IBM需要对华为各级部门做深度咨询分析，

为了提高咨询效率，由任正非直接授权，直接获取各部门的所有权限。

使用前提：

某个类需要实现某种功能，但是这个类自身，因为各种原因，无法自己实现。

需要借助于“外力”才能实现。

## 友元的两种使用形式

友元函数、友元类。

### 友元函数：

**使用全局函数作为友元函数**

需求：

计算机和计算机的升级

Computer.h

|  |
| --- |
| #pragma once  #include <string>  class Computer  {  public:  Computer();  // 使用全局函数作为友元函数  friend void upgrade(Computer\* computer);  std::string description();  private:  std::string cpu; //CPU芯片  }; |

Computer.cpp

|  |
| --- |
| #include "Computer.h"  #include <sstream>  Computer::Computer()  {  cpu = "i7";  }  std::string Computer::description()  {  std::stringstream ret;  ret << "CPU:" << cpu;  return ret.str();  } |

main.cpp

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <Windows.h>  #include "Computer.h"  void upgrade(Computer\* computer) {  computer->cpu = "i9"; //直接访问对象的私有数据成员！！！  }  int main(void) {  Computer shanxing;  std::cout << shanxing.description() << std::endl;    upgrade(&shanxing);  std::cout << shanxing.description() << std::endl;  system("pause");  return 0;  } |

**使用类的成员函数作为友元函数**

Computer.h

|  |
| --- |
| #pragma once  #include <string>  // class ComputerService;  **// 仅仅声明ComputerService不够，需要包含头文件**  **#include "ComputerService.h"**  class Computer  {  public:  Computer();  // 使用全局函数作为友元函数  friend void upgrade(Computer\* computer);  **// 使用类的成员函数，作为友元函数**  **friend void ComputerService::upgrade(Computer\* comptuer);**  std::string description();  private:  std::string cpu; //CPU芯片  }; |

Computer.cpp

不变

|  |
| --- |
| #include "Computer.h"  #include <sstream>  Computer::Computer()  {  cpu = "i7";  }  std::string Computer::description()  {  std::stringstream ret;  ret << "CPU:" << cpu;  return ret.str();  } |

ComputerService.h

|  |
| --- |
| #pragma once  class Computer;  class ComputerService  {  public:  void upgrade(Computer\* computer);  }; |

ComputerService.cpp

|  |
| --- |
| #include "ComputerService.h"  #include "Computer.h"  void ComputerService::upgrade(Computer\* computer) {  computer->cpu = "i9";  } |

main.cpp

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <Windows.h>  #include "Computer.h"  #include "ComputerService.h"  int main(void) {  Computer shanxing;  ComputerService service;  std::cout << shanxing.description() << std::endl;    service.upgrade(&shanxing);  std::cout << shanxing.description() << std::endl;  system("pause");  return 0;  } |

功能上，这两种形式，都是相同，应用场合不同。

一个是，使用普通的全局函数，作为自己的朋友，实现特殊功能。

一个是，使用其他类的成员函数，作为自己的朋友，实现特殊功能。

### 友元类

**为什么要使用友元类**

一个独立的咨询师， 给其他企业做服务时，这个咨询师作为企业的“友元函数”即可。

一个大型的咨询服务公司，比如IBM（IT事务）, 普华永道（会计事务），给其他企业做服务时，使用友元函数就不是很方便了，因为需要设计很多友元函数，不方便。

解决方案：使用“友元类”

**友元类的作用**

如果把A类作为B类的友元类，

那么A类的所有成员函数【在A类的成员函数内】，就可以直接访问【使用】B类的私有成员。

即，友元类可以直接访问对应类的所有成员！！！

实例：

Computer.h

|  |
| --- |
| #pragma once  #include <string>  class ComputerService;  class Computer  {  public:  Computer();  std::string description();  private:  std::string cpu; //CPU芯片  // 友元类  friend class ComputerService;  }; |

Computer.cpp

|  |
| --- |
| #include "Computer.h"  #include <sstream>  Computer::Computer()  {  cpu = "i7";  }  std::string Computer::description()  {  std::stringstream ret;  ret << "CPU:" << cpu;  return ret.str();  } |

ComputerService.h

|  |
| --- |
| #pragma once  class Computer;  class ComputerService  {  public:  void upgrade(Computer\* computer);  void clean(Computer\* computer); //计算机清理  void kill(Computer\* computer); //杀毒  }; |

ComputerService.cpp

|  |
| --- |
| #include "ComputerService.h"  #include "Computer.h"  #include <iostream>  void ComputerService::upgrade(Computer\* computer) {  computer->cpu = "i9";  }  void ComputerService::clean(Computer\* computer)  {  std::cout << "正在对电脑执行清理[CPU:"  << computer->cpu << "]..."  << std::endl;  }  void ComputerService::kill(Computer\* computer)  {  std::cout << "正在对电脑执行杀毒[CPU:"  << computer->cpu << "]..."  << std::endl;  } |

main.cpp

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <Windows.h>  #include "Computer.h"  #include "ComputerService.h"  int main(void) {  Computer shanxing;  ComputerService service;  std::cout << shanxing.description() << std::endl;    service.upgrade(&shanxing);  service.clean(&shanxing);  service.kill(&shanxing);  std::cout << shanxing.description() << std::endl;  system("pause");  return 0;  } |

### 使用注意

友元类，和友元函数，使用friend关键字进行声明即可，与访问权限无关，

所以，可以放在private/pulic/protected任意区域内。

# 项目精讲-万物可运算： 运算符重载

## 为什么要使用运算符重载

C/C++的运算符，支持的数据类型，仅限于基本数据类型。

问题：一头牛+一头马 = ？（牛马神兽？）

一个圆 +一个圆 = ？ （想要变成一个更大的圆）

一头牛 – 一只羊 = ？ (想要变成4只羊，原始的以物易物：1头牛价值5只羊)

解决方案：

使用运算符重载

## 运算符重载的基本用法

### 方式1：使用成员函数重载运算符

需求：  
// 规则：

// 一斤牛肉：2斤猪肉

// 一斤羊肉：3斤猪肉

Cow.h

|  |
| --- |
| #pragma once  class Pork;  class Goat;  class Cow  {  public:  Cow(int weight);  // 参数此时定义为引用类型，更合适，避免拷贝  Pork operator+(const Cow& cow); //同类型进行运算，很频繁  Pork operator+(const Goat& goat); //不同类型进行运算，比较少见  private:  int weight = 0;  }; |

