



**C++上机报告**

**实验题目 　 字符串类设计与使用**

**学 院 　计算机科学与工程学院**

**专 业 　 计算机科学与技术**

**班 级 　 计算机2101班**

**姓 名 　　　　　张清晨**

**学 号 2104230414**

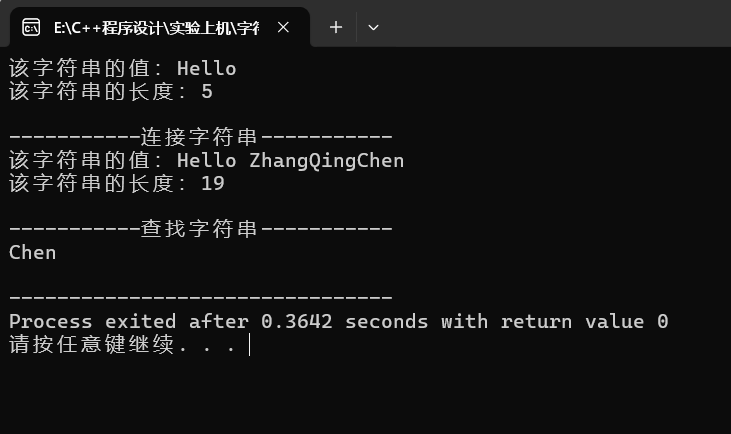
**成 绩**

**指导教师 　 宋晓宇**

**（一） 代码实现：**

|  |
| --- |
| #include <iostream>  #include <cstring>  #include <cassert>  using namespace std;  class CTString  {  char \*m\_pData; // 保存字符数据 的指针  int m\_nLen; // 记录字符长度    public:  // 构造函数  CTString();  CTString(const char\*);  CTString(const CTString&); // 拷贝构造函数  // 析构函数  ~CTString();    // 其他成员函数  CTString\* Copy(CTString\*); // 拷贝  CTString\* Connect(CTString\*); // 连接  char\* Find(CTString\*);  //char\* Find(char);  void Print();  };  // 无参构造函数  CTString::CTString()  {  m\_nLen = 0;  m\_pData = NULL;  }  // 有参构造函数  CTString::CTString(const char \*pString)  {  if (pString == NULL)  {  m\_nLen = 0;  m\_pData = NULL;  }  else  {  m\_nLen = strlen(pString);  m\_pData = new char[m\_nLen + 1]; // 数组下标从1开始，要+1  assert(m\_pData != NULL);  strcpy(m\_pData, pString);  }  }  // 拷贝构造函数  CTString::CTString(const CTString &srcString)  {  m\_nLen = srcString.m\_nLen;  if (m\_nLen == 0)  m\_pData = NULL;  else  {  m\_pData = new char[m\_nLen + 1];  assert(m\_pData != NULL);  strcpy(m\_pData, srcString.m\_pData);  }  }  // 析构函数  CTString::~CTString()  {  if (m\_pData != NULL)  delete []m\_pData;  }  // 成员函数  // connect  CTString\* CTString::Connect(CTString \*pString)  {  assert(pString != NULL);  if (pString->m\_nLen == 0)  {  return this;  }    char\* pStrTemp = m\_pData;    m\_nLen = m\_nLen + pString->m\_nLen;  m\_pData = new char[m\_nLen+1];  assert(m\_pData != NULL);    strcpy(m\_pData, pStrTemp);  strcat(m\_pData, pString->m\_pData);  delete pStrTemp;  return this;  }  // copy  CTString\* CTString::Copy(CTString \*pString)  {  assert(pString != NULL);  if (m\_pData != NULL)  delete m\_pData;  if (pString->m\_nLen == 0)  {  m\_pData = NULL;  return this;  }    m\_nLen = pString->m\_nLen;  m\_pData = new char[m\_nLen+1];  assert(m\_pData != NULL);  strcpy(m\_pData, pString->m\_pData);  return this;  }  // find 字符串  char\* CTString::Find(CTString\* pSubString)  {  assert(pSubString != NULL);  if (pSubString->m\_nLen == 0)  {  return NULL;  }  char \*pRet = NULL;  pRet = strstr(m\_pData, pSubString->m\_pData);  return pRet;  }  // 如果这样编写Print函数，思考题是有运行时错误的  void CTString::Print()  {  //cout<<"-------------------------------------------"<<endl;  cout<<"该字符串的值："<<m\_pData<<endl;  cout<<"该字符串的长度："<<m\_nLen<<endl;  //cout<<"-------------------------------------------"<<endl;  cout<<endl;  }  int main()  {  CTString s1;  CTString \*s5,\*s7;  CTString s2("Hello");  CTString s4(" ZhangQingChen");  CTString s6("Chen");  CTString s3(s4);  s5 = &s2;  s7 = &s4;  s2.Print();  //s1.Print(); // 有问题的语句  cout<<"-----------连接字符串-----------"<<endl;  s2.Connect(&s4);  s2.Print();  cout<<"-----------查找字符串-----------"<<endl;  char \*pstr = NULL;  pstr = s2.Find(&s6);  if (pstr != NULL)  {  cout << pstr << endl;  }  return 0;    } |

