1. 关系运算符重载
   1. 自定义数据类型 不会内部做 比较 == ！=
   2. 所以要重载 == ！=

|  |
| --- |
| class Person  {  public:  Person(string name, int age)  {  this->m\_Name = name;  this->m\_Age = age;  }  bool operator==( Person & p)  {  if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)  {  return true;  }  return false;  }  bool operator!=( Person & p)  {  if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)  {  return false;  }  return true;  }  public:  string m\_Name;  int m\_Age;  }; |

1. 函数调用运算符重载
   1. () 仿函数 对象() 看似像函数调用

|  |
| --- |
| class MyPrint  {  public:  void operator()( string text)  {  cout << text << endl;  }  };  void test01()  {  MyPrint myPrint;  myPrint("hello world1111"); // 仿函数  } |

* 1. MyAdd() 匿名对象

|  |
| --- |
| class MyAdd  {  public:  int operator()(int v1,int v2)  {  return v1 + v2;  }  };  void test02()  {  //MyAdd myAdd;  //cout << myAdd(1, 1) << endl;  cout << MyAdd()(1, 1) << endl; //匿名对象  } |

1. 不要重载 && 和 ||
   1. 本身有短路特性我们无法去实现这种特性
   2. 所以不要重载
2. 强化训练 –字符串封装
   1. cout 输入 自定义的字符串
   2. cin 让用户输入字符串内容
   3. 重载 = 运算符 再次复习练习
   4. 重载 + 运算符
   5. 重载 [] 运算符
   6. 重载 == 运算符

|  |
| --- |
| MyString.h文件 class MyString  {  friend ostream& operator<< (ostream& cout, MyString & str);  friend istream& operator>> (istream& cin, MyString & str);  public:  MyString(const char \* str);  MyString(const MyString & str);  ~MyString();  //重载=运算符  MyString& operator = (const char \* str);  MyString& operator=(const MyString & str);    //重载[]运算符  char& operator[](int index);  //重载 +运算符  MyString operator+(const char \* str);  MyString operator+(const MyString& str);  //重载 == 运算符  bool operator==(const char \* str);  bool operator==(const MyString & str);  private:  char \* pString; //执行堆区的指针  int m\_Size; // 字符串大小  }; |

|  |
| --- |
| MyString.cpp文件  #include "MyString.h"  ostream& operator<< (ostream& cout, MyString & str)  {  cout << str.pString;  return cout;  }  //右移运算符重载  istream& operator>> (istream& cin, MyString & str)  {  //先判断 原始是否有内容，如果有 清空  if (str.pString != NULL)  {  delete [] str.pString;  str.pString = NULL;  }  //让用户输入内容  char buf[1024];  cin >> buf;  //把用户输入的字符串 赋值给 str  str.pString = new char[strlen(buf) + 1];  strcpy(str.pString, buf);  str.m\_Size = strlen(buf);  return cin;  }  MyString::MyString(const char \* str)  {  //cout << "有参构造调用" << endl;  this->pString = new char[strlen(str) + 1];  strcpy(this->pString, str);  this->m\_Size = strlen(str);  }  MyString::MyString(const MyString & str)  {  this->pString = new char[strlen(str.pString) + 1];  strcpy(this->pString, str.pString);  this->m\_Size = str.m\_Size;  }  MyString::~MyString()  {  //cout << "析构函数调用" << endl;  if (this->pString != NULL)  {  delete[] this->pString;  this->pString = NULL;  }  }  char& MyString::operator[](int index)  {  return this->pString[index];  }  MyString& MyString::operator=(const char \* str)  {  if (this->pString!=NULL)  {  delete[] this->pString;  this->pString = NULL;  }  this->pString = new char[strlen(str) + 1];  strcpy(this->pString, str);  return \*this;  }  MyString& MyString::operator=(const MyString & str)  {  if (this->pString != NULL)  {  delete[] this->pString;  this->pString = NULL;  }  this->pString = new char[strlen(str.pString) + 1];  strcpy(this->pString, str.pString);  return \*this;  }  MyString MyString::operator+(const char \* str)  {  //计算返回的字符串开辟的大小  int newSize = this->m\_Size + strlen(str) + 1;  char \* tmp = new char[newSize];  memset(tmp, 0, newSize);  //拼接字符串  strcat(tmp, this->pString);  strcat(tmp, str);  MyString newStr(tmp);  delete[] tmp;  return newStr;  }  MyString MyString::operator+(const MyString& str)  {  int newSize = this->m\_Size + strlen(str.pString) + 1;  char \* tmp = new char[newSize];  memset(tmp, 0, newSize);  //拼接字符串  strcat(tmp, this->pString);  strcat(tmp, str.pString);  MyString newStr(tmp);  delete[] tmp;  return newStr;  }  bool MyString::operator==(const char \* str)  {  if ( strcmp( this->pString , str) == 0 && this->m\_Size == strlen(str) )  {  return true;  }  return false;    }  bool MyString::operator==(const MyString & str)  {  if (strcmp(this->pString, str.pString) == 0 && this->m\_Size == strlen(str.pString))  {  return true;  }  return false;  } |