Cow.cpp

|  |
| --- |
| #include "Cow.h"  #include "Pork.h"  #include "Goat.h"  Cow::Cow(int weight)  {  this->weight = weight;  }  // 规则：  // 一斤牛肉：2斤猪肉  // 一斤羊肉：3斤猪肉  Pork Cow::operator+(const Cow &cow)  {  int tmp = (this->weight + cow.weight) \* 2;  return Pork(tmp);  }  Pork Cow::operator+(const Goat& goat)  {  // 不能直接访问goat.weight  //int tmp = this->weight \* 2 + goat.weight \* 3;  int tmp = this->weight \* 2 + goat.getWeight() \* 3;  return Pork(tmp);  } |

Goat.h

|  |
| --- |
| #pragma once  class Goat  {  public:  Goat(int weight);  int getWeight(void) const;  private:  int weight = 0;  }; |

Goat.cpp

|  |
| --- |
| #include "Goat.h"  Goat::Goat(int weight) {  this->weight = weight;  }  int Goat::getWeight(void) const  {  return weight;  } |

Pork.h

|  |
| --- |
| #pragma once  #include <iostream>  class Pork  {  public:  Pork(int weight);  std::string description(void);  private:  int weight = 0;  }; |

Pork.cpp

|  |
| --- |
| #include "Pork.h"  #include <sstream>  Pork::Pork(int weight)  {  this->weight = weight;  }  std::string Pork::description(void)  {  std::stringstream ret;  ret << weight << "斤猪肉";  return ret.str();  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Pork.h"  #include "Cow.h"  #include "Goat.h"  int main(void) {  Cow c1(100);  Cow c2(200);  // 调用c1.operator+(c2);  //相当于：Pork p = c1.operator+(c2);  Pork p = c1 + c2;  std::cout << p.description() << std::endl;  Goat g1(100);  p = c1 + g1;  std::cout << p.description() << std::endl;    system("pause");  return 0;  } |

### 方式2：使用非成员函数【友元函数】重载运算符

Cow.h

|  |
| --- |
| #pragma once  class Pork;  class Goat;  class Cow  {  public:  Cow(int weight);  // 有友元函数实现运算符重载  friend Pork operator+(const Cow& cow1, const Cow& cow2);  friend Pork operator+(const Cow& cow1, const Goat& goat);  private:  int weight = 0;  }; |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Pork.h"  #include "Cow.h"  #include "Goat.h"  Pork operator+(const Cow &cow1, const Cow &cow2)  {  int tmp = (cow1.weight + cow2.weight) \* 2;  return Pork(tmp);  }  Pork operator+(const Cow& cow1, const Goat& goat)  {  int tmp = cow1.weight \* 2 + goat.getWeight() \* 3;  return Pork(tmp);  }  int main(void) {  Cow c1(100);  Cow c2(200);  Goat g1(100);  Pork p = c1 + c2;  std::cout << p.description() << std::endl;  p = c1 + g1; // 思考：如何实现：p = g1 + c1;  std::cout << p.description() << std::endl;  system("pause");  return 0;  } |

其他文件不变。

### 两种方式的区别

区别：

1. 使用成员函数来实现运算符重载时，少写一个参数，因为第一个参数就是this指针。

两种方式的选择：

1. 一般情况下，单目运算符重载，使用成员函数进行重载更方便（不用写参数）
2. 一般情况下，双目运算符重载，使用友元函数更直观

方便实现a+b和b+a相同的效果，成员函数方式无法实现。

例如： 100 + cow; 只能通过友元函数来实现

cow +100; 友元函数和成员函数都可以实现

特殊情况：

1. = （） [ ] -> 不能重载为类的友元函数！！！（否则可能和C++的其他规则矛盾），只能使用成员函数形式进行重载。

（2）如果运算符的第一个操作数要求使用隐式类型转换，则必须为友元函数（成员函数方式的第一个参数是this指针）

注意：

同一个运算符重载， 不能同时使用两种方式来重载，会导致编译器不知道选择哪一个（二义性）

## 运算符重载的禁区和规则

1. 为了防止对标准类型进行运算符重载，  
C++规定重载运算符的操作对象至少有一个不是标准类型，而是用户自定义的类型

比如不能重载 1+2

但是可以重载 cow + 2 和 2 + cow // cow是自定义的对象

2.不能改变原运算符的语法规则， 比如不能把双目运算符重载为单目运算

1. 不能改变原运算符的优先级
2. 不能创建新的运算符，比如 operator\*\*就是非法的, operator\*是可以的
3. 不能对以下这四种运算符，使用友元函数进行重载

= 赋值运算符，（）函数调用运算符，[ ]下标运算符，->通过指针访问类成员

6. 不能对禁止重载的运算符进行重载

**不能被重载的运算符**

|  |  |
| --- | --- |
| 成员访问 | . |
| 域运算 | ：： |
| 内存长度运算 | sizeof |
| 三目运算 | ? : : |
| 预处理 | # |

**可以被重载的运算符**

|  |  |
| --- | --- |
| 双目运算符 | + - \* / % |
| 关系运算符 | == != < <= > >= |
| 逻辑运算符 | && || ! |
| 单目运算符 | +(正号) -（负号） \*（指针） &（取地址） ++ -- |
| 位运算 | & | ~ ^ <<（左移） >>（右移） |
| 赋值运算符 | = += -= \*= /= %= &= |= ^= <<= >>= |
| 内存分配 | new delete new[ ] delete[ ] |
| 其他 | ( ) 函数调用  -> 成员访问 [ ] 下标  ， 逗号 |

# 项目精讲-运算符重载实例： 重载加减运算符+、-

略，参考《运算符重载的基本用法》

其他双目运算符，用法类似。

# 项目精讲-重载赋值运算符=

可以复习项目10的赋值构造函数：  


Boy.h

|  |
| --- |
| #pragma once  #include <string>  class Boy  {  public:  Boy(const char\* name=NULL, int age=0, int salary=0, int darkHorse=0);  ~Boy();  Boy& operator=(const Boy& boy);  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name)+1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  // 注意返回类型 和参数类型  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name)+1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "boy.h"  int main(void) {  Boy boy1("Rock", 38, 58000, 10);  Boy boy2, boy3;    std::cout << boy1.description() << std::endl;  std::cout << boy2.description() << std::endl;  std::cout << boy3.description() << std::endl;  boy3 = boy2 = boy1;  std::cout << boy2.description() << std::endl;  std::cout << boy3.description() << std::endl;  system("pause");  return 0;  } |