**（二） 运行结果：**



**（三） 思考题**

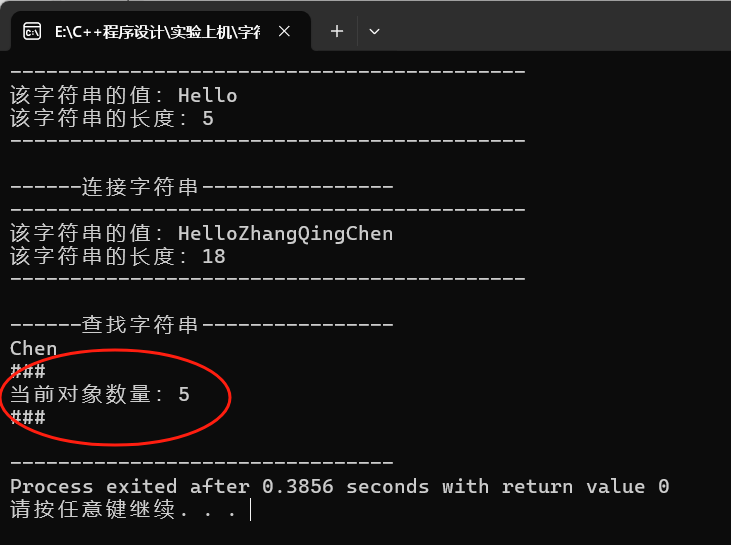
**1．**

**报错：** 145 20 E:\C++程序设计\实验上机\字符串类设计与使用\_思考题1.cpp [Error] no matching function for call to 'CTString::Copy(const char [6])'