|  |
| --- |
| 测试  //测试 MyString  void test01()  {  MyString str = "abc";  cout << str << endl;  /\*cout << "请输入str新的内容：" << endl;  cin >> str;  cout << "新内容为：" << str << endl;\*/  MyString str2(str);  MyString str3 = "aaaaaa";  str3 = str2;  str3 = "aaaa";  cout << "str3 = " << str3 << endl;  str3[0] = 'w';  cout << "str3 第一个位置为 = " << str3[0] << endl;  MyString str4 = "";  str4 = str2 + str3; //字符串拼接  cout << "str4 为 " << str4 << endl;  if (str3 == str4)  {  cout << "str3 与 str4相等" << endl;  }  else  {  cout << "str3 与 str4不相等" << endl;  }  /\*int a = 10;  cin >> a;  cout << "a = " << a << endl;\*/  } |

1. 继承的引出
   1. 网页 很多公共部分
   2. 导致实现时候有很多重复的代码
   3. 引出继承，基类 （父类） 公共网页
   4. 具体子类 实现不同的内容
   5. 语法 class 子类 :继承方式 父类

|  |
| --- |
| //继承写法  //抽象一个 基类的网页 重复的代码都写到这个网页上  class BasePage  {  public:  void header()  {  cout << "公共头部" << endl;  }  void footer()  {  cout << "公共底部" << endl;  }  void left()  {  cout << "左侧列表" << endl;  }  };  class News :public BasePage //继承 News类 继承于 BasePage类  {  public:  void content()  {  cout << "新闻播放" << endl;  }  };  class YULE :public BasePage  {  public:  void content()  {  cout << "白百合。。。" << endl;  }  };  class Game :public BasePage  {  public:  void content()  {  cout << "KPL直播" << endl;  }  }; |

1. 继承方式
   1. 不管公有继承 保护 还是私有 基类中的私有属性 ，都不可以继承下去
   2. 公有继承
      1. 父类中的protected 在子类中是 protected
      2. 父类中的public 在子类中是 public

|  |
| --- |
| class Base1  {  public:  int m\_A;  protected:  int m\_B;  private:  int m\_C;  };  //公有继承  class Son1 : public Base1  {  public:  void func()  {  //cout << m\_C << endl; //基类中私有的属性 不可继承  cout << m\_A << endl; //基类中公有的属性 可继承，还是public  cout << m\_B << endl;//基类中保护的属性 可继承，还是protected 类外访问不到  }  }; |

* 1. 保护继承
     1. 父类中的protected 在子类中是 protected
     2. 父类中的public 在子类中是 protected

|  |
| --- |
| class Base2  {  public:  int m\_A;  protected:  int m\_B;  private:  int m\_C;  };  class Son2 :protected Base2  {  public:  void func()  {  //cout << m\_C << endl; //基类中私有的属性 不可继承  cout << m\_A << endl;//基类中公有的属性 可继承，还是protected  cout << m\_B << endl;//基类中保护的属性 可继承，还是protected  }  }; |

* 1. 私有继承
     1. 父类中的protected 在子类中是 private
     2. 父类中的public 在子类中是 private

|  |
| --- |
| class Base3  {  public:  int m\_A;  protected:  int m\_B;  private:  int m\_C;  };  class Son3 :private Base3  {  public:  void func()  {  //cout << m\_C << endl; //基类中私有的属性 不可继承  cout << m\_A << endl; //基类中公有的属性 可继承，还是private  cout << m\_B << endl;//基类中保护的属性 可继承，还是private  }  };  class GrandSon3 :public Son3  {  public:  void myFunc()  {  //cout << m\_A << endl; //孙子类中 访问不到 m\_A，因为在Son3中m\_A已经是私有属性了  }  }; |

1. 继承中的对象模型
   1. 子类会继承父类中所有的内容 ，包括了 私有属性
   2. 只是我们访问不到，编译器给隐藏了

|  |
| --- |
| class Base  {  public:  int m\_A;  protected:  int m\_B;  private:  int m\_C;  };  //子类中 会继承父类的私有成员，只是被编译给隐藏起来，访问不到私有成员  class Son :public Base  {  public:  int m\_D;  };  void test01()  {  cout << sizeof(Son) << endl;  } |

* 1. cl /d1 reportSingleClassLayout类名 文件名

1. 继承中的构造和析构顺序
   1. 子类创建对象时，先调用父类的构造，然后调用自身构造
   2. 析构顺序与构造顺序相反

|  |
| --- |
| class Base  {  public:  Base()  {  m\_A = 10;  cout << "Base默认构造函数调用" << endl;  }  ~Base()  {  cout << "Base的析构函数调用" << endl;  }  int m\_A;  };  // 子类会继承父类的成员属性，成员函数  //但是 子类 不会继承 父类 构造函数 和 析构函数  //只有父类自己知道如果构造和析构自己的属性，而子类不知道  class Son :public Base  {  public:  Son()  {  cout << "Son默认构造函数调用" << endl;  }  ~Son()  {  cout << "Son的析构函数调用" << endl;  }  }; |