注意：

注意赋值运算符重载的返回类型 和参数类型。

返回引用类型，便于连续赋值

参数使用应用类型， 可以省去一次拷贝

参数使用const, 便于保护实参不被破坏。

# 第6节 项目精讲-重载关系运算符>、<、==

Boy.h

|  |
| --- |
| #pragma once  #include <string>  class Boy  {  public:  Boy(const char\* name=NULL, int age=0, int salary=0, int darkHorse=0);  ~Boy();  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name)+1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name)+1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "boy.h"  int main(void) {  Boy boy1("Rock", 38, 58000, 5);  Boy boy2("Jack", 25, 50000, 10);    if (boy1 > boy2) {  std::cout << "选择boy1" << std::endl;  }  else if (boy1 == boy2) {  std::cout << "难以选择" << std::endl;  }  else {  std::cout << "选择boy2" << std::endl;  }  system("pause");  return 0;  } |

# 项目精讲-字符串也能做下标： 重载运算符[ ]

Boy.h

|  |
| --- |
| #pragma once  #include <string>  class Boy  {  public:  Boy(const char\* name=NULL, int age=0, int salary=0, int darkHorse=0);  ~Boy();  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  int operator[](std::string index);  int operator[](int index);    std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name)+1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name)+1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index)  {  if (index == "age") {  return age;  }  else if (index == "salary") {  return salary;  }  else if (index == "darkHorse") {  return darkHorse;  }  else if (index == "power") {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "boy.h"  int main(void) {  Boy boy1("Rock", 38, 58000, 5);  Boy boy2("Jack", 25, 50000, 10);    std::cout << "age:" << boy1["age"] << std::endl;  std::cout << "salary:" << boy1["salary"] << std::endl;  std::cout << "darkHorse:" << boy1["darkHorse"] << std::endl;  std::cout << "power:" << boy1["power"] << std::endl;  std::cout << "[0]:" << boy1[0] << std::endl;  std::cout << "[1]:" << boy1[1] << std::endl;  std::cout << "[2]:" << boy1[2] << std::endl;  std::cout << "[3]:" << boy1[3] << std::endl;  system("pause");  return 0;  } |

# 项目精讲-自定义输入输出：重载<<和>>

## 为什么要重载<< 和 >>

为了更方便的实现复杂对象的输入和输出。

## 实例

### 方式1（使用成员函数， 不推荐，该方式没有实际意义）

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  using namespace std;  class Boy  {  public:  Boy(const char\* name = NULL, int age = 0, int salary = 0, int darkHorse = 0);  ~Boy();  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  int operator[](std::string index);  int operator[](int index);  ostream& operator<<(ostream& os) const;  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name) + 1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index)  {  if (index == "age") {  return age;  }  else if (index == "salary") {  return salary;  }  else if (index == "darkHorse") {  return darkHorse;  }  else if (index == "power") {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  ostream& Boy::operator<<(ostream& os) const  {  os << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return os;  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "boy.h"  int main(void) {  Boy boy1("Rock", 38, 58000, 5);  Boy boy2("Jack", 25, 50000, 10);    // 调用： boy1.operator<<(cout);  boy1 << cout;  // 先调用 boy1.operator<<(cout)  // 再调用 boy2.operator<<(cout)  boy2 << (boy1 << cout);  system("pause");  return 0;  } |

这种方式使用起来，很不方便。

### 方式2：使用友元函数

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  #define AGE\_KEY "age"  #define SALARY\_KEY "salary"  #define DARK\_HORSE\_KEY "darkHorse"  #define POWER\_KEY "power"  typedef enum {  AGE,  SALARY,  DARK\_HORSE,  POWER  }BOY\_KEY\_TYPE;  using namespace std;  class Boy  {  public:  Boy(const char\* name = NULL, int age = 0, int salary = 0, int darkHorse = 0);  ~Boy();  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  // 下标运算符的重载  int operator[](std::string index);  int operator[](int index);  // 该方式不适合  //ostream& operator<<(ostream& os) const;  **friend ostream& operator<<(ostream& os, const Boy& boy);**  **friend istream& operator>>(istream& is, Boy& boy);**  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp 【不变】

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name) + 1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index)  {  if (index == AGE\_KEY) {  return age;  }  else if (index == SALARY\_KEY) {  return salary;  }  else if (index == DARK\_HORSE\_KEY) {  return darkHorse;  }  else if (index == POWER\_KEY) {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  //ostream& Boy::operator<<(ostream& os) const  //{  // os << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  // << salary << "\t黑马系数:" << darkHorse;  // return os;  //}  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Boy.h"  using namespace std;  **ostream& operator<<(ostream& os, const Boy& boy) {**  **os << "ID:" << boy.id << "\t姓名:" << boy.name << "\t年龄:" << boy.age << "\t薪资:"**  **<< boy.salary << "\t黑马系数:" << boy.darkHorse;**  **return os;**  **}**  **istream& operator>>(istream& is, Boy& boy)**  **{**  **string name2;**  **is >> name2 >> boy.age >> boy.salary >> boy.darkHorse;**  **boy.name = (char\*)malloc((name2.length()+1) \* sizeof(char));**  **strcpy\_s(boy.name, name2.length() + 1, name2.c\_str());**  **return is;**  **}**  int main(void) {  Boy boy1("Rock", 38, 58000, 5);  Boy boy2("Jack", 25, 50000, 10);  cout << boy1 << endl;  cin >> boy1;  cout << boy1;    system("pause");  return 0;  } |

# 第9节 项目精讲-运算符重载实例： 重载类型运算符

## 普通类型 => 类类型

调用对应的只有一个参数【参数的类型就是这个普通类型】的构造函数

需求：  
 Boy boy1 = 10000; // 薪资 构造函数Boy(int);

Boy boy2 = "Rock" // 姓名 构造函数Boy(char \*);

