英雄：(1-战士、2-法师)" << endl;
    CHero* pAryHeros[2] = { nullptr };

    //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
    //for (auto& pHero : pAryHeros)
    for (CHero*& pHero : pAryHeros)
    {
        int nIdx = 0;          //输入对应的数值，输出对应类对象数据
        cin >> nIdx;
        switch (nIdx)
        {
        case 1:
            pHero = new CWarrior;
            break;
        case 2:
            pHero = new CMage;
            break;
        default:
            break;
        }
    }

    //释放技能（输入对应的数值，输出对应的技能）
    while (true)
    {
        int nSkill = 0;
        cout << "输入0~4即可释放对应的技能：" << endl;
        cin >> nSkill;
        //pHero为pAryHeros起了一个别名，其类型为数组元素的类型：CHero*
        //for (auto& pHero : pAryHeros)
        for (CHero*& pHero : pAryHeros)
        {
            //符合多态
            //多态的定义：不关心子类对象的类型，调用属于其自己的成员函数
            //m_AryHeroFuncs数组只存储了父类的成员信息
            //(pHero->*(pHero->m_AryHeroFuncs[0].m_pfnHeroSpeak))();
            //(pHero->*(pHero->m_AryHeroFuncs[1].m_pfnHeroUseSkills))(nSkill);

```

```

        (pHero->*(pHero->m_pfnHeroFuncs[0].m_pfnHeroSpeak)) ();
        (pHero->*(pHero->m_pfnHeroFuncs[1].m_pfnHeroUseSkills)) (nSkill);

    cout << endl;
}
}

return 0;
}

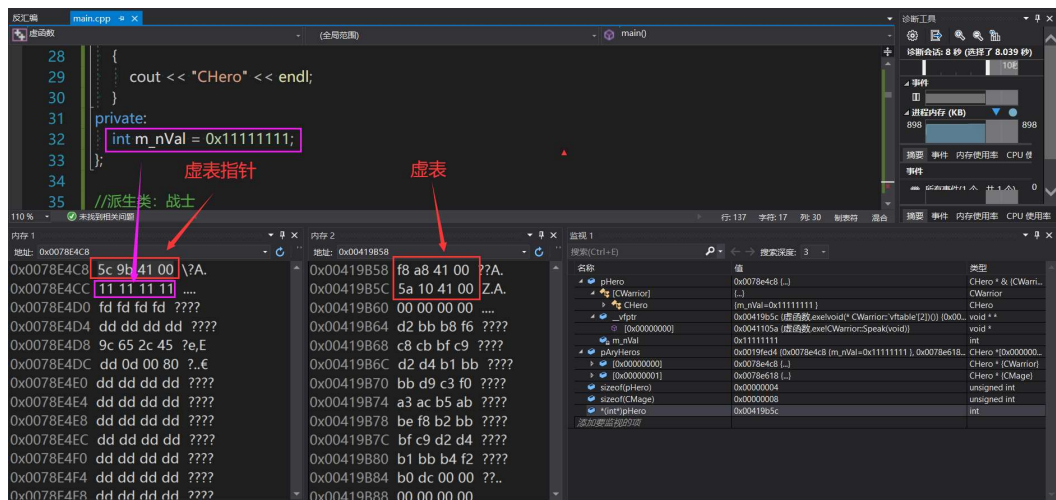
```

## 虚函数内部实现 虚表、虚表指针

虚函数数组 --> 虚表

指向数组的地址 --> 虚表指针

虚表指针放置的位置 --> 放置在类偏移为0的位置上



虚表和虚表指针赋值的时机：

父类子类各有各的虚表指针和虚表，各自的类把各自虚表赋值到各自对象的虚表指针上

虚表在编译期就构造完成

虚函数的内部实现：

内存情况:

\_\_vfptr: 虚表指针

vtable: 虚表

存储了成员函数的地址

CWarrior obj为例:



以CWarrior派生类为例:

编译期:

- 1、CHero 编译器构建虚表 vtable[] = {&CHero::UseSkills, &CHero::Speak}
- 2、CWarrior 编译器拷贝父类的虚表到子类  
vtable[] = {&CHero::UseSkills, &CHero::Speak}
- 3、CWarrior 编译器把子类中重写的父类的函数覆盖虚表中的对应项  
vtable[] = {&CWarrior::UseSkills, &CWarrior::Speak}
- 4、子类没有重写, 使用父类的虚表

运行期:

实例化一个战士的对象

- 1、CHero 构造 \_\_vfptr = CHero::vtable
- 2、CWarrior 构造 \_\_vfptr = CWarrior::vtable

使用:

pHero->UseSkills(nSkill);

先拿到虚表指针, 然后取虚表的第一项(数组下标寻址), 之后拿到虚表中UseSkills的地址, 接着调用  
(pHero->(\_\_vfptr[0]))(nSkill);

以CWarrior派生类为例:

编译期:

- 1、CHero 编译器构建虚表 vtable[] = {&CHero::UseSkills, &CHero::Speak}
- 2、CWarrior 编译器拷贝父类的虚表到子类  
vtable[] = {&CHero::UseSkills, &CHero::Speak}
- 3、CWarrior 编译器把子类中重写的父类的函数覆盖虚表中的对应项  
vtable[] = {&CWarrior::UseSkills, &CWarrior::Speak}
- 4、子类没有重写, 使用父类的虚表

运行期:

实例化一个战士的对象

- 1、CHero 构造 \_\_vfptr = CHero::vtable
- 2、CWarrior 构造 \_\_vfptr = CWarrior::vtable

使用:

pHero->UseSkills(nSkill);

先拿到虚表指针, 然后取虚表的第一项(数组下标寻址), 之后拿到虚表中UseSkills的地址, 接着调用

(pHero->(\_\_vfptr[0]))(nSkill);

# 多态

**多态**按字面的意思就是**多种形态**。

当类之间存在**层次结构**，并且类之间是**通过继承关联**时，就会用到多态。

C++ 多态意味着**调用成员函数时，会根据调用函数的对象的类型来执行不同的函数**。

在多态（虚函数中），派生类会重写基类的虚函数。

概念：

虚函数 是在基类中使用关键字 **virtual** 声明的函数。在派生类中重新定义基类中定义的虚函数时，会告诉编译器不要静态链接到该函数。我们想要的是在程序中任意点可以根据所调用的对象类型来选择调用的函数，这种操作被称为动态链接，或**后期绑定**。

虚函数的语法：

- 1、子类与父类的函数声明要一致
- 2、父类必须有虚函数（必须添加关键字），子类可以不添加关键字
- 3、使用父类的指针或者父类的引用, 才会有多态效果

