2020/05/15_C++_第9课_继承

笔记本: C++
创建时间: 2020/5/15 星期五 15:36
作者: ileemi
标签: 继承

C语言是如何解决代码重复的

继承解决的问题：代码重用(代码冗余、代码不通用)

C语言实现代码重用的简单展示：

```
#include <iostream>
//#include <stdio.h>
using namespace std;

//C语言代码封装变化
struct tagPoint
{
  int m_nX;
  int m_nY;
};
//打印坐标
void ShowPoint(tagPoint* pPt)
{
  //printf("Point X: %d Y: %d\r\n", pPt->m_nX, pPt->m_nY);
  printf("Point X: %d Y: %d", pPt->m_nX, pPt->m_nY);

}

//添加需求，增加3D坐标
struct tag3DPoint
{
  //int m_nX;
  //int m_nY;
```

```
    /*
    由于m_nX、m_nY上面之前写过，为了不使程序的代码过于冗余，
    可以将上面定义的结构体包含到这里来
    */
    tagPoint  m_n2DPoint;
    int     m_nZ;
};

//显示3D坐标
void Show3DPoint(tag3DPoint* p3Dpt)
{
    //printf("Point X: %d Y: %d\r\n", p3Dpt->m_n2DPoint.m_nX,
    //  p3Dpt->m_n2DPoint.m_nY, p3Dpt->m_nZ);
    //其实输出 点X和点Y轴的坐标的在上面做过，这里直接调用即可
    ShowPoint(&p3Dpt->m_n2DPoint);
    printf(" Z: %d\r\n", p3Dpt->m_nZ);

}

bool g_bFlag2D = true;
int main()
{
    //显示2D坐标
    //初始化坐标
    tagPoint pt1 = { 11,11 };
    tagPoint pt2 = { 21,12 };
    tagPoint pt3 = { 31,13 };
    tagPoint pt4 = { 41,14 };
    tagPoint pt5 = { 51,15 };
    tagPoint pt6 = { 61,16 };
    tagPoint pt7 = { 71,17 };
    tagPoint pt8 = { 81,18 };

    //此段代码不通用，如果需求更改，需要改动的比较大
    //printf("Point X: %d Y: %d\r\n", pt1.m_nX, pt1.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt2.m_nX, pt2.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt3.m_nX, pt3.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt4.m_nX, pt4.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt5.m_nX, pt5.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt6.m_nX, pt6.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt7.m_nX, pt7.m_nY);
    //printf("Point X: %d Y: %d\r\n", pt8.m_nX, pt8.m_nY);

    //显示2D坐标
    if (g_bFlag2D)
    {
        ShowPoint((tagPoint*)&pt1);
```

```
    ShowPoint((tagPoint*)&pt2);
    ShowPoint((tagPoint*)&pt3);
    ShowPoint((tagPoint*)&pt4);
    ShowPoint((tagPoint*)&pt5);
    ShowPoint((tagPoint*)&pt6);
    ShowPoint((tagPoint*)&pt7);
    ShowPoint((tagPoint*)&pt8);
  }
  else
  {
    //显示3D坐标
    ShowPoint(&pt1);
    ShowPoint(&pt2);
    ShowPoint(&pt3);
    ShowPoint(&pt4);
    ShowPoint(&pt5);
    ShowPoint(&pt6);
    ShowPoint(&pt7);
    ShowPoint(&pt8);
  }
  //初始化坐标
  //tagPoint pt1 = { 11,11 };
  //tagPoint pt2 = { 21,12 };
  //tagPoint pt3 = { 31,13 };
  //tagPoint pt4 = { 41,14 };
  //tagPoint pt5 = { 51,15 };
  //tagPoint pt6 = { 61,16 };
  //tagPoint pt7 = { 71,17 };
  //tagPoint pt8 = { 81,18 };

  return 0;
}
```

当我们定义相同属性的结构体的时候，会出现如下的代码冗余的情况：

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

//设计一个游戏

//该结构标识战士的基本数据信息
struct tagWarrior
{
        int m_nAttack;          //攻击力
        int m_nBlood;           //血量
        int m_nArmor;           //护甲
        int m_nMagicResistance;  //魔抗
```

```
        int m_nAnger;            //怒气，用于英雄放大招时使用
};

//对该战士的基本信息进行初始化
void InitWarrior(tagWarrior* pWarrior, int nAttack, int nBlood, int
nArmor, int nMR, int nAnger)
{
        pWarrior->m_nAttack = nAttack;
        pWarrior->m_nBlood = nBlood;
        pWarrior->m_nArmor = nArmor;
        pWarrior->m_nMagicResistance = nMR;
        pWarrior->m_nAnger = nAnger;
}

