# C++对C加强专题讲座

## 程序发展历程

|  |
| --- |
| 当面向过程方法论暴露越来越多的缺陷的时候，业界开始考虑在工程项目中引入面向对象的设计方法，而第一个需要解决的问题就是：高效的面向对象语言，并且能够兼容已经存在的代码。  **C语言 + 面向对象方法论===》Objective C /C++** |
| C语言和C++并不是对立的竞争关系  C++是C语言的加强，是一种更好的C语言  C++是以C语言为基础的，并且完全兼容C语言的特性 |

## namespace命名空间

|  |
| --- |
| C中的命名空间  在C语言中只有一个全局作用域  C语言中所有的全局标识符共享同一个作用域  标识符之间可能发生冲突  C++中提出了命名空间的概念  命名空间将全局作用域分成不同的部分  不同命名空间中的标识符可以同名而不会发生冲突  命名空间可以相互嵌套  全局作用域也叫默认命名空间 |
| 1. 当使用<iostream>的时候，该[头文件](http://baike.baidu.com/view/668911.htm)没有定义全局命名空间，必须使用namespace std；这样才能正确使用cout。若不引入using namespace std ,需要这样做。std::cout。 2. c++标准为了和C区别开，也为了正确使用命名空间，规定[头文件](http://baike.baidu.com/view/668911.htm)不使用后缀.h。 3. C++命名空间的定义： namespace name { … } 4. using namespace NameSpaceA;   5）namespce定义可嵌套 |

实用性增强

|  |
| --- |
| //C语言中的变量都必须在作用域开始的位置定义！！  //C++中更强调语言的“实用性”，所有的变量都可以在需要使用时再定义。 |

## register关键字增强

|  |
| --- |
| register关键字的变化  register关键字请求“编译器”将局部变量存储于寄存器中  C语言中无法取得register变量地址  在C++中依然支持register关键字  C++编译器有自己的优化方式，不使用register也可能做优化  C++中可以取得register变量的地址  //2  C++编译器发现程序中需要取register变量的地址时，register对变量的声明变得无效。  //3  早期C语言编译器不会对代码进行优化，因此register变量是一个很好的补充。 |

## 函数检测增强

|  |
| --- |
|  |

## struct类型加强

|  |
| --- |
| struct类型的加强：  C语言的struct定义了一组变量的集合，C编译器并不认为这是一种新的类型  C++中的struct是一个新类型的定义声明 |

## C++中类型检测更加严格

|  |
| --- |
| /\*  在C语言中  int f( )；表示返回值为int，接受任意参数的函数  int f(void)；表示返回值为int的无参函数  在C++中  int f( );和int f(void)具有相同的意义，都表示返回值为int的无参函数  \*/  C++更加强调类型，任意的程序元素都必须显示指明类型 |

## bool类型

|  |
| --- |
| C++中的布尔类型  C++在C语言的基本类型系统之上增加了bool  C++中的bool可取的值只有true和false  理论上bool只占用一个字节，  如果多个bool变量定义在一起，可能会各占一个bit，这取决于编译器的实现  true代表真值，编译器内部用1来表示  false代表非真值，编译器内部用0来表示  bool类型只有true（非0）和false（0）两个值  C++编译器会在赋值时将非0值转换为true，0值转换为false |

## 三目运算符增强

|  |
| --- |
| int main()  {  int a = 10;  int b = 20;  //返回一个最小数 并且给最小数赋值成3  //三目运算符是一个表达式 ，表达式不可能做左值  (a < b ? a : b )= 30;  printf("a = %d, b = %d\n", a, b);  system("pause");  return 0;  }  1）C语言返回变量的值 C++语言是返回变量本事  2）  C语言中的三目运算符返回的是变量值，不能作为左值使用  C++中的三目运算符可直接返回变量本身，因此可以出现在程序的任何地方  注意：  3）  三目运算符可能返回的值中如果有一个是常量值，则不能作为左值使用  (a < b ? 1 : b )= 30;  4）C语言如何支持类似C++的特性那？  变量的本质是内存空间的别名，是一个标号。 |