**原因：**传入的参数类型不正确，需要传入的是 CTString类实例化的对象，不能是字符串，所以 Copy("hello") 报错。

**2．**

**改写：**添加一个静态成员变量number，每次调用构造函数或者拷贝构造函数时number++，调用析构函数时number--。







**C++上机报告**

**实验题目 　 派生类的设计与使用**

**学 院 　计算机科学与工程学院**

**专 业 　 计算机科学与技术**

**班 级 　 计算机2101班**

**姓 名 　　　　　张清晨**

**学 号 2104230414**

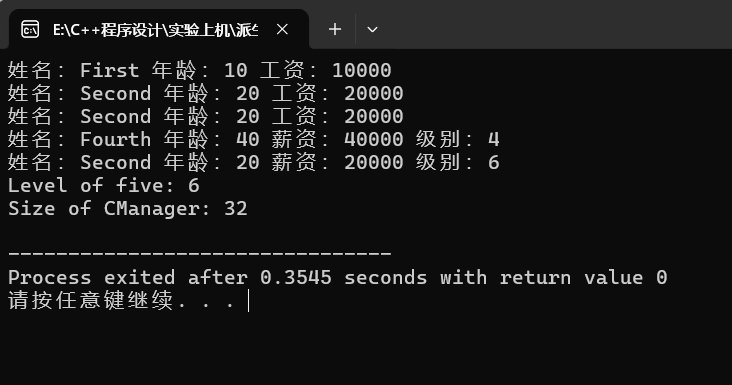
**成 绩**

**指导教师 　 宋晓宇**

1. **代码实现**

|  |
| --- |
| #include<iostream>  #include<cstring>  using namespace std;  class CEmpoyee  {  public:  char\* m\_pName;//姓名  int m\_nAge; //年龄  float m\_fSalary;//薪水  public:  //构造函数和析构函数  CEmpoyee(char\* pName = NULL, int age = 0, float salary = 0.0);  CEmpoyee(const CEmpoyee&);  virtual ~CEmpoyee();  //其他成员函数  void SetName(char\* p\_name);  char\* Getname();  void SetAge(int age);  int GetAge();  void SetSalary(float salary);  float Getsalary();  void Print();  };  //class CManager :private CEmpoyee  class CManager :public CEmpoyee  {  public:  int m\_nlevel;//级别  public:  //构造函数和析构函数  CManager(char\* pName = NULL, int age = 0, float salary = 0.0, int nLevel = 0);  CManager(const CEmpoyee&, int);  ~CManager();  //其他成员函数  void SetLevel(int);  int GetLevel();  void Print();  };  // CEmpoyee 成员函数：  //构造函数  CEmpoyee::CEmpoyee(char\* pName, int age, float salary)  {  if (pName == NULL)  {  return;  }  this->m\_pName = new char[strlen(pName) + 1];  strcpy(this->m\_pName, pName);  //for (int i = 0; i <= strlen(pName); i++)  //{  // this->m\_pName[i] = pName[i];  //}  this->m\_nAge = age;  this->m\_fSalary = salary;  }  //拷贝构造函数  CEmpoyee::CEmpoyee(const CEmpoyee& empoyee)  {  if (empoyee.m\_pName != NULL)  {  this->m\_pName = new char[strlen(empoyee.m\_pName) + 1];  strcpy(this->m\_pName, empoyee.m\_pName);  }  else{  this->m\_pName = NULL;  }  // for (int i = 0; i <= strlen(empoyee.m\_pName); i++)  // {  // this->m\_pName[i] = empoyee.m\_pName[i];  // }  this->m\_nAge = empoyee.m\_nAge;  this->m\_fSalary = empoyee.m\_fSalary;  }  //析构函数  CEmpoyee::~CEmpoyee()  {  if (this->m\_pName != NULL)  {  delete[] this->m\_pName;  this->m\_pName = NULL;  }  }  //设置名字  void CEmpoyee::SetName(char\* p\_name)  {  if (this->m\_pName != NULL)  {  delete[] this->m\_pName;  this->m\_pName = NULL;  }  if (p\_name != NULL)  {  this->m\_pName = new char[strlen(p\_name) + 1];  strcpy(this->m\_pName, p\_name);  }  else  {  this->m\_pName = NULL;  }  }  //获取名字  char\* CEmpoyee::Getname()  {  return this->m\_pName;  }  //设置年龄  void CEmpoyee::SetAge(int age)  {  this->m\_nAge = age;  }  //获取年龄  int CEmpoyee::GetAge()  {  return this->m\_nAge;  }  //设置薪水  void CEmpoyee::SetSalary(float salary)  {  this->m\_fSalary = salary;  }  //获取薪水  float CEmpoyee::Getsalary()  {  return this->m\_fSalary;  }  //打印信息  void CEmpoyee::Print()  {  cout << "姓名：" << this->Getname() << " 年龄：" << this->GetAge() << " 工资：" << this->Getsalary() << endl;  }  // CManager 成员函数：  //构造函数  CManager::CManager(char\* pName, int age, float salary, int nLevel) :CEmpoyee(pName, age, salary)  {  this->m\_nlevel = nLevel;  }  //拷贝构造函数  CManager::CManager(const CEmpoyee& Empoyee, int level) :CEmpoyee(Empoyee)  {  this->m\_nlevel = level;  }  //析构函数  CManager::~CManager()  {}  //设置级别  void CManager::SetLevel(int level)  {  this->m\_nlevel = level;  }  //获取级别  int CManager::GetLevel()  {  return this->m\_nlevel;  }  //打印输出  void CManager::Print()  {  cout << "姓名：" << this->Getname() << " 年龄：" << this->GetAge() << " 薪资：" << this->Getsalary() << " 级别：" << this->GetLevel() << endl;  }  int main()  {  CEmpoyee one("First", 10, 10000);  CEmpoyee two("Second", 20, 20000);  CEmpoyee three(two);    one.Print();  two.Print();  three.Print();    CManager four("Fourth", 40, 40000, 4);  CManager five(three, 5);  five.SetLevel(6);    four.Print();  five.Print();    cout << "Level of five: " << five.GetLevel() << endl;  cout << "Size of CManager: " << sizeof(five) << endl;    return 0;  } |