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  #define AGE\_KEY "age"  #define SALARY\_KEY "salary"  #define DARK\_HORSE\_KEY "darkHorse"  #define POWER\_KEY "power"  typedef enum {  AGE,  SALARY,  DARK\_HORSE,  POWER  }BOY\_KEY\_TYPE;  using namespace std;  class Boy  {  public:  //Boy(const char\* name = NULL, int age = 0, int salary = 0, int darkHorse = 0);  Boy(const char\* name, int age, int salary, int darkHorse);  ~Boy();  Boy(int salary);  Boy(const char\* name);  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  // 下标运算符的重载  int operator[](std::string index);  int operator[](int index);  // 该方式不适合  //ostream& operator<<(ostream& os) const;  friend ostream& operator<<(ostream& os, const Boy& boy);  friend istream& operator>>(istream& is, Boy& boy);  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy::Boy(int salary)  {  const char \*defaultName = "未命名";  name = new char[strlen(defaultName) + 1];  strcpy\_s(name, strlen(defaultName) + 1, defaultName);  age = 0;  this->salary = salary;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy::Boy(const char\* name) {  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  age = 0;  this->salary = 0;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name) + 1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index)  {  if (index == AGE\_KEY) {  return age;  }  else if (index == SALARY\_KEY) {  return salary;  }  else if (index == DARK\_HORSE\_KEY) {  return darkHorse;  }  else if (index == POWER\_KEY) {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Boy.h"  using namespace std;  ostream& operator<<(ostream& os, const Boy& boy) {  os << "ID:" << boy.id << "\t姓名:" << boy.name << "\t年龄:" << boy.age << "\t薪资:"  << boy.salary << "\t黑马系数:" << boy.darkHorse;  return os;  }  istream& operator>>(istream& is, Boy& boy)  {  string name2;  is >> name2 >> boy.age >> boy.salary >> boy.darkHorse;  boy.name = (char\*)malloc((name2.length() + 1) \* sizeof(char));  strcpy\_s(boy.name, name2.length() + 1, name2.c\_str());  return is;  }  int main()  {  Boy boy1 = 10000;  Boy boy2 = "Rock";  cout << boy1 << endl;  cout << boy2 << endl;  boy1 = 20000; //boy1 = Boy(20000);  cout << boy1 << endl;  return 0;  } |

## 类类型 => 普通类型

调用特殊的运算符重载函数，类型转换函数，不需要写返回类型

类型转换函数：operator 普通类型 ( )

需求：

Boy boy1(“Rock”, 28, 10000, 5);

int power = boy1; // power();

char \*name = boy1; // “Rock”

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  #define AGE\_KEY "age"  #define SALARY\_KEY "salary"  #define DARK\_HORSE\_KEY "darkHorse"  #define POWER\_KEY "power"  typedef enum {  AGE,  SALARY,  DARK\_HORSE,  POWER  }BOY\_KEY\_TYPE;  using namespace std;  class Boy  {  public:  //Boy(const char\* name = NULL, int age = 0, int salary = 0, int darkHorse = 0);  Boy(const char\* name, int age, int salary, int darkHorse);  ~Boy();  Boy(int salary);  Boy(const char\* name);  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  // 下标运算符的重载  int operator[](std::string index);  int operator[](int index);  // 该方式不适合  //ostream& operator<<(ostream& os) const;  friend ostream& operator<<(ostream& os, const Boy& boy);  friend istream& operator>>(istream& is, Boy& boy);  // 特殊的运算符重载：类型转换函数，不需要写返回类型  operator int() const;  operator char\* () const;  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  }; |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy::Boy(int salary)  {  const char \*defaultName = "未命名";  name = new char[strlen(defaultName) + 1];  strcpy\_s(name, strlen(defaultName) + 1, defaultName);  age = 0;  this->salary = salary;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy::Boy(const char\* name) {  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  age = 0;  this->salary = 0;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name) + 1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index)  {  if (index == AGE\_KEY) {  return age;  }  else if (index == SALARY\_KEY) {  return salary;  }  else if (index == DARK\_HORSE\_KEY) {  return darkHorse;  }  else if (index == POWER\_KEY) {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  Boy::operator int() const  {  return power();  }  Boy::operator char\* () const  {  return name;  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Boy.h"  using namespace std;  ostream& operator<<(ostream& os, const Boy& boy) {  os << "ID:" << boy.id << "\t姓名:" << boy.name << "\t年龄:" << boy.age << "\t薪资:"  << boy.salary << "\t黑马系数:" << boy.darkHorse;  return os;  }  istream& operator>>(istream& is, Boy& boy)  {  string name2;  is >> name2 >> boy.age >> boy.salary >> boy.darkHorse;  boy.name = (char\*)malloc((name2.length() + 1) \* sizeof(char));  strcpy\_s(boy.name, name2.length() + 1, name2.c\_str());  return is;  }  int main()  {  Boy boy1("Rock", 28, 10000, 5);  Boy boy2("Rock");  int power = boy1;  char\* name = boy2;  cout << power << endl;  cout << name << endl;  system("pause");  return 0;  } |

## 类类型A => 类类型B

调用对应的只有一个参数【参数的类型就是类类型A】的构造函数

也可以使用类型转换函数，但是使用对应的构造函数更合适。

实例：

把Boy类型，转换为Man类型

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  #define AGE\_KEY "age"  #define SALARY\_KEY "salary"  #define DARK\_HORSE\_KEY "darkHorse"  #define POWER\_KEY "power"  typedef enum {  AGE,  SALARY,  DARK\_HORSE,  POWER  }BOY\_KEY\_TYPE;  using namespace std;  class Boy  {  public:  //Boy(const char\* name = NULL, int age = 0, int salary = 0, int darkHorse = 0);  Boy(const char\* name, int age, int salary, int darkHorse);  ~Boy();  Boy(int salary);  Boy(const char\* name);  Boy& operator=(const Boy& boy);  bool operator>(const Boy& boy);  bool operator<(const Boy& boy);  bool operator==(const Boy& boy);  // 下标运算符的重载  int operator[](std::string index) const;  int operator[](int index) const;  // 该方式不适合  //ostream& operator<<(ostream& os) const;  friend ostream& operator<<(ostream& os, const Boy& boy);  friend istream& operator>>(istream& is, Boy& boy);  // 特殊的运算符重载：类型转换函数，不需要写返回类型  operator int() const;  operator char\* () const;  std::string description(void);  private:  char\* name;  int age;  int salary;  int darkHorse; //黑马值，潜力系数  unsigned int id; // 编号  static int LAST\_ID;  int power() const; //综合能力值  };  ostream& operator<<(ostream& os, const Boy& boy);  istream& operator>>(istream& is, Boy& boy); |