//该结构标识法师的基本数据信息
struct tagMage
{
        int m_nMaAttack; //攻击力
        int m_nBlood;            //血量
        int m_nArmor;            //护甲
        int m_nMagicResistance;  //魔抗
        int m_nMana;             //法力值
};

//对该法师的基本信息进行初始化
void InitWarrior(tagMage* pMage, int nMaAttack, int nBlood, int nArmor,
int nMR, int nMana)
{
        pMage->m_nMaAttack = nMaAttack;
        pMage->m_nBlood = nBlood;
        pMage->m_nArmor = nArmor;
        pMage->m_nMagicResistance = nMR;
        pMage->m_nMana = nMana;
}
int main()
{
        //战士
        tagWarrior warrior;
        InitWarrior(&warrior, 100, 1000, 150, 120, 0);
        printf("当前战士的基本属性为：攻击力：%d 血量：%d 护甲：%d 魔
抗：%d 怒气：%d\r\n",
    warrior.m_nAttack, warrior.m_nBlood, warrior.m_nArmor,
warrior.m_nMagicResistance, warrior.m_nAnger);

        //法师
        tagMage mage;
        InitWarrior(&mage, 150, 1000, 100, 200, 30);
```

```
        printf("当前战士的基本属性为：攻击力：%d 血量：%d 护甲：%d 魔
抗：%d 怒气：%d\r\n",
    mage.m_nMaAttack, mage.m_nBlood, mage.m_nArmor,
mage.m_nMagicResistance, mage.m_nMana);
        return 0;
}
```

提取相同属性的数据，减少程序中的代码冗余问题。
解决上述的代码冗余问题，我们可以将出现重复的数据进行封装，提取具有相同属性的数据，将其放在一个新的结构体中，然后后面的相关结构体使用的时候就可以将其包进去。

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

//设计一个游戏

//提取具有相同属性的数据
struct tagHero   //英雄
{
        int m_nBlood;           //血量
        int m_nArmor;           //护甲
        int m_nMagicResistance;  //魔抗
};

//对英雄的基本信息进行初始化
void InitHero(tagHero* pHero, int nBlood, int nArmor, int nMR)
{
        pHero->m_nBlood = nBlood;
        pHero->m_nArmor = nArmor;
        pHero->m_nMagicResistance = nMR;
}
//该结构标识战士的基本数据信息
struct tagWarrior
{
        tagHero m_hear;
        int m_nAttack;          //攻击力
        int m_nAnger;           //怒气，用于英雄放大招时使用
};

//对该战士的基本信息进行初始化
void InitWarrior(tagWarrior* pWarrior, int nAttack, int nBlood, int
nArmor, int nMR, int nAnger)
{
        InitHero((tagHero*)pWarrior, nBlood, nArmor, nMR);        //调用
之前初始化的数据
        pWarrior->m_nAttack = nAttack;
```

```
        pWarrior->m_nAnger = nAnger;
}

//该结构标识法师的基本数据信息
struct tagMage
{
        tagHero m_Hero;
        int m_nMaAttack;    //攻击力
        int m_nMana;            //法力值
};

//对该法师的基本信息进行初始化
void InitMage(tagMage* pMage, int nMaAttack, int nBlood, int nArmor, int
nMR, int nMana)
{
        InitHero((tagHero*)pMage, nBlood, nArmor, nMR);   //调用之前初始
化的数据
        pMage->m_nMaAttack = nMaAttack;
        pMage->m_nMana = nMana;
}
int main()
{
        //战士
        tagWarrior warrior;
        InitWarrior(&warrior, 100, 1000, 150, 120, 0);
        printf("当前战士的基本属性为：攻击力：%d 血量：%d 护甲：%d 魔
抗：%d 怒气：%d\r\n",
                warrior.m_nAttack, warrior.m_hear.m_nBlood,
                warrior.m_hear.m_nArmor,
warrior.m_hear.m_nMagicResistance, warrior.m_nAnger);

        //法师
        tagMage mage;
        InitMage(&mage, 150, 1000, 100, 200, 30);
        printf("当前战士的基本属性为：攻击力：%d 血量：%d 护甲：%d 魔
抗：%d 怒气：%d\r\n",
                mage.m_nMaAttack, mage.m_Hero.m_nBlood,
                mage.m_Hero.m_nArmor, mage.m_Hero.m_nMagicResistance,
mage.m_nMana);
        return 0;
}
```

