项目需求：

因为各种不确定原因，包括认为原因，ODU设备会自动的切换到其它类型的设备，而切换后的设备，和原设备有很多不同的地方。如何完美的实现这个切换呢？

解决方案：

使用多态。

聚会案例：

Demo.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  class Father {  public:  void play() {  cout << "到KTV唱歌..." << endl;  }  };  class Son :public Father {  public:  void play() {  cout << "一起打王者吧！" << endl;  }  };  void party(Father \*\*men, int n) {  for (int i = 0; i<n; i++) {  men[i]->play();  }  }  int main(void) {  Father father;  Son son1, son2;  Father\* men[] = { &father, &son1, &son2 };  party(men, sizeof(men) / sizeof(men[0]));  system("pause");  return 0;  } |

解决方案:

通过虚函数，实现多态。

|  |
| --- |
| #include <iostream>  using namespace std;  class Father {  public:  **virtual** void play() {  cout << "到KTV唱歌..." << endl;  }  };  class Son :public Father {  public:  virtual void play() {  cout << "一起打王者吧！" << endl;  }  };  void party(Father \*\*men, int n) {  for (int i = 0; i<n; i++) {  men[i]->play();  }  }  int main(void) {  Father father;  Son son1, son2;  Father\* men[] = { &father, &son1, &son2 };  party(men, sizeof(men) / sizeof(men[0]));  system("pause");  return 0;  } |

# 项目精讲-实现多态：虚函数

多态的本质：

形式上，使用统一的父类指针做一般性处理，

但是实际执行时，这个指针可能指向子类对象，

形式上，原本调用父类的方法，但是实际上会调用子类的同名方法。

【注意】

程序执行时，父类指针指向父类对象，或子类对象时，在形式上是无法分辨的！

只有通过多态机制，才能执行真正对应的方法。

## 基础：虚函数的使用

虚函数的定义：

在函数的返回类型之前使用virtual

只在成员函数的声明中添加virtual, 在成员函数的实现中不要加virtual

虚函数的继承：

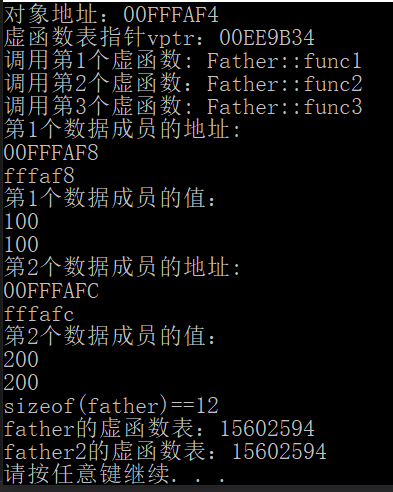
* 如果某个成员函数被声明为虚函数，那么它的子类【派生类】，以及子类的子类中，所继承的这个成员函数，也自动是虚函数。
* **如果在子类中重写这个虚函数，可以不用再写virtual, 但是仍建议写virtual, 更可读！**

## 进阶1：虚函数的原理-虚函数表

### 单个类的虚函数表

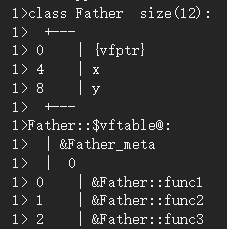
|  |
| --- |
| #include <iostream>  using namespace std;  class Father {  public:  virtual void func1() { cout << "Father::func1" << endl; }  virtual void func2() { cout << "Father::func2" << endl; }  virtual void func3() { cout << "Father::func3" << endl; }  void func4() { cout << "非虚函数：Father::func4" << endl; }  public: //为了便于测试，特别该用public  int x = 100;  int y = 200;  static int z;  };  typedef void (\*func\_t)(void);  int Father::z = 1;  int main(void) {  Father father;  // 含有虚函数的对象的内存中，最先存储的就是“虚函数表”  cout << "对象地址：" << (int\*)&father << endl;  int\* vptr = (int\*)\*(int\*)&father;  cout << "虚函数表指针vptr：" << vptr << endl;  cout << "调用第1个虚函数: ";  ((func\_t) \* (vptr + 0))();  cout << "调用第2个虚函数：";  ((func\_t) \* (vptr + 1))();  cout << "调用第3个虚函数: ";  ((func\_t) \* (vptr + 2))();    cout << "第1个数据成员的地址: " << endl;  cout << &father.x << endl;  cout << std::hex << (int)&father + 4 << endl;  cout << "第1个数据成员的值：" << endl;  cout << std::dec << father.x << endl;  cout << \*(int\*)((int)&father + 4) << endl;  cout << "第2个数据成员的地址: " << endl;  cout << &father.y << endl;  cout << std::hex << (int)&father + 8 << endl;  cout << "第2个数据成员的值：" << endl;  cout << std::dec << father.y << endl;  cout << \*(int\*)((int)&father + 8) << endl;  cout << "sizeof(father)==" << sizeof(father) << endl;  Father father2;  cout << "father的虚函数表：";  cout << \*(int\*)(\*(int\*)&father) << endl;  cout << "father2的虚函数表：";  cout << \*(int\*)(\*(int\*)&father2) << endl;  system("pause");  return 0;  } |