1. **运行结果**



1. **思考题**

**1．**

* 初始化方式：

构造函数初始化列表：在CManager的构造函数中，通过构造函数初始化列表显式调用了基类CEmpoyee的构造函数。例如，在CManager的两个构造函数定义中：

|  |
| --- |
| CManager(char\* pName, int age, float salary, int nLevel) : CEmpoyee(pName, age, salary)  {  this->m\_nlevel = nLevel;  }  CManager(const CEmpoyee& Empoyee, int level) : CEmpoyee(Empoyee)  {  this->m\_nlevel = level;  } |

这里，不论是传入具体参数还是一个CEmpoyee对象，都确保了基类部分在派生类对象创建时得到恰当的初始化。

* 销毁方式：

析构函数：虽然在CManager类中没有显式编写析构函数的具体实现（除了空的析构函数体~CManager() {}），但派生类对象销毁时，C++编译器会自动调用基类的析构函数。这意味着，即使在派生类的析构函数中没有显式写出来，CEmpoyee类的析构函数也会被正确调用，从而释放基类对象占用的资源。具体到这个例子中，CEmpoyee的析构函数会负责释放动态分配给m\_pName的内存。

**2．**

答：Y类中的print函数不能正常编译运行。

原因：

在Y类的print函数内部，调用print()时，编译器无法确定你想要调用的是Y类本身的print函数，还是基类X中的print函数。

X::print(); 才可以。





**C++上机报告**

**实验题目 　 运算符重载**

**学 院 　计算机科学与工程学院**

**专 业 　 计算机科学与技术**

**班 级 　 计算机2101班**

**姓 名 　　　　　张清晨**

**学 号 2104230414**

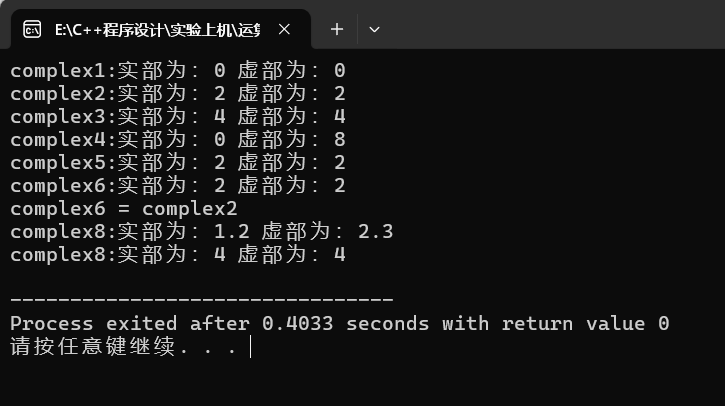
**成 绩**

**指导教师 　 宋晓宇**

1. **代码实现**

|  |
| --- |
| #include <iostream>  #include <cmath>  using namespace std;  class CComplex  {  float real;  float imag;  public:  CComplex()  {  this->real = 0.0;  this->imag = 0.0;  }  CComplex(float real, float imag)  {  this->real = real;  this->imag = imag;  }  CComplex& operator=(CComplex& complex)  {  this->real = complex.real;  this->imag = complex.imag;  return \*this;  }  CComplex operator\*(CComplex& complex)  {  CComplex temp;  temp.real = this->real \* complex.real - this->imag \* complex.imag;  temp.imag = this->real \* complex.imag + complex.imag \* this->real;  return temp;  }  CComplex operator+(CComplex& complex)  {  CComplex temp;  temp.real = this->real + complex.real;  temp.imag = this->imag + complex.imag;  return temp;  }  CComplex operator-(CComplex& complex)  {  CComplex temp;  temp.real = this->real - complex.real;  temp.imag = this->imag - complex.imag;  return temp;  }  CComplex operator/(CComplex& complex)  {  if (complex.imag == 0 || complex.real == 0)  {  cout << "zero can't be div" << endl;  exit(0);  }  CComplex temp;  temp.real = (this->real \* complex.real + this->imag \* complex.imag) / (pow(complex.real, 2) + pow(complex.imag, 2));  temp.imag = (this->imag \* complex.real - this->real \* complex.imag) / (pow(complex.real, 2) + pow(complex.imag, 2));  return temp;  }  CComplex operator[](int index)  {  }  friend bool operator==(CComplex& complex1, CComplex& complex2);  friend bool operator!=(CComplex& complex1, CComplex& complex2);  void print()  {  cout << "实部为：" << this->real << " 虚部为：" << this->imag << endl;  }  };  bool operator==(CComplex& complex1, CComplex& complex2)  {  if (complex1.real == complex2.real && complex1.imag == complex2.imag)  return true;  else  return false;  }  bool operator!=(CComplex& complex1, CComplex& complex2)  {  if (complex1.real != complex2.real || complex1.imag != complex2.imag)  return true;  else  return false;  }  int main()  {  CComplex complex1;  cout << "complex1:";  complex1.print();  CComplex complex2(2.0, 2.0);  cout << "complex2:";  complex2.print();  CComplex complex3 = complex2 + complex2;  cout << "complex3:";  complex3.print();  CComplex complex4 = complex2 \* complex2;  cout << "complex4:";  complex4.print();  CComplex complex5 = complex3 - complex2;  cout << "complex5:";  complex5.print();  CComplex complex6 = complex4 / complex2;  cout << "complex6:";  complex6.print();  if (complex6 == complex2)  {  cout << "complex6 = complex2" << endl;  }  else  {  cout << "complex6 != complex2" << endl;  }  CComplex complex8(1.2, 2.3);  cout << "complex8:";  complex8.print();  complex8 = complex3;  cout << "complex8:";  complex8.print();  return 0;  } |

1. **运行结果**



1. **思考题**

**1.**

答：C语言不支持函数重载是因为其设计追求简洁高效，编译器复杂度较低，且类型系统相对简单，不符合函数重载所需的类型检查和解析机制。而C++为了增强面向对象编程能力，引入了更复杂的类型系统和编译器技术，从而支持函数重载，提高了代码的可读性和灵活性。

**2.**

答：->和 () 两个运算符重载：

|  |
| --- |
| struct ComplexData {  float data;  };  ComplexData\* pData = nullptr;  ComplexData\* operator->()  {  if (pData == nullptr)  {  pData = new ComplexData();  }  return pData;  }  float operator()() const  {  return sqrt(real \* real + imag \* imag);  } |