使用C++继承语法实现代码重用的简单操作过程：

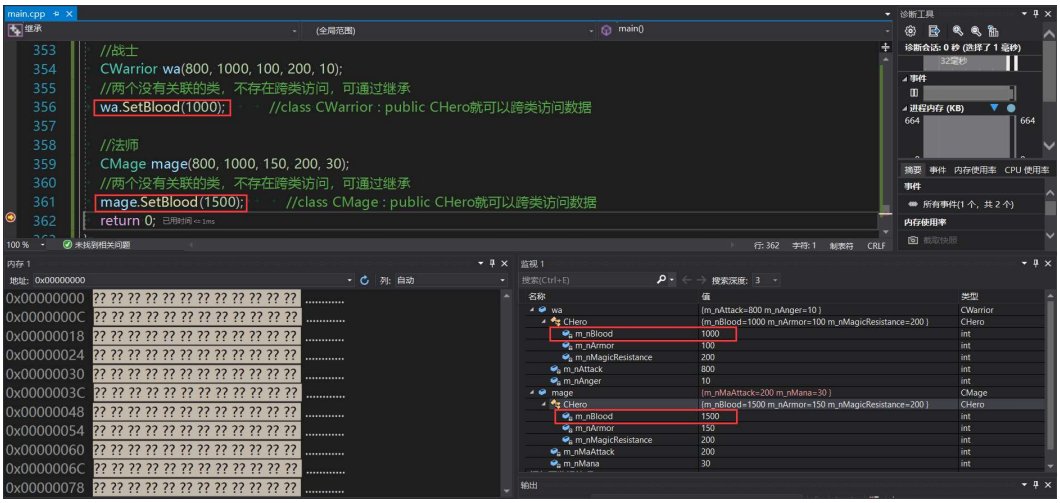
```
#include <iostream>
#include <time.h>
using namespace std;
```

```
class CHero      //英雄
{
public:
        //构造初始化数据
        //对英雄的基本信息进行初始化
        CHero(int nBlood, int nArmor, int nMR)
        {
                m_nBlood = nBlood;
                m_nArmor = nArmor;
                m_nMagicResistance = nMR;
        }
        //对数据成员提供Get Set方法
        //血量
        void SetBlood(int nBlood)
        {
                //判断英雄的血量是否小于0
                if (nBlood <= 0)
                {
                        m_nBlood = 0;
                        return;
                }
                m_nBlood = nBlood;
        }
        int GetBlood() const
        {
                return m_nBlood;
        }
        //护甲
        void SetArmor(int nArmor)
        {
                m_nArmor = nArmor;
        }
        int GetArmor() const
        {
                return m_nArmor;
        }
        //魔抗
        void SetMR(int nMR)
        {
                m_nMagicResistance = nMR;
        }
        int GetMR() const
        {
                return m_nMagicResistance;
        }
```