## const专题讲座

|  |
| --- |
| const int a;  int const b;  const int \*c;  int \* const d;  const int \* const e ; |
| //第一个第二个意思一样 代表一个常整形数  //第三个 c是一个指向常整形数的指针(所指向的内存数据不能被修改，但是本身可以修改)  //第四个 d 常指针（指针变量不能被修改，但是它所指向内存空间可以被修改）  //第五个 e一个指向常整形的常指针（指针和它所指向的内存空间，均不能被修改） |
| C语言中的const变量  C语言中const变量是只读变量，有自己的存储空间  C++中的const常量  可能分配存储空间,也可能不分配存储空间  当const常量为全局，并且需要在其它文件中使用  当使用&操作符取const常量的地址 |
| int main()  {  const int a = 1;  const int b = 2;  int array[a + b] = {0};  int i = 0;    for(i=0; i<(a+b); i++)  {  printf("array[%d] = %d\n", i, array[i]);  }    printf("Press enter to continue ...");  getchar();    return 0;  } |
| C++中的const常量类似于宏定义  const int c = 5; ≈ #define c 5  C++中的const常量在与宏定义不同  const常量是由编译器处理的，提供类型检查和作用域检查  宏定义由预处理器（CPP）处理，单纯的文本替换 |
| void fun1()  {  #define a 10  const int b = 20;  //#undef a  } |
|  |

## 引用专题讲座

|  |
| --- |
| 1. 引用是C++的概念，属于C++编译器对C的扩展 |
| 2、引用的本质  1）引用在C++中的内部实现是一个常指针  Type& name 🡸🡺Type\* const name  2）C++编译器在编译过程中使用常指针作为引用的内部实现，因此引用所占用的空间大小与指针相同。  3）从使用的角度，引用会让人误会其只是一个别名，没有自己的存储空间。这是C++为了实用性而做出的细节隐藏 |
| 引用在实现上，只不过是把：  间接赋值成立的三个条件的后两步和二为一。  //当实参传给形参引用的时候，只不过是c++编译器帮我们程序员手工取了一个实参地址，传给了形参引用（常量指针） |
| 引用当左值 |
| //基础类型a返回的时候，也会有一个副本  int& getAA2()  {  int a;  a = 10;  return a;  }  void main21()  {  int a1 = 0;  int a2 = 0;  a1 = getAA1();  a2 = getAA2();  int &a3 = getAA2(); //乱码  printf("a1:%d \n", a1);  printf("a2:%d \n", a2);  printf("a3:%d \n", a3);  system("pause");  } |
| 当函数返回值为引用时  若返回栈变量  不能成为其它引用的初始值  不能作为左值使用  若返回静态变量或全局变量  可以成为其他引用的初始值  即可作为右值使用，也可作为左值使用  C++链式编程中，经常用到引用，运算符重载专题 |
| 指针引用 |
| //指针的引用而已  int getTe2(Teacher\* &myp)  {  myp = (Teacher \*)malloc(sizeof(Teacher));  myp->age = 34;  return 0;  } |
| 1 const引用让变量拥有只读属性 |
| void printTe2(const Teacher1 \*const pt)  {  }  //const引用让变量(所指内存空间)拥有只读属性  void printTe(const Teacher1 &t)  {  //t.age = 11;  } |
| 2 Const引用 用字面量初始化const引用 |
| void main()  {  const int b = 10;  printf("b:%d", &b);  //int &a1 = 19; 如果不加const编译失败  const int &a = 19;  printf("&a:%d \n", &a);  system("pause");  } |
|  |
|  |

## C++对C函数加强专题讲座

### 内联函数

|  |
| --- |
| 结论： 1）内联函数在编译时直接将函数体插入函数调用的地方  2）inline只是一种请求，编译器不一定允许这种请求  3）内联函数省去了普通函数调用时压栈，跳转和返回的开销 |
|  |