**3.**

答：虚函数的内存结构有以下好处：

1. 多态性：

虚函数允许基类指针或引用指向派生类的对象，并能够通过这个指针或引用调用正确的派生类函数，即使不知道具体是哪个派生类的对象。

1. 动态绑定：

虚函数的调用是通过虚函数表（Virtual Function Table，简称 VFT 或 vtable）来实现的，这是一种运行时动态绑定技术。这意味着函数的调用不是在编译时确定的，而是在运行时根据对象的实际类型来确定调用哪个函数。

1. 扩展性：

虚函数使得基类可以包含派生类的行为，而不需要知道具体派生类的实现细节。这使得基类可以被扩展，而无需修改基类的代码。

1. 继承和封装：

虚函数是实现继承和封装的重要手段。派生类可以通过重写基类的虚函数来实现多态性，同时保持封装，即隐藏内部实现细节。

1. 提高代码的可维护性：

虚函数使得代码更加模块化，易于理解和维护。开发者可以专注于派生类的行为，而不需要关心基类如何实现。

1. 减少代码冗余：

虚函数允许基类提供一种通用行为，而派生类可以重写这些函数以提供特定行为。这样可以减少代码冗余，提高代码的复用性。

1. 灵活性：

虚函数提供了更大的灵活性，使得程序可以更容易地适应未来的变化和扩展。

动态多态：允许运行时确定调用对象的实际函数版本，实现接口重用和行为动态变化。





**C++上机报告**

**实验题目 　 类模板设计和使用**

**学 院 　计算机科学与工程学院**

**专 业 　 计算机科学与技术**

**班 级 　 计算机2101班**

**姓 名 　　　　　张清晨**

**学 号 2104230414**

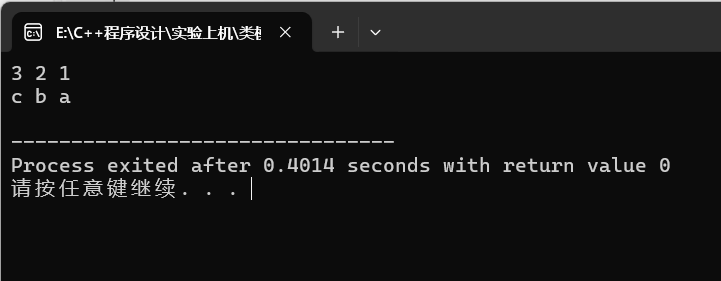
**成 绩**

**指导教师 　 宋晓宇**

1. **代码实现**

|  |
| --- |
| #include <iostream>  using namespace std;  template<class T>  class LinkStack  {  private:  struct Node // 定义链表节点结构体  {  T data; // 节点存储的数据  Node\* next; // 指向下一个节点的指针  Node(const T& d = T(), Node\* n = nullptr) : data(d), next(n) {}  };  Node\* top; // 栈顶指针  public:  LinkStack(); // 构造函数  ~LinkStack(); // 析构函数  bool IsEmpty() const; // 判断栈是否为空  void Push(const T& x); // 入栈操作  T Pop(); // 出栈操作  };  template<class T>  LinkStack<T>::LinkStack() : top(nullptr) {}  template<class T>  LinkStack<T>::~LinkStack()  {  if (top != nullptr) {  Node\* p;  p = top;  top = top->next;  delete p;  }  }  template<class T>  bool LinkStack<T>::IsEmpty() const  {  return top == nullptr;  }  template<class T>  void LinkStack<T>::Push(const T& x)  {  Node\* p;  p = new Node;  p->data = x;  if (top) p->next = top;  top = p;  }  template<class T>  T LinkStack<T>::Pop()  {  Node\* p;  T i;  i = top->data;  p = top;  top = top->next;  delete p;  return i;  }  int main()  {  LinkStack<int> s;  s.Push(1);  s.Push(2);  s.Push(3);  while (!s.IsEmpty())  {  cout << s.Pop() << " ";  }  cout << endl;  LinkStack<char> m;  m.Push('a');  m.Push('b');  m.Push('c');  while (!m.IsEmpty())  {  cout << m.Pop() << " ";  }  cout << endl;  return 0;  } |