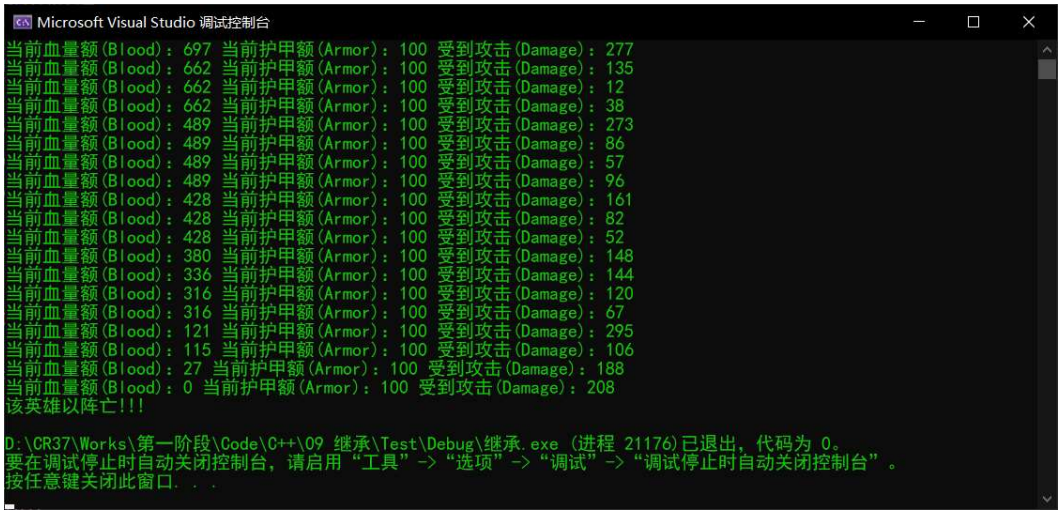
```
        //计算英雄受到攻击的伤害
        void Hurt(int nAttack)
        {
                //受到伤害减去护甲额
                int nDamage = nAttack - GetArmor();
                if (nDamage >= 0)
                {
                        SetBlood(GetBlood() - nDamage);          //从新
设置英雄血量
                }
        }
private:
        int m_nBlood;            //血量
        int m_nArmor;            //护甲
        int m_nMagicResistance;  //魔抗
};


//战士类
class CWarrior : public CHero
{
public:
        //使用构造对该战士的基本信息进行初始化
        CWarrior(int nAttack, int nBlood, int nArmor, int nMR, int
nAnger):
                CHero(nBlood, nArmor, nMR)
        {
                m_nAttack = nAttack;
                m_nAnger = nAnger;
        }
        //攻击力
        void SetAttack(int nAttack)
        {
                m_nAttack = nAttack;
        }
        int GetAttack() const
        {
                return m_nAttack;
        }
        //怒气
        void SetAnger(int nAnger)
        {
                m_nAnger = nAnger;
        }
        int GetAnger() const
        {
                return m_nAnger;
```

```
        }
private:
        int m_nAttack;          //攻击力
        int m_nAnger;           //怒气，用于英雄放大招时使用
};


//法师
class CMage : public CHero
{
public:
        //使用构造对该战士的基本信息进行初始化
        CMage(int nAttack, int nBlood, int nArmor, int nMR, int nMana):
                CHero(nBlood, nArmor, nMR)
        {
                m_nMaAttack = nMR;
                m_nMana = nMana;
        }
        //攻击力
        void SetMaAttack(int nMaAttack)
        {
                m_nMaAttack = nMaAttack;
        }
        int GetMaAttack() const
        {
                return m_nMaAttack;
        }
        //法力值
        void SetMana(int nMana)
        {
                m_nMana = nMana;
        }
        int GetMana() const
        {
                return m_nMana;
        }
private:
        int m_nMaAttack;        //攻击力
        int m_nMana;                    //法力值
};

int main()
{
        //战士
        CWarrior wa(800, 1000, 100, 200, 10);
        //两个没有关联的类，不存在跨类访问，可通过继承
        wa.SetBlood(1000);                      //class CWarrior :
```

```
public CHero就可以跨类访问数据

        //法师
        CMage mage(800, 1000, 150, 200, 30);
        //两个没有关联的类，不存在跨类访问，可通过继承
        mage.SetBlood(1500);                    //class CMage : public
CHero就可以跨类访问数据

        //英雄根据随机种子数据受到伤害
        srand(time(NULL));
        for (int i = 0; i < 100; i++)
        {
                int nDamage = rand() % 300;
                wa.Hurt(nDamage);

                //输出每次血量的变化
                //此时的血量有Bug，SetBlood处加个判断条件
                cout << "当前血量额(Blood)：" << wa.GetBlood()
                        << " 当前护甲额(Armor)：" << wa.GetArmor()
                        << " 受到攻击(Damage)：" << nDamage
                        << endl;
                //当该英雄的血量值等于0的时候，循环退出
                if (wa.GetBlood() <= 0)
                {
                        cout << "该英雄以阵亡!!!" << endl;
                        break;
                }
        }
        return 0;
}
```



数据仅供参考：



继承

面向对象程序设计中最重要的一个概念是继承。继承允许我们依据另一个类来定义一个类，这使得创建和维护一个应用程序变得更容易。这样做，也达到了重用代码功能和提高执行效率的效果。

当创建一个类时，您不需要重新编写新的数据成员和成员函数，只需指定新建的类继承了一个已有的类的成员即可。这个已有的类称为**基类（父类）**，新建的类称为**派生类（子类）**。
继承代表了 **is a** 关系。例如，越野车是车，公交车是车，哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。

1、继承相关的称呼：
基类 & 派生类：

一个类可以派生自多个类，这意味着，它可以从多个基类继承数据和函数。定义一个派生类，我们使用一个类派生列表来指定基类。类派生列表以一个或多个基类命名，形式如下：

```
class derived-class: access-specifier base-class
```

其中，访问修饰符 access-specifier 是 public、protected 或 private 其中的一个，base-class 是之前定义过的某个类的名称。如果未使用访问修饰符 access-specifier，则默认为 private。
如下：A 为基类，B 为 A 的派生类

```
class A
{};

class B : public A
{};
```

B 继承 A，A->父类(parent)，B->子类(child)
A 派生出 B，A->基类(base)，B->派生类(derive)

2、公有继承的权限

派生类可以访问基类中所有的非私有成员。因此基类成员如果不想被派生类的成员函数访问，则应在基类中声明为 private，根据访问权限可以总结出下面的不同的访问类型：

访问	同一个类是否可以访问	子类(派生类)是否可以访问	外部的类(类外)是否可以访问
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