### 默认参数及默认参数的规则

### 默认参数和占位参数

|  |
| --- |
| int func(int a, int b, int = 0)  {  return a + b;  } |

### 函数重载

|  |
| --- |
| 函数重载(Function Overload)  用同一个函数名定义不同的函数  当函数名和不同的参数搭配时函数的含义不同 |
| 函数重载至少满足下面的一个条件：  参数个数不同  参数类型不同  参数顺序不同 |
| 函数重载调用规则。。。。。 |
| //当函数默认参数遇上函数重载会发生什么  /\*  int func(int a, int b, int c = 0)  {  return a \* b \* c;  }  Int func(int a, int b)  {  return a + b;  }  int main()  {  int c = 0;  c = func(1, 2); // 存在二义性，调用失败，编译不能通过  printf("c = %d\n", c);  printf("Press enter to continue ...");  getchar();  return 0;  }  \*/ |
| 函数重载与函数指针 |
|  |

# C++中类的封装

基本概念

|  |
| --- |
| //类是把属性和方法封装。。。  //类的内部  //类的外部  //我们抽象了一个类，用类去定义对象  //类是一个数据类型，类是抽象的  //对象是一个具体的变量。。占用内存空间。  //类做函数参数的时候，类封装了属性和方法，在被调用函数里面， 不但可以使用属性，而且可以使用方法（成员函数）； |
| //面向过程编程加工的是：函数  //面向对象编程加工的是：类 |
| 类的访问控制，三个关键字  public  成员变量和成员函数可以在类的内部和外界访问和调用  private  成员变量和成员函数只能在类的内部被访问和调用  类的真正形态  在用struct定义类时，所有成员的默认属性为public  在用class定义类时，所有成员的默认属性为private |

# C++中的构造和析构专题讲座

## 为什么对象需要初始化 显示调用方案及局限性分析

|  |
| --- |
|  |
| void main()  {  Test t1;  t1.init(10);  //不好用蹩脚的地方  //  Test t11, t12, t13;  t11.init(11);  t12.init(12);  t13.init(13);  Test array[3999] = {t11, t12, t13};  system("pause");  } |

## C++编译器提供的构造方案

|  |
| --- |
| class Test  {  public:  //构造函数 无参构造函数 默认构造函数  //  Test()  {  a = 10;  }    //带参数的构造函数  //调用方法3中  Test(int mya)  {  a = mya;  }    //第三中初始化对象的方法  //赋值构造函数 copy构造函数  //copy构造函数的用法 4中应用场景  Test(const Test & obj)  {  printf("我完成对象的初始化，我也是是构造函数aaaaa")  }  protected:  private:  int a;  }; |
|  |

## 有参构造函数的三种调用方法

|  |
| --- |
| void main()  {  //1 （）  Test t1(10); //c++默认调用有参构造函数 自动调用  // =  Test t2 = 11; //c++默认调用有参构造函数自动调用  //手工调  Test t3 = Test(12); //我们程序员手动调用构造函数  } |
|  |

## 赋值构造函数（copy构造函数）四种应用场景

|  |
| --- |
| 第一个应用场景  //单独搭建一个舞台  void ObjPlay01()  {  AA a1; //变量定义    //赋值构造函数的第一个应用场景  //我用对象1 初始化 对象2  AA a2 = a1; //定义变量并初始化  a2 = a1; //用a1来=号给a2 编译器给我们提供的浅copy  } |
| 第二个应用场景  //单独搭建一个舞台  void ObjPlay02()  {  AA a1(10); //变量定义  //赋值构造函数的第一个应用场景  //我用对象1 初始化 对象2  AA a2(a1); //定义变量并初始化  //a2 = a1; //用a1来=号给a2 编译器给我们提供的浅copy  a2.getA();  } |
| 第三个应用场景  //alt + f8 排版  void f ( Location p )  {  cout << "Funtion:" << p.GetX() << "," << p.GetY() << endl ;  }  void mainobjplay()  {  Location A ( 1, 2 ) ;  f ( A ) ;  } |
| 第四个应用场景  //40 =等号操作  //42 对象初始化操作  //对象初始化操作 和 =等号操作 是两个不同的概念  void mainobjplay()  {  //Location B;  //B = g();  Location B = g();  //如果返回的匿名对象，来初始化另外一个同类型的类对象，那么匿名对象会直接转成新的对象。。。  //匿名对象的去和留，关键看，返回时如何接过来。  cout<<"测试测试"<<endl;  } |
| //40 =等号操作  //42 对象初始化操作 是两个不同概念 |
|  |
|  |