Boy.cpp

|  |
| --- |
| #include "boy.h"  #include <string.h>  #include <sstream>  int Boy::LAST\_ID = 0; //初始值是0  Boy::Boy(const char\* name, int age, int salary, int darkHorse)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  this->darkHorse = darkHorse;  this->id = ++LAST\_ID;  }  Boy::~Boy()  {  if (name) {  delete name;  }  }  Boy::Boy(int salary)  {  const char \*defaultName = "未命名";  name = new char[strlen(defaultName) + 1];  strcpy\_s(name, strlen(defaultName) + 1, defaultName);  age = 0;  this->salary = salary;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy::Boy(const char\* name) {  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  age = 0;  this->salary = 0;  darkHorse = 0;  this->id = ++LAST\_ID;  }  Boy& Boy::operator=(const Boy& boy)  {  if (name) {  delete name; //释放原来的内存  }  name = new char[strlen(boy.name) + 1]; //分配新的内存  strcpy\_s(name, strlen(boy.name) + 1, boy.name);  this->age = boy.age;  this->salary = boy.salary;  this->darkHorse = boy.darkHorse;  //this->id = boy.id; //根据需求来确定是否要拷贝id  return \*this;  }  bool Boy::operator>(const Boy& boy)  {  // 设置比较规则：  // 薪资 \* 黑马系数 + (100-年龄）\*100  if (power() > boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator<(const Boy& boy)  {  if (power() < boy.power()) {  return true;  }  else {  return false;  }  }  bool Boy::operator==(const Boy& boy)  {  if (power() == boy.power()) {  return true;  }  else {  return false;  }  }  int Boy::operator[](std::string index) const  {  if (index == AGE\_KEY) {  return age;  }  else if (index == SALARY\_KEY) {  return salary;  }  else if (index == DARK\_HORSE\_KEY) {  return darkHorse;  }  else if (index == POWER\_KEY) {  return power();  }  else {  return -1;  }  }  int Boy::operator[](int index)const  {  if (index == 0) {  return age;  }  else if (index == 1) {  return salary;  }  else if (index == 2) {  return darkHorse;  }  else if (index == 3) {  return power();  }  else {  return -1;  }  }  Boy::operator char\* () const  {  return name;  }  std::string Boy::description(void)  {  std::stringstream ret;  ret << "ID:" << id << "\t姓名:" << name << "\t年龄:" << age << "\t薪资:"  << salary << "\t黑马系数:" << darkHorse;  return ret.str();  }  int Boy::power() const  {  // 薪资\* 黑马系数 + (100 - 年龄） \* 1000  int value = salary \* darkHorse + (100 - age) \* 100;  return value;  }  ostream& operator<<(ostream& os, const Boy& boy) {  os << "ID:" << boy.id << "\t姓名:" << boy.name << "\t年龄:" << boy.age << "\t薪资:"  << boy.salary << "\t黑马系数:" << boy.darkHorse;  return os;  }  istream& operator>>(istream& is, Boy& boy)  {  string name2;  is >> name2 >> boy.age >> boy.salary >> boy.darkHorse;  boy.name = (char\*)malloc((name2.length() + 1) \* sizeof(char));  strcpy\_s(boy.name, name2.length() + 1, name2.c\_str());  return is;  } |

Man.h

|  |
| --- |
| #pragma once  #include <iostream>  using namespace std;  class Boy;  class Man  {  public:  Man(const char \*name, int age, int salary);  Man(const Boy& boy);  ~Man();  friend ostream& operator<<(ostream &os, const Man& man);  private:  char\* name;  int age;  int salary;  };  ostream& operator<<(ostream &os, const Man& man); |

Man.cpp

|  |
| --- |
| #include "Man.h"  #include "Boy.h"  #include <string.h>  Man::Man(const char\* name, int age, int salary)  {  if (!name) {  name = "未命名";  }  this->name = new char[strlen(name) + 1];  strcpy\_s(this->name, strlen(name) + 1, name);  this->age = age;  this->salary = salary;  }  Man::Man(const Boy& boy)  {  int len = strlen((char\*)boy) + 1;  name = new char[len];  strcpy\_s(name, len, (char\*)boy);  age = boy[AGE];  salary = boy[SALARY];  }  Man::~Man() {  delete name;  }  ostream& operator<<(ostream &os, const Man& man) {  os << "【男人】姓名:" << man.name  << "\t年龄 : " << man.age  << "\t薪资 : " << man.salary;  return os;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "Boy.h"  #include "Man.h"  using namespace std;  int main()  {  Boy boy("Rock", 28, 10000, 5);  Man man = boy;    cout << boy << endl;  cout << man << endl;  system("pause");  return 0;  } |

# 第10节 英语不是障碍：计算机英语加油站

|  |  |
| --- | --- |
| operator | 操作，运算 |
| friend | 朋友 |

# 第11节 项目实现

Single.h [未修改]

|  |
| --- |
| #pragma once  #include <string>  using namespace std;  class Single  {  public:  Single();  Single(string name, int age);  ~Single();  int getAge()const;  string getName()const;  protected:  int age;  string name;  }; |

Single.cpp[未修改]

|  |
| --- |
| #include "Single.h"  Single::Single()  {  this->name = "无名";  this->age = 0;  }  Single::Single(string name, int age)  {  this->name = name;  this->age = age;  }  Single::~Single()  {  }  int Single::getAge() const {  return age;  }  string Single::getName() const {  return name;  } |

Boy.h

|  |
| --- |
| #pragma once  #include <string>  #include <vector>  #include <iostream>  #include "Single.h"  using namespace std;  class Girl;  class Boy : public Single  {  public:  Boy();  Boy(int age, string name, int salary);  ~Boy();  Boy(const Boy& other);  //int getAge()const;  //string getName()const;  int getSalary()const;  bool satisfied(const Girl& s) const;  string description() const;  bool operator>(const Boy& boy);  friend ostream& operator<<(ostream& os, const Boy& boy);  static void inputBoys(vector<Boy>& boys);  private:  //int age;  //string name;  int salary;  };  ostream& operator<<(ostream& os, const Boy& boy); |