一个派生类继承了所有的基类方法，但下列情况除外：

- 基类的构造函数、析构函数和拷贝构造函数
- 基类的重载运算符
- 基类的友元函数

扩展：关于继承的类型

当一个类派生自基类，该基类可以被继承为 public、protected 或 private 几种类型。

几乎不使用 **protected** 或 **private** 继承，通常使用 **public** 继承。当使用不同类型的继承时，遵循以下几个规则：

- **公有继承（public）**：当一个类派生自**公有**基类时，基类的**公有**成员也是派生类的**公有**成员，基类的**保护**成员也是派生类的**保护**成员，基类的**私有成员不能直接被派生类访问**，但是可以通过**调用基类的公有和保护成员来访问。**
- **保护继承（protected）**： 当一个类派生自**保护**基类时，基类的**公有**和**保护**成员将成为派生类的**保护**成员。
- **私有继承（private）**：当一个类派生自**私有**基类时，基类的**公有**和**保护**成员将成为派生类的**私有**成员。

代码示例：

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

class A
{
public:
        //构造
        A(int nValA) : m_nValA(nValA)
        {

        }
        int GetValA() const
        {
```

```
                return m_nValA;
        }
        void SetValA(int nValA)
        {
                m_nValA = nValA;
        }
        void Test()
        {
                //      类的内部可以访问内部的 protected中的数据成员
                m_nProtected = 999;
        }
protected:
        int m_nProtected;
private:
        int m_nValA;
};

class B : public A
{
public:
        //构造
        B(int nValA, int nValB) : A(nValA), m_nValB(nValB)
        {
                /*
                B(int nValB) : m_nValB(nValB)    //error C2512: “A”:没有合适的默认构造函数可用
                B(int nValA, int nValB) : A(nValA), m_nValB(nValB)
                */
        }
        int GetValB() const
        {
                return m_nValB;
        }
        void SetValB(int nValB)
        {
                m_nValB = nValB;
        }

        //B 继承 A，可以访问 A 中的共有成员
        void Test()
        {
                SetValA(999);
                //m_nValA = 666;
                //error C2248 : “A::m_nValA” : 无法访问 private 成员(在“A”类中声明)

                m_nProtected = 666;
```

```
        }
private:
        /*
        此时在main函数中通过对象不能进行访问
        error C2248: “B::Test”: 无法访问 private 成员(在“B”类中声
明)
        */
        //void Test()
        //{
        //      SetValA(999);
        //      //m_nValA = 666;
        //      //error C2248 : “A::m_nValA” : 无法访问 private 成员
(在“A”类中声明)
        //}
        // B 继承 A，可以访问 A 中的 protected 成员
        int m_nValB;
};

int main()
{
        B b(0xaaaaaaaa, 0xbbbbbbbb);
        b.Test();
        //error C2248: “A::m_nValA”: 无法访问 private 成员(在“A”类中
声明)
        //b.m_nValA = 999;

        //error C2248: “A::m_nProtected”: 无法访问 protected 成员
(在“A”类中声明)
        //b.error C2248: “A::m_nProtected”: 无法访问 protected 成员
(在“A”类中声明) = 666;

        A a(999);
        //error C2248: “A::m_nProtected”: 无法访问 protected 成员
(在“A”类中声明)
        //类外 类 A 的对象也无法访问 A中的受保护成员，父类中的受保护成
员只限子类调用使用
        //a.m_nProtected = 777;
        return 0;
}
```

注意：

- 类外 类 A 的对象也无法访问 A中的受保护成员，父类中的受保护成员只限子类调用使用

继承中的构造和析构的顺序

构造: 先父类, 再成员, 再子类

析构: 先子类, 再成员, 再父类

(成员和子类绑定在一起，成员先来，然后子类才能调用成员)

代码示例:

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

class CFoo
{
public:
        CFoo()
        {
                cout << "CFoo::CFoo()" << endl;
        }
        ~CFoo()
        {
                cout << "CFoo::~CFoo()" << endl;
        }
};

class A
{
public:
        A(int nA)
        {
                cout << "A::A()" << endl;
        }
        ~A()
        {
                cout << "A::~A()" << endl;
        }
};

class B :public A
{
public:
        B(int nValA, int nValB) :
                A(nValA)
        {
                cout << "B::B()" << endl;
        }
        ~B() { cout << "B::~B()" << endl; }

        CFoo m_foo;
```

```
};

int main()
{
  B b(1, 2);

  return 0;
  /*
  运行结果:
  A::A()
  CFoo::CFoo()
  B::B()
  B::~B()
  CFoo::~CFoo()
  A::~A()
  */
}
```

C++继承中的数据隐藏

- 当子类内含有和父类同名的成员时, 只会访问子类的
- 指定使用父类的成员 父类名::成员名

代码示例:

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

class A
{
public:
        void Test()
        {
                cout << "A::Test" << endl;
        }
        void TestA()
        {
                cout << "A::TestA" << endl;
        }
        int m_nValA;
};

class B :public A
{
public:
```

```
        void Test0()
        {
                m_nValA = 99;
                Test(); //将访问 B 中的Test()
                A::Test();  //如果需要访问 A 中的Test(),需要添加作用域
限定符
        }
        void Test()
        {
                cout << "B::Test" << endl;
        }
        int m_nValA;
};

int main()
{
        B b;
        b.Test0();  //此时访问 B 中的Test0

        B* p = &b;