执行效果：

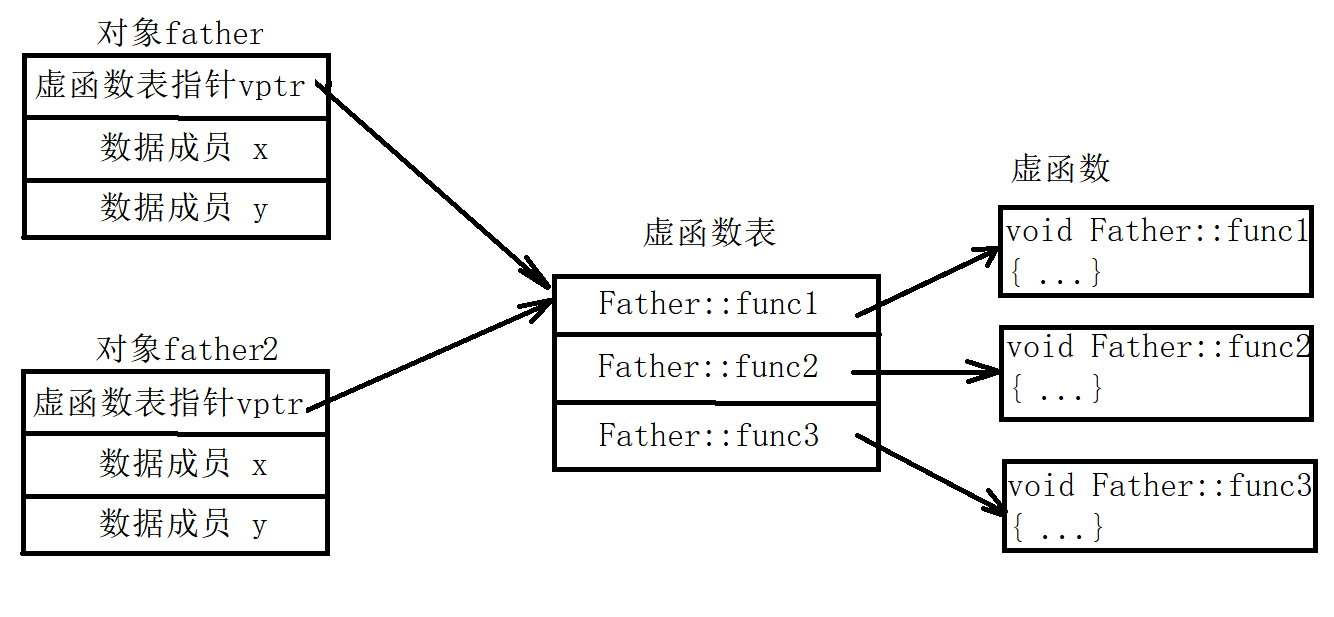


VS的对象内存分布分析：

项目的命令行配置中添加： /d1 reportSingleClassLayoutFather



手绘内存分布：



对象内，首先存储的是“虚函数表指针”，又称“虚表指针”。

然后再存储非静态数据成员。

对象的非虚函数，保存在类的代码中！

对象的内存，只存储虚函数表和数据成员

（类的静态数据成员，保存在数据区中，和对象是分开存储的）

添加虚函数后，对象的内存空间不变！仅虚函数表中添加条目

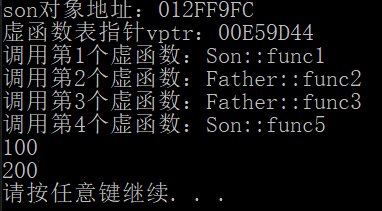
多个对象，共享同一个虚函数表！

### 使用继承的虚函数表

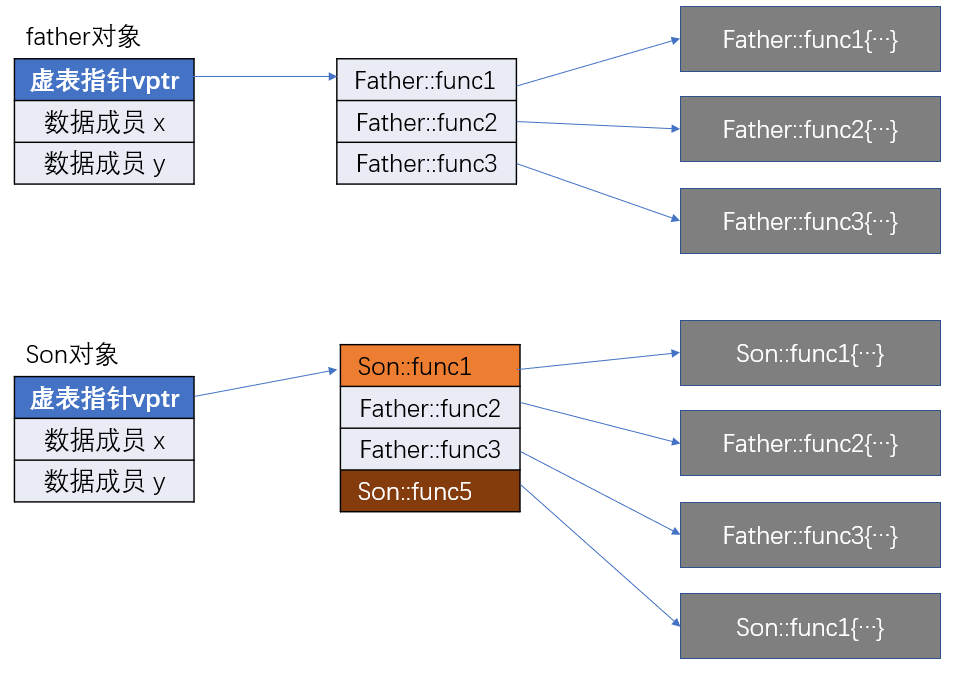
Demo.cpp

|  |
| --- |
| #include <iostream>  using namespace std;  class Father {  public:  virtual void func1() { cout << "Father::func1" << endl; }  virtual void func2() { cout << "Father::func2" << endl; }  virtual void func3() { cout << "Father::func3" << endl; }  void func4() { cout << "非虚函数：Father::func4" << endl; }  public: //为了便于测试，特别该用public  int x = 100;  int y = 200;  };  class Son : public Father {  public:  void func1() { cout << "Son::func1" << endl; }  virtual void func5() { cout << "Son::func5" << endl; }  };  typedef void (\*func\_t)(void);  int main(void) {  Father father;  Son son;  // 含有虚函数的对象的内存中，最先存储的就是“虚函数表”  cout << "son对象地址：" << (int\*)&son << endl;  int\* vptr = (int\*)\*(int\*)&son;  cout << "虚函数表指针vptr：" << vptr << endl;  for (int i = 0; i < 4; i++) {  cout << "调用第" << i + 1 << "个虚函数：";  ((func\_t) \* (vptr + i))();  }    for (int i = 0; i < 2; i++) {  // +4 是因为先存储了虚表指针  cout << \*(int\*)((int)&son + 4 + i \* 4) << endl;  }  system("pause");  return 0;  } |