Boy.cpp

|  |
| --- |
| #include "Boy.h"  #include "Girl.h"  #include <iostream>  #include <sstream>  #define SALARY\_FACTOR 0.006  Boy::Boy() {  }  Boy::Boy(const Boy& other) :Single(other.getName(), other.getAge()) {  salary = other.salary;  //name = other.name;  //age = other.age;  }  Boy::Boy(int age, string name, int salary) :Single(name, age) {  //this->age = age;  //this->name = name;  this->salary = salary;  }  Boy::~Boy() {  }  /\*  int Boy::getAge() const {  return age;  }  string Boy::getName() const {  return name;  }  \*/  int Boy::getSalary() const {  return salary;  }  bool Boy::satisfied(const Girl& s) const {  if (s.getYanZhi() >= salary \* SALARY\_FACTOR) {  return true;  }  else {  return false;  }  }  string Boy::description()const {  // 规范一下字符串格式：  // 性别:男\t\t\t姓名:Rock\t\t\t薪资:35000\t\t\t年龄:39  stringstream ret;  //ret << name << "-男-薪资(" << salary << ")-年龄(" << age << ")";  //ret << getName() << "-男-薪资(" << salary << ")-年龄(" << getAge() << ")";  ret << "性别:男"  << "\t\t\t姓名:" << name  << "\t\t\t薪资:" << salary  << "\t\t\t年龄:" << age;  return ret.str();  }  bool Boy::operator>(const Boy& boy)  {  return salary > boy.salary;  }  void Boy::inputBoys(vector<Boy>& boys) {  int age;  string name;  int salary;  int n = 1;  while (1) {  cout << "请输入第" << n << "位小哥哥的年龄【输入0结束】：";  cin >> age;  if (age == 0) {  break;  }  cout << "请输入第" << n << "位小哥哥的姓名：";  cin >> name;  cout << "请输入第" << n << "位小哥哥的薪资：";  cin >> salary;  n++;  boys.push\_back(Boy(age, name, salary));  }  }  ostream& operator<<(ostream& os, const Boy& boy) {  os << "性别:男"  << "\t\t\t姓名:" << boy.name  << "\t\t\t薪资:" << boy.salary  << "\t\t\t年龄:" << boy.age;  return os;  } |

Girl.h

|  |
| --- |
| #pragma once  #include <string>  #include <vector>  #include <iostream>  #include "Single.h"  using namespace std;  class Boy;  class Girl : public Single  {  public:  Girl();  Girl(int age, string name, int yanZhi);  ~Girl();  //int getAge()const;  //string getName()const;  int getYanZhi()const;  bool satisfied(const Boy& s) const;  string description()const;  bool operator>(const Girl& girl);  friend ostream& operator<<(ostream& os, const Girl& girl);  static void inputGirls(vector<Girl>& girls);  private:  //int age;  //string name;  int yanZhi;  };  ostream& operator<<(ostream& os, const Girl& girl); |

Girl.cpp

|  |
| --- |
| #include "Girl.h"  #include "Boy.h"  #include <sstream>  #include <iostream>  #include <iomanip>  #define YANZHI\_FACTOR 100  Girl::Girl() {  }  Girl::Girl(int age, string name, int yanZhi) :Single(name, age) {  //this->age = age;  //this->name = name;  this->yanZhi = yanZhi;  }  Girl::~Girl() {  }  /\*  int Girl::getAge() const {  return age;  }  string Girl::getName() const {  return name;  }  \*/  int Girl::getYanZhi() const {  return yanZhi;  }  bool Girl::satisfied(const Boy& s) const {  if (s.getSalary() >= yanZhi \* YANZHI\_FACTOR) {  return true;  }  else {  return false;  }  }  string Girl::description() const {  // 规范一下字符串格式：  // 性别:女\t\t\t姓名:Rock\t\t\t颜值:99\t\t\t年龄:39  stringstream ret;  //ret << name << "-女-颜值(" << yanZhi << ")-年龄(" << age << ")";  //ret << getName() << "-女-颜值(" << yanZhi << ")-年龄(" << getAge() << ")"; return ret.str();  ret << "性别:女"  << "\t\t\t姓名:" << name  << "\t\t\t颜值:" << setw(3) << setiosflags(ios::left) << yanZhi  << "\t\t\t年龄:" << age;  return ret.str();  }  bool Girl::operator>(const Girl& girl)  {  return yanZhi > girl.yanZhi;  }  void Girl::inputGirls(vector<Girl>& girls) {  int age;  string name;  int yanZhi;  int n = 1;  while (1) {  cout << "请输入第" << n << "位小姐姐的年龄【输入0结束】：";  cin >> age;  if (age == 0) {  break;  }  cout << "请输入第" << n << "位小姐姐的姓名：";  cin >> name;  cout << "请输入第" << n << "位小姐姐的颜值：";  cin >> yanZhi;  n++;  girls.push\_back(Girl(age, name, yanZhi));  }  }  ostream& operator<<(ostream& os, const Girl& girl) {  os << "性别:女"  << "\t\t\t姓名:" << girl.name  << "\t\t\t颜值:" << setw(3) << setiosflags(ios::left) << girl.yanZhi  << "\t\t\t年龄:" << girl.age;  return os;  } |

Database.h

|  |
| --- |
| #pragma once  #include <vector>  #include "Boy.h"  #include "Girl.h"  using namespace std;  /\*  功能：  init() //初始化， 从文件中读取数据信息， 来初始化用户数据  autoPair() //自动配对  print() // 打印该数据库中的所有用户信息  数据：  vector<Boy> boys; //所有的单身男信息  vector<Girl> girls; //所有单生女信息  \*/  class Database  {  public:  Database();  //初始化， 从文件中读取数据信息， 来初始化用户数据  void init();  //自动配对  void autoPair();  //自动配对最佳伴侣  void autoPairBest();  // 打印该数据库中的所有用户信息  void print();  private:  //所有的单身男信息  vector<Boy> boys;  //所有单生女信息  vector<Girl> girls;  // 用文件中的信息来初始化boys  void initBoysFromFile();  // 用文件中的信息来初始化girls  void initGirlsFromFile();  // 保存boys的信息到文件中  void saveBoys();  // 保存girls的信息到文件中  void saveGirls();  }; |