        //调用 A 中的 Test方法
        p->A::Test();
        p->TestA();

        //调用 B 中的 Test方法
        p->Test();
        p->B::Test();


        A a;
        a.Test(); //访问基类A中的Test

        return 0;
}
```

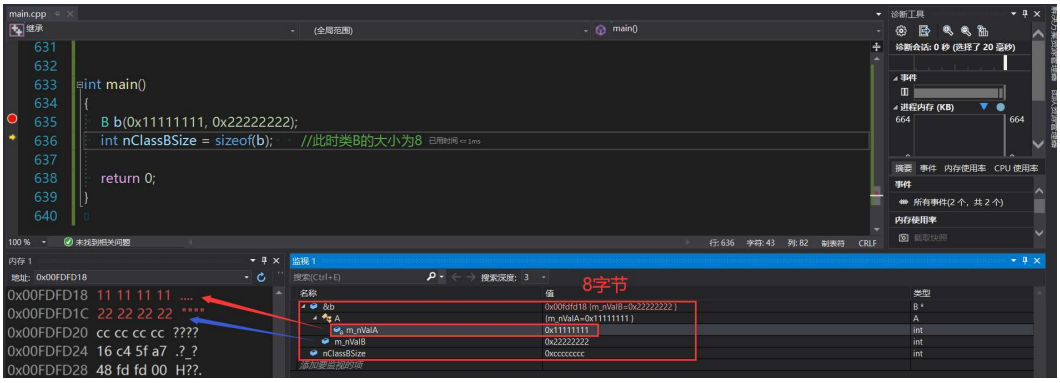
继承的内存排布内存

基类->成员->派生类

基于基类的派生类的内存大小，示例：

```
class A
{
public:
```

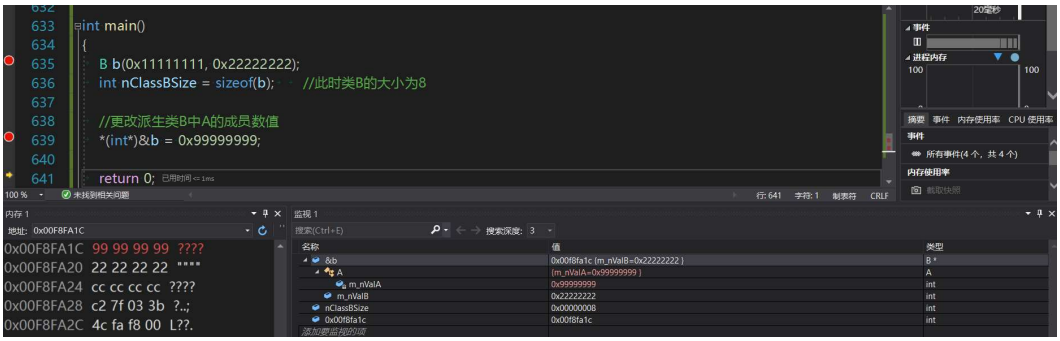
```
        A(int nA) :m_nValA(nA)
        {
        }
        void Test()
        {
        }
private:
        int m_nValA;
};

class B :public A
{
public:
        B(int nValA, int nValB) :
                A(nValA),
                m_nValB(nValB)
        {
        }
public:
        int m_nValB;
};