## 深copy浅copy专题

|  |
| --- |
|  |
| class Name  {  public:  Name(const char \*pname)  {  size = strlen(pname);  pName = (char \*)malloc(size + 1);  strcpy(pName, pname);  }  Name(Name &obj)  {  //用obj来初始化自己  pName = (char \*)malloc(obj.size + 1);  strcpy(pName, obj.pName);  size = obj.size;  }  ~Name()  {  cout<<"开始析构"<<endl;  if (pName!=NULL)  {  free(pName);  pName = NULL;  size = 0;  }  }  void operator=(Name &obj3)  {  if (pName != NULL)  {  free(pName);  pName = NULL;  size = 0;  }  cout<<"测试有没有调用我。。。。"<<endl;  //用obj3来=自己  pName = (char \*)malloc(obj3.size + 1);  strcpy(pName, obj3.pName);  size = obj3.size;  }  protected:  private:  char \*pName;  int size;  };  //对象的初始化 和 对象之间=号操作是两个不同的概念  void playObj()  {  Name obj1("obj1.....");  Name obj2 = obj1; //obj2创建并初始化  Name obj3("obj3...");  //重载=号操作符  obj2 = obj3; //=号操作  cout<<"业务操作。。。5000"<<endl;  }  void main61()  {  playObj();  system("pause");  } |
|  |

## 类的三种构造函数的调用规则研究

|  |
| --- |
| /\*  1 当类中没有定义任何一个构造函数时，c++编译器会提供无参构造函数和拷贝构造函数  2 当类中定义了任意的非拷贝构造函数（无参、有参），c++编译器不会提供无参构造函数  3 当类中定义了拷贝构造函数时，c++编译器不会提供无参数构造函数  4 默认拷贝构造函数成员变量简单赋值  总结：只要你写了构造函数，那么你必须用。  \*/ |
| 构造函数是C++中用于初始化对象状态的特殊函数  构造函数在对象创建时自动被调用  构造函数和普通成员函数都遵循重载规则  拷贝构造函数是对象正确初始化的重要保证  必要的时候，必须手工编写拷贝构造函数 |
| 构造函数和析构函数调用顺序总结  构造函数与析构函数的调用顺序  当类中有成员变量是其它类的对象时  首先调用成员变量的构造函数  调用顺序与声明顺序相同  之后调用自身类的构造函数  析构函数的调用秩序与对应的构造函数调用秩序相反 |
|  |

## 对象组合中的初始化列表（为什么有，怎么样的解决方案）

|  |
| --- |
|  |
| //对象做函数参数  //1 研究拷贝构造  //2 研究构造函数，析构函数的调用顺序  //总结 构造和析构的调用顺序  #include "iostream"  using namespace std;  class ABCD  {  public:  ABCD(int a, int b, int c)  {  this->a = a;  this->b = b;  this->c = c;  printf("ABCD() construct, a:%d,b:%d,c:%d \n", this->a, this->b, this->c);  }  ~ABCD()  {  printf("~ABCD() construct,a:%d,b:%d,c:%d \n", this->a, this->b, this->c);  }  int getA()  {  return this->a;  }  protected:  private:  int a;  int b;  int c;  };  class MyE  {  public:  MyE():abcd1(1,2,3),abcd2(4,5,6),m(100)  {  cout<<"MyD()"<<endl;  }  ~MyE()  {  cout<<"~MyD()"<<endl;  }  MyE(const MyE & obj):abcd1(7,8,9),abcd2(10,11,12),m(100)  {  printf("MyD(const MyD & obj)\n");  }  protected:  //private:  public:  ABCD abcd1; //c++编译器不知道如何构造abc1  ABCD abcd2;  const int m;  };  int doThing(MyE mye1)  {  printf("doThing() mye1.abc1.a:%d \n", mye1.abcd1.getA());  return 0;  }  int run2()  {  MyE myE;  doThing(myE);  return 0;  }  //  int run3()  {  printf("run3 start..\n");  ABCD abcd = ABCD(100, 200, 300);  //若直接调用构造函数哪  //想调用构造函数对abc对象进行再复制，可以吗？  //在构造函数里面调用另外一个构造函数，会有什么结果？  //ABCD(400, 500, 600); //临时对象的生命周期  printf("run3 end\n");  return 0;  }  int main()  {  //run2();  run3();  system("pause");  return 0;  } |