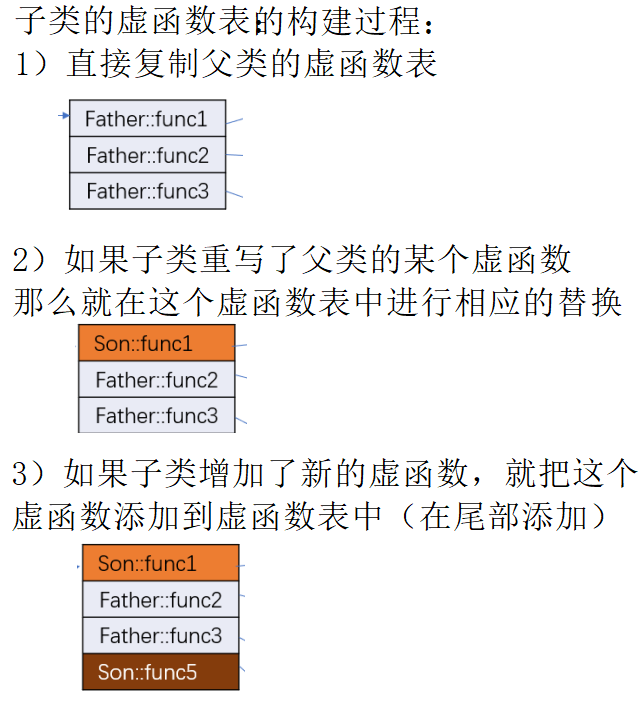
执行效果：



内存分布：



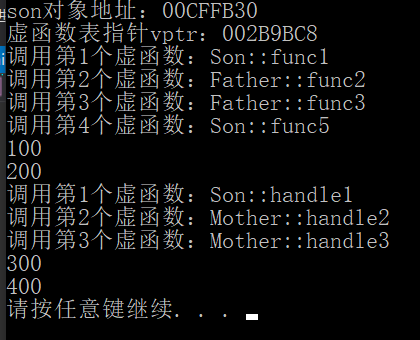
补充：



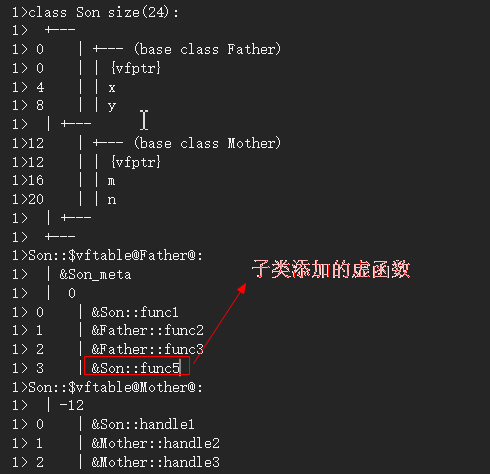
### 多重继承的虚函数表

|  |
| --- |
| #include <iostream>  using namespace std;  class Father {  public:  virtual void func1() { cout << "Father::func1" << endl; }  virtual void func2() { cout << "Father::func2" << endl; }  virtual void func3() { cout << "Father::func3" << endl; }  void func4() { cout << "非虚函数：Father::func4" << endl; }  public:  int x = 200;  int y = 300;  static int z;  };  class Mother {  public:  virtual void handle1() { cout << "Mother::handle1" << endl; }  virtual void handle2() { cout << "Mother::handle2" << endl; }  virtual void handle3() { cout << "Mother::handle3" << endl; }  public: //为了便于测试，使用public权限  int m = 400;  int n = 500;  };  class Son : public Father, public Mother {  public:  void func1() { cout << "Son::func1" << endl; }  virtual void handle1() { cout << "Son::handle1" << endl; }  virtual void func5() { cout << "Son::func5" << endl; }  };  int Father::z = 0;  typedef void(\*func\_t)(void);  int main(void) {  Son son;  int\* vptr = (int\*) \* (int\*)&son;  cout << "第一个虚函数表指针：" << vptr << endl;  for (int i = 0; i < 4; i++) {  cout << "调用第" << i + 1 << "个虚函数:";  ((func\_t) \* (vptr + i))();  }  for (int i = 0; i < 2; i++) {  cout << \*(int\*)((int)&son + 4 + i \* 4) << endl;  }  int\* vptr2 = (int\*) \* ((int\*)&son + 3);  for (int i = 0; i < 3; i++) {  cout << "调用第" << i + 1 << "个虚函数:";  ((func\_t) \* (vptr2 + i))();  }  for (int i = 0; i < 2; i++) {  cout << \*(int\*)((int)&son + 16 + i \* 4) << endl;  }  system("pause");  return 0;  } |