int main()
{
        B b(0x11111111, 0x22222222);
        int nClassBSize = sizeof(b); //此时类B的大小为8
    //派生类B继承的基类A的所有数据
        return 0;
}
```

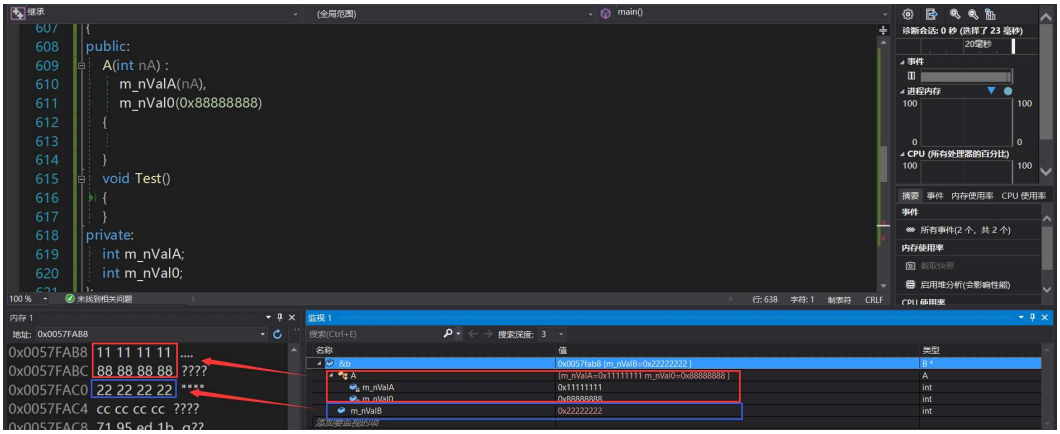


更改派生类B中基类A的成员数值

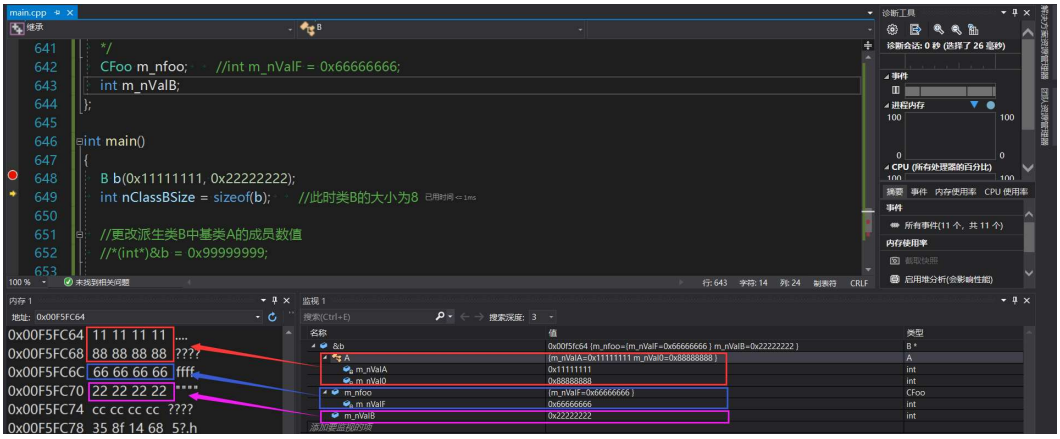
```
*(int*)&b = 0x99999999;
```

继承中数据成员的内存排布：





继承中，派生类中调用成员后，其内存分布如下：

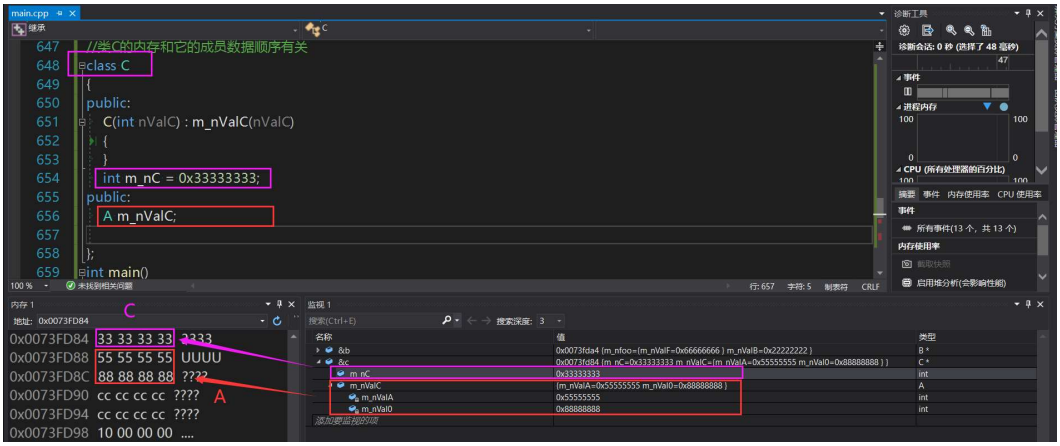


代码示例：

```
class CFoo
{
        int m_nValF = 0x66666666;
};

class A
{
public:
        A(int nA) :
                m_nValA(nA),
                m_nVal0(0x88888888)
        {

        }
        void Test()
```

```
        {
        }
private:
        int m_nValA;
        int m_nVal0;
};

class B :public A
{
public:
        B(int nValA, int nValB) :
                A(nValA),
                m_nValB(nValB)
        {
        }
public:
        /*
        在派生类B内定义一个成员，其属于B
        根据前面的构造析构循序可以得出，先基类A，在成员foo，在派生类B
        查看内存排布，进一步论证
        */
        CFoo m_nfoo; //int m_nValF = 0x66666666;
        int m_nValB;
};

int main()
{
        B b(0x11111111, 0x22222222);
        int nClassBSize = sizeof(b); //此时类B的大小为8