## 构造析构综合练习

|  |
| --- |
| 直接调用构造函数，关键看有没有对象去接受匿名对象。。 |
| 构造函数中调用构造函数 |
|  |

## newdelete专题

|  |
| --- |
| //1 new delete 操作符号 c++的关键字  //malloc free函数  //2 new 在堆上分配内存 delete  //分配基础类型 、分配数组类型、分配对象 |
| void main()  {  Test t1(1, 2);//在临时区分配内存  //new + 类型 返回一个 内存首地址  //new操作符也会自动的调用这个类的构造函数 、  //delete自动的调用这个类的析构函数  //相当于我们程序员可以手工控制类的对象的生命周期  Test \*p = new Test(1, 2);  cout<<p->getA()<<endl;  delete p;  getTestObj(&p);  delete p;  system("pause");  } |
| 混搭风。。。深入理解malloc和new的区别、delete和free的区别 |

## static关键字

|  |
| --- |
|  |
| class BB  {  public:  int getC()  {  return c;  }  void setC(int nyc)  {  c = nyc;    }  //静态成员函数是属于整个类，  //在类的静态数据成员函数中，是不能调用具体的对象的变量的属性。。。。（抛砖）  //不能调用普通成员变量  static void getMem()  {  //cout<<a<<endl;  cout<<c<<endl;  }  protected:  private:  int a;  int b;  static int c;  };  int BB::c = 0;  //static修饰的变量，是属于类，，所有的对象都能共享用。  void main111()  {  BB b1;  BB b2;  cout<<b2.getC()<<endl;;  b1.setC(100);  cout<<b2.getC()<<endl;;  system("pause");  }  void main()  {  //调用静态成员函数的方法1  BB::getMem();  //调用静态成员函数的方法2  BB b1;  b1.getMem();  system("pause");  } |

# c++对象模型初探专题讲座（重点）

|  |
| --- |
|  |
| 1）C++类对象中的成员变量和成员函数是分开存储的  成员变量：  普通成员变量：存储于对象中，与struct变量有相同的内存布局和字节对齐方式  静态成员变量：存储于全局数据区中  成员函数：存储于代码段中。  问题出来了：很多对象公用一块代码？代码是如何区分具体对象的那？  换句话说：int getK() const { return k; }，代码是如何区分，具体obj1、obj2、obj3对象的k值？ |
| 2）C++编译器对普通成员函数的内部处理 |
| **总结**  **1、**C++类对象中的成员变量和成员函数是分开存储的。C语言中的内存四区模型仍然有效！  **2、C++中类的普通成员函数都隐式包含一个指向当前对象的this指针。**  3、静态成员函数、成员变量属于类  静态成员函数与普通成员函数的区别  静态成员函数不包含指向具体对象的指针  普通成员函数包含一个指向具体对象的指针 |