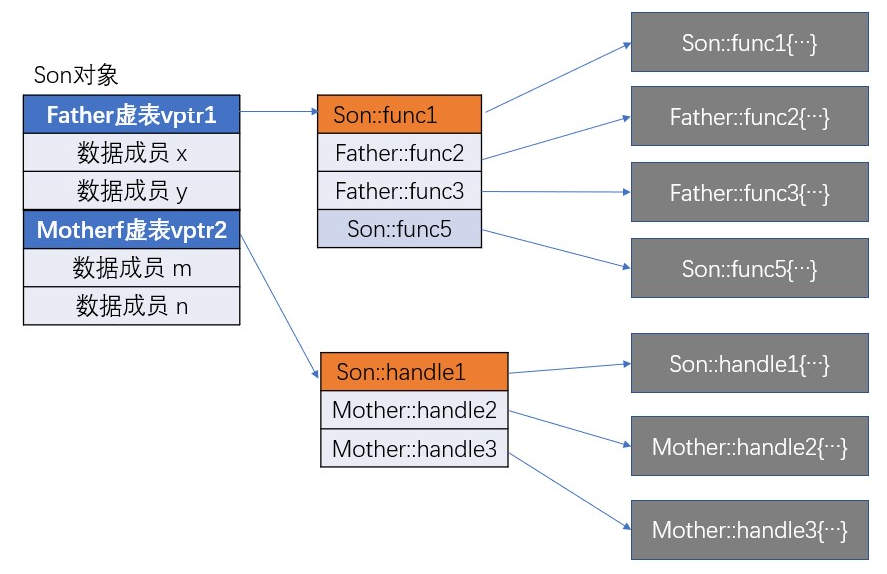
执行结果



VS分析：



内存分布



## 进阶2：final

用来修饰类，让该类不能被继承

理解：使得该类终结！

|  |
| --- |
| class XiaoMi {  public:  XiaoMi(){}  };  class XiaoMi2 **final** : public XiaoMi {  XiaoMi2(){}  };  class XiaoMi3 : public XiaoMi2 { //不能把XiaoMi2作为基类  }; |

用来修饰类的虚函数，使得该虚函数在子类中，不能被重写

理解：使得该功能终结！

|  |
| --- |
| class XiaoMi {  public:  virtual void func() final;  };  void XiaoMi::func() { //不需要再写final  cout << "XiaoMi::func" << endl;  }  class XiaoMi2 : public XiaoMi {  public:  void func() {}; // 错误！不能重写func函数  }; |

## 进阶3：override

override仅能用于修饰虚函数。

作用：

1. 提示程序的阅读者，这个函数是重写父类的功能。
2. 防止程序员在重写父类的函数时，把函数名写错。

|  |
| --- |
| #include <iostream>  using namespace std;  class XiaoMi {  public:  virtual void func() { cout << "XiaoMi::func" << endl; };  };  class XiaoMi2 : public XiaoMi {  public:  void func() override {}  //void func() override; 告诉程序员func是重写父类的虚函数  //void func1() override{} 错误！因为父类没有func1这个虚函数  };  int main(void) {  XiaoMi2 xiaomi;  return 0;  } |