Database.cpp

|  |
| --- |
| #include <fstream>  #include <iostream>  #include <string>  #include "Database.h"  #define BOYS\_FILE "boys.txt"  #define GIRLS\_FILE "girls.txt"  Database::Database()  {  }  void Database::init()  {  // 从之前已经保存的文件中， 读取用户信息，  // 用来初始化内部数据boys和girls  initBoysFromFile();  initGirlsFromFile();  }  void Database::autoPair()  {  cout << endl << "自动配对结果：" << endl;  string line(100, '-');  cout << line << endl;  for (int i = 0; i < boys.size(); i++) {  for (int j = 0; j < girls.size(); j++) {  if (boys[i].satisfied(girls[j]) &&  girls[j].satisfied(boys[i])) {  cout << boys[i].description() << endl;  cout << girls[j].description() << endl;  cout << line << endl;  }  }  }  for (int i = 0; i < girls.size(); i++) {  for (int j = 0; j < boys.size(); j++) {  if (boys[j].satisfied(girls[i]) &&  girls[i].satisfied(boys[j])) {  cout << girls[i].description() << endl;  cout << boys[j].description() << endl;  cout << line << endl;  }  }  }  }  void Database::autoPairBest()  {  cout << endl << "自动配对最佳伴侣：" << endl;  string line(100, '-');  cout << line << endl;  for (int i = 0; i < boys.size(); i++) {  Girl\* girlBest = NULL;  for (int j = 0; j < girls.size(); j++) {  if (boys[i].satisfied(girls[j]) &&  girls[j].satisfied(boys[i])) {  if (!girlBest) {  girlBest = &girls[j];  }  else if (girls[j] > \*girlBest) {  girlBest = &girls[j];  }  }  }  if (girlBest) {  cout << boys[i] << endl;  cout << \*girlBest << endl;  cout << line << endl;  }  }  for (int i = 0; i < girls.size(); i++) {  Boy\* boyBest = NULL;  for (int j = 0; j < boys.size(); j++) {  if (boys[j].satisfied(girls[i]) &&  girls[i].satisfied(boys[j])) {  if (!boyBest) {  boyBest = &boys[j];  }  else if (boys[j] > \*boyBest) {  boyBest = &boys[j];  }  }  }  if (boyBest) {  cout << girls[i] << endl;  cout << \*boyBest << endl;  cout << line << endl;  }  }  }  void Database::print()  {  cout << "男嘉宾信息：\n";  for (int i = 0; i < boys.size(); i++) {  cout << boys[i].description() << endl;  }  cout << "\n女嘉宾信息：\n";  for (int i = 0; i < girls.size(); i++) {  cout << girls[i].description() << endl;  }  }  void Database::initBoysFromFile()  {  // 打开文件 读文件内容 初始化boys  ifstream stream;  stream.open(BOYS\_FILE);  if (!stream.is\_open()) {  // 最开始时， 没有文件!  // 假设有这个需求：  // 最开始没有文件时， 让用户输入基础用户数据  cout << "===输入基础用户【男嘉宾】数据===" << endl;  Boy::inputBoys(this->boys);  saveBoys();  stream.close();  return;  }  /\*  V  xxxxx xxx xxxx xxxxx EOF  \*/  // boys.txt打开成功  while (1) {  string line;  char name[64] = "";  int salary;  int age;  getline(stream, line);  if (stream.eof()) {  break;  }  // 解析读到的一行数据line  // to do.  // 文件的格式： 性别:男\t\t\t姓名:Rock\t\t\t薪资:35000\t\t\t年龄:39  int ret = sscanf\_s(line.c\_str(), "性别:男 姓名:%s 薪资:%d 年龄:%d",  name, sizeof(name), &salary, &age);  if (ret <= 0) {  cout << "男嘉宾数据库格式匹配失败" << endl;  exit(1);  }  //使用读到的一个单身男信息，构建一个Boy对象， 并添加到boys  boys.push\_back(Boy(age, string(name), salary));  }  }  void Database::initGirlsFromFile()  {  ifstream stream;  stream.open(GIRLS\_FILE);  if (!stream.is\_open()) {  // 最开始时， 没有文件!  // 假设有这个需求：  // 最开始没有文件时， 让用户输入基础用户数据  cout << "===输入基础用户【女嘉宾】数据===" << endl;  Girl::inputGirls(this->girls);  saveGirls();  stream.close();  return;  }  // boys.txt打开成功  while (1) {  string line;  char name[64] = "";  int yanzhi;  int age;  getline(stream, line);  if (stream.eof()) {  break;  }  // 解析读到的一行数据line  // to do.  // 性别:女\t\t\t姓名:Rock\t\t\t颜值:99\t\t\t年龄:39  int ret = sscanf\_s(line.c\_str(), "性别:女 姓名:%s 颜值:%d 年龄:%d",  name, sizeof(name), &yanzhi, &age);  if (ret <= 0) {  cout << "女嘉宾数据库格式匹配失败" << endl;  exit(1);  }  //使用读到的一个单身男信息，构建一个Boy对象， 并添加到boys  girls.push\_back(Girl(age, string(name), yanzhi));  }  }  void Database::saveBoys()  {  // 把boys中的所有单身男信息写入文件  ofstream stream;  stream.open(BOYS\_FILE);  if (!stream.is\_open()) {  cout << BOYS\_FILE << "写入失败" << endl;  exit(1);  }  for (int i = 0; i < boys.size(); i++) {  stream << boys[i].description() << endl;  }  stream.close();  }  void Database::saveGirls()  {  ofstream stream;  stream.open(GIRLS\_FILE);  if (!stream.is\_open()) {  cout << GIRLS\_FILE << "写入失败" << endl;  exit(1);  }  for (int i = 0; i < girls.size(); i++) {  stream << girls[i].description() << endl;  }  stream.close();  } |

main.cpp

|  |
| --- |
| #include <stdio.h>  #include <iostream>  #include <Windows.h>  #include <string>  #include <vector>  #include <vector>  #include "Boy.h"  #include "Girl.h"  #include "Database.h"  /\*  需求： 要使用文件来保存用户信息  分析： 设计一个类， 来实现信息的保存功能  Database 数据库  功能：  init() //初始化， 从文件中读取数据信息， 来初始化用户数据  autoPair() //自动配对  print() // 打印该数据库中的所有用户信息  数据：  vector<Boy> boys; //所有的单身男信息  vector<Girl> girls; //所有单生女信息  \*/  int main(void) {  Database data;  data.init();  data.print();  data.autoPair();  data.autoPairBest();  system("pause");  return 0;  }  /\*  输入用例：  25 杨过 15000  28 郭靖 8000  35 段誉 50000  26 陆展元 15500  0  18 小龙女 95  25 如花 79  26 秋香 90  26 李莫愁 100  0  \*/ |

