1. 静态联编和动态联编
   1. 多态分类
      1. 静态多态 函数重载
      2. 动态多态 虚函数 继承关系
   2. 静态联编
      1. 地址早绑定 编译阶段绑定好地址
   3. 动态联编
      1. 地址晚绑定 ，运行时候绑定好地址
   4. 多态
      1. 父类的引用或指针指向子类对象

|  |
| --- |
| class Animal  {  public:  virtual void speak()  {  cout << "动物在说话" << endl;  }  virtual void eat()  {  cout << "动物在吃饭" << endl;  }  };  class Cat :public Animal  {  public:  void speak()  {  cout << "小猫在说话" << endl;  }  virtual void eat()  {  cout << "小猫在吃鱼" << endl;  }  };  //调用doSpeak ，speak函数的地址早就绑定好了，早绑定，静态联编，编译阶段就确定好了地址  //如果想调用猫的speak，不能提前绑定好函数的地址了，所以需要在运行时候再去确定函数地址  //动态联编，写法 doSpeak方法改为虚函数,在父类上声明虚函数，发生了多态  // 父类的引用或者指针 指向 子类对象  void doSpeak(Animal & animal) //Animal & animal = cat  {  animal.speak();  }  //如果发生了继承的关系，编译器允许进行类型转换  void test01()  {  Cat cat;  doSpeak(cat);  } |

1. 多态原理解析
   1. 当父类中有了虚函数后，内部结构就发生了改变
   2. 内部多了一个 vfprt
      1. virtual function pointer 虚函数表指针
      2. 指向 vftable 虚函数表
   3. 父类中结构 vfptr &Animal::speak
   4. 子类中 进行继承 会继承 vfptr vftable
   5. 构造函数中 会将虚函数表指针 指向自己的虚函数表
   6. 如果发生了重写，会替换掉虚函数表中的原有的speak，改为 &Cat::speak
   7. 深入剖析，内部到底如何调用
   8. ((void(\*)()) (\*(int\*)\*(int\*)animal))();
   9. 猫吃鱼的函数调用（编译器的调用）

|  |
| --- |
| void test02()  {  //cout << sizeof(Animal) << endl;  //父类指针指向子类对象 多态  Animal \* animal = new Cat;  //animal->speak();  // \*(int\*)\*(int\*)animal 函数地址  ((void(\*)()) (\*(int\*)\*(int\*)animal))();  // \*((int\*)\*(int\*)animal+1)猫吃鱼的地址  ((void(\*)()) (\*((int\*)\*(int\*)animal + 1)))();  } |

1. 多态案例 – 计算器案例
   1. 早期方法 是不利于扩展
   2. 开发有原则 开闭原则 -- 对扩展开放 对修改关闭
   3. 利用多态实现 – 利于后期扩展，结构性非常好，可读性高， 效率稍微低，发生多态内部结构复杂

|  |
| --- |
| //利用多态实现计算器  class abstractCalculator  {  public:  //虚函数 virtual int getResult(){ return 0; };  //纯虚函数  //如果父类中有了 纯虚函数 子类继承父类，就必须要实现 纯虚函数  //如果父类中 有了 纯虚函数 ，这个父类 就无法实例化对象了  //这个类有了纯虚函数，通常又称为 抽象类  virtual int getResult() = 0;  void setv1(int v)  {  this->val1 = v;  }  void setv2(int v)  {  this->val2 = v;  }  public:  int val1;  int val2;  };  //如果父类中有了 纯虚函数 子类继承父类，就必须要实现 纯虚函数  class A:public abstractCalculator  {  public:  virtual int getResult()  {  return 0;  }  };  //加法计算器  class PlusCalculator :public abstractCalculator  {  public:  virtual int getResult()  {  return val1 + val2;  };  };  class SubCalculator : public abstractCalculator  {  public:  virtual int getResult()  {  return val1 - val2;  };  };  class ChengCalculator :public abstractCalculator  {  public:  virtual int getResult()  {  return val1 \* val2;  };  };  void test02()  {  abstractCalculator \* abc ;  //加法计算器  abc = new PlusCalculator;  abc->setv1(10);  abc->setv2(20);  cout << abc->getResult() << endl;  delete abc;  abc = new SubCalculator;  abc->setv1(10);  abc->setv2(20);  cout << abc->getResult() << endl;  delete abc;  abc = new ChengCalculator;  abc->setv1(10);  abc->setv2(20);  cout << abc->getResult() << endl;  //如果父类有了纯虚函数，不能实例化对象了  /\*abstractCalculator aaa;  abstractCalculator \* abc = new abstractCalculator;\*/  } |

1. 抽象类 和 纯虚函数
   1. 纯虚函数写法 virtual void func() = 0;
   2. 抽象类型
   3. 抽象类 不可以实例化对象
   4. 如果类 继承了抽象类， 必须重写抽象类中的纯虚函数
2. 虚析构和纯虚析构
   1. 虚析构
      1. virtual ~类名() {}
      2. 解决问题： 通过父类指针指向子类对象释放时候不干净导致的问题
   2. 纯虚析构函数
      1. 写法 virtual ~类名() = 0
      2. 类内声明 类外实现
      3. 如果出现了纯虚析构函数，这个类也算抽象类，不可以实例化对象

|  |
| --- |
| class Animal  {  public:  virtual void speak()  {  cout << "动物在说话" << endl;  }  //普通析构 是不会调用子类的析构的，所以可能会导致释放不干净  //利用虚析构来解决这个问题  //virtual ~Animal()  //{  // cout << "Animal的析构调用" << endl;  //}  //纯虚析构 写法如下  //纯虚析构 ，需要声明 还需要实现 类内声明，类外实现  virtual ~Animal() = 0;  //如果函数中出现了 纯虚析构函数，那么这个类也算抽象类  //抽象类 不可实例化对象  };  Animal::~Animal()  {  //纯虚析构函数实现  cout << "Animal的纯虚析构调用" << endl;  }  // 如果出现纯虚析构，类也算抽象类，不能实例化对象  //void func()  //{  // Animal an;  // Animal \* animal = new Animal;  //}  class Cat:public Animal  {  public:  Cat(const char \* name)  {  this->m\_Name = new char[strlen(name) + 1];  strcpy(this->m\_Name, name);  }  virtual void speak()  {  cout << "小猫在说话" << endl;  }  ~Cat()  {  cout << "Cat的析构调用" << endl;  if (this->m\_Name !=NULL)  {  delete[] this->m\_Name;  this->m\_Name = NULL;  }  }  char \* m\_Name;  };  void test01()  {  Animal \* animal = new Cat("TOM");  animal->speak();  delete animal;  } |

1. 向上类型转换和向下类型转换
   1. 基类转派生类
      1. 向下类型转换 不安全的
   2. 派生类转 基类
      1. 向上类型转换 安全
   3. 如果发生多态
      1. 总是安全的
   4. 父类中如果写了虚函数，而子类没有任何重写，有意义吗？
      1. 没有意义