override只需在函数声明中使用，不需要在函数的实现中使用。

# 项目精讲-遗失的子类析构函数

Demo：

|  |
| --- |
| #include <iostream>  #include <Windows.h>  #include <string.h>  using namespace std;  class Father {  public:  Father(const char\* addr ="中国"){  cout << "执行了Father的构造函数" << endl;  int len = strlen(addr) + 1;  this->addr = new char[len];  strcpy\_s(this->addr, len, addr);  }  // 把Father类的析构函数定义为virtual函数时，  // 如果对 Father类的指针使用delete操作时，  // 就会对该指针使用“动态析构”：  // 如果这个指针，指向的是子类对象，  // 那么会先调用该子类的析构函数，再调用自己类的析构函数  virtual ~Father(){  cout << "执行了Father的析构函数" << endl;  if (addr) {  delete addr;  addr = NULL;  }  }  private:  char\* addr;  };  class Son :public Father {  public:  Son(const char \*game="吃鸡", const char \*addr="中国")  :Father(addr){  cout << "执行了Son的构造函数" << endl;  int len = strlen(game) + 1;  this->game = new char[len];  strcpy\_s(this->game, len, game);  }  ~Son(){  cout << "执行了Son的析构函数" << endl;  if (game) {  delete game;  game = NULL;  }  }  private:  char\* game;  };  int main(void) {  cout << "----- case 1 -----" << endl;  Father\* father = new Father();  delete father;  cout << "----- case 2 -----" << endl;  Son\* son = new Son();  delete son;  cout << "----- case 3 -----" << endl;  father = new Son();  delete father;  system("pause");  return 0;  } |

【注意】

为了防止内存泄露，最好是在基类析构函数上添加virtual关键字，使基类析构函数为虚函数

目的在于，当使用delete释放基类指针时，会实现动态的析构：

如果基类指针指向的是基类对象，那么只调用基类的析构函数

如果基类指针指向的是子类对象，那么先调用子类的析构函数，再调用父类的析构函数

# 第5节 项目精讲-纯虚也有用：纯虚函数与抽象类

## 什么时候使用纯虚函数

某些类，在现实角度和项目实现角度，都不需要实例化（不需要创建它的对象），

这个类中定义的某些成员函数，只是为了提供一个形式上的接口，准备让子类来做具体的实现。

此时，这个方法，就可以定义为“纯虚函数”， 包含纯虚函数的类，就称为抽象类。

## 纯虚函数的使用方法

用法：纯虚函数，使用virtual和 =0

Demo

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class Shape {  public:  Shape(const string& color = "white") { this->color = color; }  virtual float area() = 0; //不用做具体的实现  string getColor() { return color; }  private:  string color;  };  class Circle : public Shape {  public:  Circle(float radius = 0, const string& color="White")  :Shape(color), r(radius){}  float area();  private:  float r; //半径  };  float Circle::area() {  return 3.14 \* r \* r;  }  int main() {  //使用抽象类创建对象非法！  //Shape s;  Circle c1(10);  cout << c1.area() << endl;  Shape\* p = &c1;  cout << p->area() << endl;  system("pause");  return 0;  } |

## 纯虚函数的注意事项：

父类声明某纯虚函数后，

那么它的子类，

1. 要么实现这个纯虚函数 （最常见）
2. 要么继续把这个纯虚函数声明为纯虚函数，这个子类也成为抽象类
3. 要么不对这个纯虚函数做任何处理，等效于上一种情况（该方式不推荐）

# 第6节 英语不是障碍：计算机英语加油站

|  |  |
| --- | --- |
| virtual | 虚拟的 |
| final | 最后的 |
| override | 覆盖，覆写 |

# 项目实现

ODU.h

|  |
| --- |
| #pragma once  #include <string>  #define ODU\_TYPE\_331\_FLAG "331"  #define ODU\_TYPE\_335\_FLAG "335"  enum class ODU\_TYPE {  ODU\_TYPE\_331,  ODU\_TYPE\_335,  ODU\_TYPE\_UNKNOWN  };  // ODU类，用于处理老型号ODU331设备  class ODU  {  public:  ODU();  virtual int getTxFre(); //获取发射频率  virtual bool setTxFre(int); //设置发射频率  virtual int getRxFre(); //获取接收频率  virtual bool setRxFre(int); //获取接收频率  virtual float getTxPower(); //获取发射功率  virtual bool setTxPower(float); //设置发射功率  virtual float getRxL(); //获取接收电平  virtual bool heartBeat(); //心跳包  virtual std::string name(); //获取该设备的名称  ODU\_TYPE getODUType(); //获取当前ODU的类型  protected:  int txFre; //发射频率  int rxFre; //接收频率  float txPower; //发射功率  float rxL; //接收电平  ODU\_TYPE type;  }; |