# C++运算符重载专题讲座

运算符重载基础

|  |
| --- |
| 全局函数和类成员函数的区别 |
| //全局函数  //如果把全局函数转成成员函数，少了一个操作数，通过this指针，被隐藏。  Test add(Test &t1, Test &t2)  {  Test t3(t1.getA()+t2.getA(), t1.getB() + t2.getB()) ;  return t3;  }  //把成员函数转成全局函数，需要多一个参数。。。。  void printAB(Test \*pthis)  {  cout<<"a:"<<pthis->getA()<<"b:"<<pthis->getB()<<endl;  } |
| //为什么要有运算符重载  //原因 Complex是用户自定义类型。。编译器根本不知道如何进行加减。。。  //编译器给你提供了一种机制，让用户自己去完成，自定义类型的加减操作。。。。。  //这个机制就是运算符重载机制 |
| #include "iostream"  using namespace std;  //a + bi //复数  class Complex  {  public:  friend ostream& operator<<(ostream &out, Complex &c1);  Complex(int a, int b)  {  this->a = a;  this->b = b;  }  void printCom()  {  cout<<a<<" + "<<b<<"i "<<endl;  }  //通过类成员函数完成-操作符重载  Complex operator-(Complex &c2)  {  Complex tmp(a - c2.a, this->b - c2.b);  return tmp;  }  Complex operator+(Complex &c2)  {  Complex tmp(a + c2.a, this->b + c2.b);  return tmp;  }  //通过成员函数完成前置--  Complex& operator--()  {  this->a--;  this->b--;  return \*this;  }  //通过成员函数完成前置--  Complex& operator++()  {  this->a++;  this->b++;  return \*this;  }  //通过类成员函数完成后置--  Complex operator--(int)  {  Complex tmp = \*this;  this->a--;  this->b--;  return tmp;  }  //通过类成员函数完成后置--  Complex operator++(int)  {  Complex tmp = \*this;  this->a++;  this->b++;  return tmp;  }  private:  int a;  int b;  };  void main61()  {  Complex c1(1, 2), c2(3, 4);  //全局函数方法去实现操作符重载  //1 承认操作符重载是一个函数  //2 根据操作数，1个操作数 2个操作数，完成函数的参数  //3 根据函数原型，实现需要业务  //Complex c31 = operator+(c1, c2);  Complex c3 = c1 + c2;  //用类成员函数实现-运算符重载  Complex c4 = c1 - c2;  c4.printCom();  //c1.operator-(c2);  {  //问题1 根据类型  int a = 10, b = 1;  int c = a + b;  }  //前置++ 全局函数  ++c1;  c1.printCom();  //前置-- 成员函数  --c1;  c1.printCom();  //c1.operator--()  //后置++ 全局函数  c1++; //operator++(c1);  c1.printCom();  //后置-- 类成员函数  c1--; //c1.operator--()  c1.printCom();  system("pause");  }  ostream& operator<<(ostream &out, Complex &c1)  {  //out<<"12345，生活真是苦"<<endl;  out<<c1.a<<" + "<<c1.b<<"i "<<endl;  return out;  }  void main()  {  Complex c1(1, 2), c2(3, 4);  int a = 10;  char \*p = "addddd";  cout<<"a"<<a<<endl;  cout<<"p"<<p<<endl;  //Complex自定义类型 。a + bi ab ba  cout<<c1;    //全局函数  //cout<<c1;  //operator<<(cout, c1);  //1  //类成员函数  //因为你没有方法拿到cout这个类的源码。。。。AOP C++  //cout.operator<<(c1);  //2 支持链式编程  cout<<c1<<"abcc";  //函数返回值当左值，要求返回一个引用。。。。。    //cout.operator<<(c1).operator<<("abcd");  //s<<"abcd"  //s cout.operator<<(c1);  //s.operator<<("abcd");  system("pause");  } |
|  |

运算符重载在项目开发中的应用

|  |
| --- |
| //功能1 []运算符放在=右边和放在=左边。。。。  所以重载函数原型 int& Array::operator[](int i);  //一下是运算符重载函数  //从数组里面拿元素  int& Array::operator[](int i)  {  return mSpace[i];  } |
| 功能2：  关键1支持一个=  关键点2：  支持多个=号 Array& Array::operator=(Array &a1);  关键点3：释放旧内存  f (this->mSpace != NULL)  {  delete[] mSpace;  this->mLength = 0;  }  }  关键点4：两个函数原型区别  Array& Array::operator=(Array &a1)  Array Array::operator=(Array &a1)  Array& Array::operator=(Array &a1)  {  int i = 0;  if (this->mSpace != NULL)  {  delete[] mSpace;  this->mLength = 0;  }  //a1的所有元素赋给a3  this->mLength = a1.mLength;  this->mSpace = new int[a1.mLength];  for (i=0; i<this->mLength; i++)  {  mSpace[i] = a1[i];  }  return \*this;  } |
| 功能3、4  bool operator==(Array &a2);  bool operator!=(Array &a2);  //功能3  //if (a1 == a2) //功能3  bool Array::operator==(Array &a2)  {  if (this->mLength != a2.mLength)  {  return false;  }  for (int i=0; i<this->mLength; i++)  {  if (this->mSpace[i] != a2[i])  {  return false;  }  }  return true;  }  // if (a1 != a2) //功能4  bool Array::operator!=(Array &a2)  {  return !(\*this == a2);  } |