# 第12节 常见错误总结

## const导致的异常BUG

小结：

const对象，只能调用对应的const方法

所以：

类的成员函数，如果已经确定不会修改任何数据成员，

那么，最好把这个成员函数，定义为const函数（在函数体的前面，参数列表的后面添加const）

错误代码，详见：《常见错误总结-错误1》

## operator=的参数问题

赋值运算符的重载，应该使用这种方式：

Boy& operator=(const Boy &boy);

就是：参数要使用引用！

如果定义成：

Boy& operator=(const Boy \*boy);

将会没有效果，编译器不会识别为赋值运算符的重载，

也就是：boy2 = boy1时不会调用这个函数

如果定义：

Boy& operator=(const Boy boy);

有效果，但是在调用时，会执行参数的传递：

比如：boy2 = boy1;

就会执行： boy2.operator=(boy1);

就会执行: const Boy boy = boy1;

就会执行： Boy类的赋值构造函数

有两个影响：

1. 浪费性能
2. 如果没有自定义的拷贝构造函数，而且这个类又有指针成员时，就会调用自动生成的拷贝构造函数，导致浅拷贝

如果析构函数中，对这个指针指向的内存做了释放，那就导致数据损坏或崩溃！

小结：

1）赋值运算符的重载，一定要使用引用参数

2）如果一个类有指针成员，而且使用了动态内存分配，那么一定要定义自己的拷贝构造函数【要使用深拷贝】，避免调用自动生成的拷贝构造函数

因为自动生成的拷贝构造函数，是浅拷贝！

SpriteStone.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  //C++ 11 引入  enum class SpriteStoneLevel {  PRIMARY\_LEVEL,  MIDDLE\_LEVEL,  ADVANCE\_LEVEL,  SPRRITE\_STONE\_LEVEL\_COUNT  };  class SpriteStone  {  public:  SpriteStone(  int count = 0,  SpriteStoneLevel level = SpriteStoneLevel::PRIMARY\_LEVEL);  std::string str() const;  friend std::ostream& operator<<(std::ostream& os, const SpriteStone& stone);  private:  int count; //数量：块  SpriteStoneLevel level;  };  std::ostream& operator<<(std::ostream& os, const SpriteStone& stone); |

SpriteStone.cpp

|  |
| --- |
| #include "SpriteStone.h"  #include <sstream>  SpriteStone::SpriteStone(int count, SpriteStoneLevel level)  {  this->count = count;  this->level = level;  }  std::string SpriteStone::str() const  {  std::stringstream ret;  ret << count << "块";  switch (level) {  case SpriteStoneLevel::PRIMARY\_LEVEL:  ret << "初阶灵石";  break;  case SpriteStoneLevel::MIDDLE\_LEVEL:  ret << "中阶灵石";  break;  case SpriteStoneLevel::ADVANCE\_LEVEL:  ret << "高阶灵石";  break;  default:  ret << "未知灵石";  break;  }  return ret.str();  }  std::ostream& operator<<(std::ostream& os, const SpriteStone& stone)  {  os << stone.str();  return os;  } |

测试：

main.cpp

|  |
| --- |
| #include <iostream>  #include "SpriteStone.h"  void testSpriteStone() {  SpriteStone stone(100, SpriteStoneLevel::MIDDLE\_LEVEL);  std::cout << stone << std::endl;  }  int main(void) {  testSpriteStone();  return 0;  } |

### 实现“妖兽”类

Monster.h

|  |
| --- |
| #pragma once  #include <string>  #include <iostream>  class SpriteStone;  class Monster  {  public:  Monster(int level, const std::string &category);  SpriteStone getValue();  friend std::ostream& operator<<(std::ostream& os, const Monster& monster);  private:  std::string category; //种类  int level; //1-9级  };  std::ostream& operator<<(std::ostream& os, const Monster& monster); |

Monster.cpp

|  |
| --- |
| #include "Monster.h"  #include "SpriteStone.h"  Monster::Monster(int level, const std::string &category)  {  this->level = level;  this->category = category;  }  SpriteStone Monster::getValue() {  int stoneCount[] = { 100, 200, 500, 1000,2000,5000,10000,20000,100000};  int count = stoneCount[level - 1];  return SpriteStone(count, SpriteStoneLevel::PRIMARY\_LEVEL);  }  std::ostream& operator<<(std::ostream& os, const Monster& monster)  {  os << monster.level << "级" << monster.category << "妖兽";  return os;  } |

main.cpp

|  |
| --- |
| #include <iostream>  #include "SpriteStone.h"  #include "Monster.h"  void testSpriteStone() {  SpriteStone stone(100, SpriteStoneLevel::MIDDLE\_LEVEL);  std::cout << stone << std::endl;  }  void testMonster() {  Monster monster(5, "蛟龙");  std::cout << monster << std::endl;  }  int main(void) {  testSpriteStone();  testMonster();  return 0;  } |

### 实现“修仙者”类

为了便于实现贸易活动，

在灵石类中重载加法+运算

在灵石类中重载减法-运算

在灵石类中重载比较运算>=

在灵石类中添加获取块数的方法

在妖兽类中重载==运算

在妖兽类中添加获取战斗力的方法

完整参考代码，参见群文件:

**项目13-练习1-参考代码.rar**

## 练习2:【选修】

对练习1进行补充：

1）修仙者之间可以进行比武，实力更强大的胜利。

2）修仙者之间还可能进行争斗，实力更强大的生，另一方亡，生者夺取亡者的所有资产。

3）修仙者闭关进阶，可能成功，可能失败，进阶成功后，自己的级别加1