ODU.cpp

|  |
| --- |
| #include "ODU.h"  #include <iostream>  #include <string>  using namespace std;  ODU::ODU()  {  txFre = 34400;  rxFre = 31100;  txPower = 20;  rxL = 0;  type = ODU\_TYPE::ODU\_TYPE\_331;  cout << "调用ODU()" << endl;  }  int ODU::getTxFre()  {  return txFre;  }  bool ODU::setTxFre(int frequence)  {  txFre = frequence;  cout << name() << "发射频率已经设置为" << txFre << "Hz" << endl;  return true;  }  int ODU::getRxFre()  {  return rxFre;  }  bool ODU::setRxFre(int frequence)  {  rxFre = frequence;  cout << name() << "接收频率已经设置为" << rxFre << "Hz" << endl;  return true;  }  float ODU::getTxPower()  {  return txPower;  }  bool ODU::setTxPower(float power)  {  txPower = power;  cout << name() << "发射功率已经设置为" << txPower << "dBm" << endl;  return true;  }  float ODU::getRxL()  {  return rxL;  }  bool ODU::heartBeat()  {  cout << name() << "模拟串口协议读取数据：获取心跳包的反馈...【"  << ODU\_TYPE\_331\_FLAG << "】";    string response;  cin >> response;  bool ret = false;  if (response == ODU\_TYPE\_331\_FLAG) {  type = ODU\_TYPE::ODU\_TYPE\_331;  ret = true;  }    return ret;  }  std::string ODU::name()  {  string ret;  switch (type) {  case ODU\_TYPE::ODU\_TYPE\_331:  ret = "ODU331";  break;  case ODU\_TYPE::ODU\_TYPE\_335:  ret = "ODU335";  break;  case ODU\_TYPE::ODU\_TYPE\_UNKNOWN:  default:  ret = "ODU\_UNKNOWN";  break;  }  return ret;  }  ODU\_TYPE ODU::getODUType()  {  return type;  } |

ODU335.h

|  |
| --- |
| #pragma once  #include "ODU.h"  class ODU335 :  public ODU  {  public:  ODU335();  bool heartBeat();  }; |

ODU335.cpp

|  |
| --- |
| #include "ODU335.h"  #include <iostream>  using namespace std;  ODU335::ODU335()  {  cout << "调用ODU335()" << endl;  type = ODU\_TYPE::ODU\_TYPE\_335;  }  bool ODU335::heartBeat()  {  cout << name() << "模拟串口协议读取数据：获取心跳包的反馈...【"  << ODU\_TYPE\_335\_FLAG << "】";  string response;  cin >> response;  bool ret = false;  if (response == ODU\_TYPE\_335\_FLAG) {  type = ODU\_TYPE::ODU\_TYPE\_335;  ret = true;  }  return ret;  } |

Main.cpp

|  |
| --- |
| #include <iostream>  #include <Windows.h>  #include <thread>  #include "ODU.h"  #include "ODU335.h"  using namespace std;  ODU\* odu = NULL;  void test(ODU\* odu) {  odu->setTxFre(35588);  cout << odu->getTxFre() << endl;  odu->heartBeat();  }  void oduMonitorHandler() {  while (1) {  if (odu->heartBeat() == false) {  // 切换ODU  ODU\_TYPE type = odu->getODUType();  switch (type) {  case ODU\_TYPE::ODU\_TYPE\_331:  delete odu;  odu = new ODU335;  break;  case ODU\_TYPE::ODU\_TYPE\_335:  delete odu;  odu = new ODU;  break;  default:  odu = NULL;  return;  }  }  Sleep(3000);  }  }  int main(void) {  odu = new ODU();  // 创建一个线程，用于对ODU进行监测  std::thread oduMonitor(oduMonitorHandler);  // 主线程等待线程oduMonitor的结束  oduMonitor.join();  return 0;  } |

# 常见错误总结

1. 虚函数的函数原型

子类在重新实现继承的虚函数时，要和主要函数的原型一致

如果已经继承虚函数：

bool heartBeat();

那么重写虚函数时，函数原型必须保持完全一致：

bool heartBeat();

而且子类不能添加：  
 int heartBeat();

//因为仅函数的返回类型不同时，不能区别两个函数。

但是可以添加：  
 int heartBeat(int);

1. 析构函数是否使用虚函数

有子类时，析构函数就应该使用虚函数