1. **运行结果**

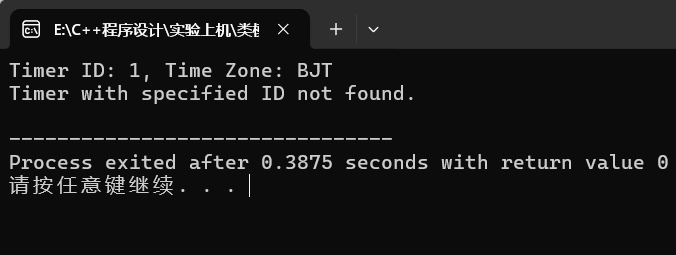


1. **附加题**

**1 代码实现**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <stdexcept>  // 基类CTimer  class CTimer {  protected:  unsigned long m\_TimerID; // 时钟编号  public:  CTimer(unsigned long id) : m\_TimerID(id) {}  virtual ~CTimer() {}  // 获取时区代号的虚函数，需要子类重载  virtual std::string GetTimeZone() const = 0;  // 获取TimerID的纯虚函数  virtual unsigned long GetTimerID() const = 0;  // 用于测试打印当前对象的时区  void PrintTimeZone() const {  std::cout << "Timer ID: " << GetTimerID() << ", Time Zone: " << GetTimeZone() << std::endl;  }  };  // 派生类CBeijingTimer  class CBeijingTimer : public CTimer {  public:  CBeijingTimer(unsigned long id) : CTimer(id) {}  // 重载GetTimeZone函数  std::string GetTimeZone() const override {  return "BJT"; // 北京时间代号  }  // 实现GetTimerID  unsigned long GetTimerID() const override {  return CTimer::m\_TimerID;  }  };  // 时钟管理类  class TimerManager {  private:  std::vector<CTimer\*> m\_Timers; // 保存所有Timer的指针  public:  void AddTimer(CTimer\* timer) {  m\_Timers.push\_back(timer);  }  void RemoveTimerByID(unsigned long id) {  for (auto it = m\_Timers.begin(); it != m\_Timers.end(); ++it) {  if ((\*it)->GetTimerID() == id) {  delete \*it;  m\_Timers.erase(it);  return;  }  }  throw std::runtime\_error("Timer with specified ID not found.");  }  void PrintAllTimeZones() const {  for (const auto& timer : m\_Timers) {  timer->PrintTimeZone();  }  }  CTimer\* FindTimerByID(unsigned long id) const {  for (const auto& timer : m\_Timers) {  if (timer->GetTimerID() == id) {  return timer;  }  }  return nullptr;  }  };  int main() {  TimerManager manager;  // 创建北京时钟并添加  CBeijingTimer\* beijingTimer = new CBeijingTimer(1);  manager.AddTimer(beijingTimer);  // 打印所有时区  manager.PrintAllTimeZones();  // 尝试删除不存在的TimerID  try {  manager.RemoveTimerByID(999);  } catch (const std::exception& e) {  std::cerr << e.what() << std::endl;  }  // 删除刚才添加的北京时钟  manager.RemoveTimerByID(1);  return 0;  } |

**2 运行结果**



1. **思考题**

定义基类模板 Array：

|  |
| --- |
| template <typename T>  class Array {  public:  Array(int size) : m\_size(size), m\_array(new T[size]) {}  ~Array() { delete[] m\_array; }  T& operator[](int index) { return m\_array[index]; }  const T& operator[](int index) const { return m\_array[index]; }  private:  int m\_size;  T\* m\_array;  }; |

定义派生类模板 SortedArray，它基于 Array 类模板：

|  |
| --- |
| template <typename T>  class SortedArray : public Array<T> {  public:  SortedArray(int size) : Array<T>(size) {}  void Sort() {    }  }; |