* 1. 子类是不会继承父类的构造函数和析构函数
  2. 补充内容，如果父类中没有合适默认构造，那么子类可以利用初始化列表的方式显示的调用父类的其他构造
  3. 子类中继承需要调父类的默认构造。若没有，通过初始化列表的方式调用父类其他构造

|  |
| --- |
| class Base2  {  public:  Base2(int a)  {  this->m\_A = a;  cout << "有参构造函数调用" << endl;  }  int m\_A;  };  class Son2:public Base2  {  public:  Son2(int a ) : Base2(a)//利用初始化列表方式 显示调用 有参构造  {  }  };  void test02()  {  Son2 s2(1000);  } |

1. 继承中的同名处理
   1. 成员属性 直接调用先调用子类，如果想调用父类 需要作用域
   2. 成员函数 直接调用先调用子类，父类的所有版本都会被隐藏，除非显示用作用域运算符去调用

|  |
| --- |
| class Base  {  public:  Base()  {  m\_A = 100;  }  void fun()  {  cout << "Base func调用" << endl;  }  void fun(int a)  {  cout << "Base func (int a)调用" << endl;  }  int m\_A;  };  class Son :public Base  {  public:  Son()  {  m\_A = 200;  }  void fun()  {  cout << "Son func调用" << endl;  }  int m\_A;  };  void test01()  {  Son s1;  cout << s1.m\_A << endl;  //想调用 父类中 的m\_A  cout << s1.Base::m\_A << endl;  s1.fun();  //调用父类的func  s1.Base::fun(10);  }  //如果子类和父类拥有同名的函数 属性 ，子类会覆盖父类的成员吗？ 不会  //如果子类与父类的成员函数名称相同，子类会把父类的所有的同名版本都隐藏掉  //想调用父类的方法，必须加作用域 |

1. 继承中静态成员的处理
   1. 类似非静态成员函数处理
   2. 如果想访问父类中的成员，加作用域即可
   3. 静态成员属性 子类可以继承下来

|  |
| --- |
| class Base  {  public:  static void func()  {  cout << "base fun()" << endl;  }  static void func(int a)  {  cout << "base fun(int)" << endl;  }  static int m\_A;  };  int Base::m\_A = 10;  class Son :public Base  {  public:  static void func()  {  cout << "son fun()" << endl;  }  static int m\_A;  };  int Son::m\_A = 20;  //静态成员属性 子类可以继承下来  void test01()  {  cout << Son::m\_A << endl;  //访问父类的m\_A  cout << Base::m\_A << endl;  Son::func();  //访问 父类中同名的函数  Son::Base::func(10);  } |

1. 多继承的概念以及问题
   1. class A : public B1, public B2,….
   2. 引发二义性问题
   3. 想解决二义性问题，就需要通过作用域来进行区分

|  |
| --- |
| class Base1  {  public:  Base1()  {  m\_A = 10;  }  public:  int m\_A;  };  class Base2  {  public:  Base2()  {  m\_A = 20;  }  public:  int m\_A;  };  //多继承  class Son :public Base1, public Base2  {  public:  int m\_C;  int m\_D;  };  //多继承中很容易引发二义性  void test01()  {  cout << sizeof(Son) << endl;  Son s1;  //s1.m\_A; //二义性  cout << s1.Base1::m\_A << endl;  cout << s1.Base2::m\_A << endl;  } |

1. 菱形继承问题以及解决
   * 1. 解决问题利用虚基类
     2. sheepTuo内部结构
        1. vbptr 虚基类指针
        2. 指向一张 虚基类表
        3. 通过表找到偏移量
        4. 找到共有的数据

|  |
| --- |
| class Animal  {  public:  int m\_Age;  };  //虚基类 Sheep  class Sheep :virtual public Animal  {  };  //虚基类 Tuo  class Tuo :virtual public Animal  {  };  class SheepTuo :public Sheep, public Tuo  {  };  //菱形继承的解决方案 利用虚继承  //操作的是共享的一份数据  void test01()  {  SheepTuo st;  st.Sheep::m\_Age = 10;  st.Tuo::m\_Age = 20;  cout << st.Sheep::m\_Age << endl;  cout << st.Tuo::m\_Age << endl;  cout << st.m\_Age << endl; //可以直接访问，原因已经没有二义性的可能了，只有一份m\_Age  } |

* 1. 内部工作原理 （可以不掌握）

|  |
| --- |
| //通过地址 找到 偏移量  //内部工作原理  void test02()  {  SheepTuo st;  st.m\_Age = 100;  //找到Sheep的偏移量操作  //cout<< \*(int \*)((int \*)\*(int \*)&st + 1) << endl;  cout << \*(int\*)((int\*)\*(int \*)&st + 1) << endl;  //找到Tuo的偏移量  cout << \*((int \*)((int \*)\*((int \*)&st + 1) + 1)) << endl;    //输出Age  cout << ((Animal\*)((char \*)&st + \*(int\*)((int\*)\*(int \*)&st + 1)))->m\_Age << endl;  } |