        //更改派生类B中基类A的成员数值
        //*(int*)&b = 0x99999999;

        return 0;
}
```

派生类包含基类成员，和派生类继承基类，代码如下：



子类转父类指针以及父类指针转子类的问题

结论：

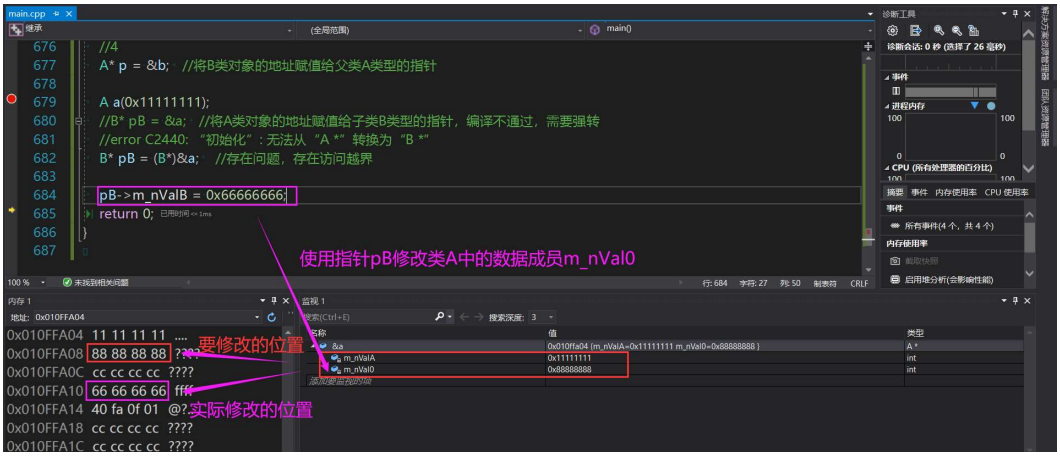
- **子类转父类指针是安全的**
- **父类指针转子类是不安全的**

论证：//4

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

class CFoo
{
        int m_nValF = 0x66666666;
};

class A
{
public:
        A(int nA) :
                m_nValA(nA),
                m_nVal0(0x88888888)
        {

        }
        void Test()
        {
                cout << "A:Test" << endl;
        }
private:
        int m_nValA;
        int m_nVal0;
};

class B :public A
{
public:
```

```
        B(int nValA, int nValB) :
                A(nValA),
                m_nValB(nValB)
        {
        }
public:
        /*
        在派生类B内定义一个成员，其属于B
        根据前面的构造析构循序可以得出，先基类A，在成员foo，在派生类B
        查看内存排布，进一步论证
        */
        CFoo m_nfoo;              //int m_nValF = 0x66666666;
        int m_nValB;
};

//类C的内存和它的成员数据顺序有关
class C
{
public:
        C(int nValC) : m_nValC(nValC)
        {
        }
        int m_nC = 0x33333333;
public:
        A m_nValC;

};
int main()
{
        //1
        B b(0x11111111, 0x22222222);
        int nClassBSize = sizeof(b);              //此时类B的大小为8

        //2
        //更改派生类B中基类A的成员数值
        //*(int*)&b = 0x99999999;

        //3
        //派生类对象b可以直接调用基类A的成员
        b.Test();
        //类 C 如何调类 A 中的成员
        C c(0x55555555);
        c.m_nValC.Test();

        //4
        //安全的
        A* p = &b;       //将B类对象的地址赋值给父类A类型的指针
```

```
        A a(0x11111111);
        //B* pB = &a;     //将A类对象的地址赋值给子类B类型的指针，编译不
通过，需要强转
        //error C2440: “初始化”: 无法从“A *”转换为“B *”
        //不安全，存在访问越界问题
        B* pB = (B*)&a;

        pB->m_nValB = 0x66666666;
        return 0;
}
```



什么时候使用继承比较合适

公交车, 出租车, 摩托车, 车
class 车;
class 公交车:public 车;
class 出租车:public 车;
class 摩托车:public 车;

冰箱, 洗衣机, 彩电, 家用电器
class 家用电器;
class 冰箱:public 家用电器;
class 洗衣机:public 家用电器;
class 彩电:public 家用电器;

公交车, 出租车, 摩托车, 飞机, 游轮
父类: 交通工具
子类: 公交车, 出租车, 摩托车, 飞机, 游轮

台式机, 主板, cpu, 内存条, 显卡
class 台式机
{
主板;
cpu;
内存条;

显卡;
};

笔记本, 台式机, 主板, cpu, 内存条, 显卡
class 计算机
{
主板;
cpu;
内存条;
显卡;
}
class 台式机 :public 计算机;
class 笔记本:public 计算机;

继承: is-a 是
组合: has-a 有

引擎, 轮子, 车架, 传动轴, 出租车, 公交车
class 车
{
引擎
轮子
车架
传动轴
}
class 出租车:public 车;
class 公交车:public 车;