# C++继承专题讲座

## 继承基本概念

|  |
| --- |
| 继承概念 |
| C++中的类成员访问级别（public、private、protected）  思考：如何恰当的使用public，protected和private为成员声明访问级别？  1、需要被外界访问的成员直接设置为public  2、只能在当前类中访问的成员设置为private  3、只能在当前类和子类中访问的成员设置为protected，protected成员的访问权限介于public和private之间。 |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 父类成员访问级别 | | | | | 继  承  方  式 |  | public | proteced | private | | public | public | proteced | private | | proteced | proteced | proteced | private | | private | private | private | private | |
|  |
| 三看  C++中的继承方式（public、private、protected）会影响子类的对外访问属性  判断某一句话，能否被访问  1）看调用语句，这句话写在子类的内部、外部  2）看子类如何从父类继承（public、private、protected）  3）看父类中的访问级别（public、private、protected） |
| 继承中的同名成员变量处理方法 |
| class A  {  public:  int a;  int b;  void printfB()  {  cout<<"我是父类的B"<<b<<endl;  }  void print()  {  cout<<"我是父类"<<endl;  }  protected:    private:  };  class B : public A  {  public:  int b;  int c;  void print()  {  cout<<"我是子类"<<endl;  }  protected:  private:  };  void main()  {    B b1;  b1.b = 10;  b1.A::b = 11;  b1.printfB();  b1.print();  b1.A ::print();  b1.B::print();  system("pause");  } |
| Static关键字遇上继承 |
| class A  {  public:  static int a;  int b;  void printfB()  {  cout<<"我是父类的B"<<b<<endl;  }  void print()  {  //cout<<"我是父类"<<endl;  //cout<<a<<endl;  }  protected:  private:  };  //这句话的意思是 给静态变量初始化0  //告诉C++编译器为整个类分配 4个字节的内存空间  int A::a = 0;  class B : private A  {  public:    int c;  void print()  {  //cout<<"我是子类"<<endl;  //cout<<a<<endl;  }  protected:  private:  };  void main()  {  //A a1;  B b1;  b1.print();    system("pause");  } |

## 继承中的构造和析构

|  |
| --- |
| **继承中的构造析构调用原则**  1、子类对象在创建时会首先调用父类的构造函数  2、父类构造函数执行结束后，执行子类的构造函数  3、当父类的构造函数有参数时，需要在子类的初始化列表中显示调用  4、析构函数调用的先后顺序与构造函数相反  **继承与组合混搭情况下，构造和析构调用原则**  原则： 先构造父类，再构造成员变量、最后构造自己  先析构自己，在析构成员变量、最后析构父类  //先构造的对象，后释放 |
| //继承中的构造和析构  //继承和组合混搭情况下，构造函数、析构函数调用顺序研究  #include <iostream>  using namespace std;  class Object  {  public:  Object(const char\* s)  {  cout<<"Object()"<<" "<<s<<endl;  }  ~Object()  {  cout<<"~Object()"<<endl;  }  };  class Parent : public Object  {  public:  Parent(const char\* s) : Object(s)  {  cout<<"Parent()"<<" "<<s<<endl;  }  ~Parent()  {  cout<<"~Parent()"<<endl;  }  };  class Child : public Parent  {  protected:  Object o1;  Object o2;  public:  Child() : o2("o2"), o1("o1"), Parent("Parameter from Child!")  {  cout<<"Child()"<<endl;  }  ~Child()  {  cout<<"~Child()"<<endl;  }  };  void run05()  {  Child child;  }  int main(int argc, char \*argv[])  {    run05();  system("pause");  return 0;  } |
|  |

## 赋值兼容性原则

|  |
| --- |
| **赋值兼容性原则**  子类对象可以当作父类对象使用  子类对象可以直接赋值给父类对象  子类对象可以直接初始化父类对象  父类指针可以直接指向子类对象  父类引用可以直接引用子类对象  总结：子类就是特殊的父类 (**base \*p = &child;)** |
|  |

多继承二义性及虚继承解决方案

|  |
| --- |
| 相当于图，问题赋值了。这种多继承带来的便利性远远小于软件维护、软件架构复杂性 |
| #include "iostream"  using namespace std;  class B  {  public:  int i;  protected:  private:  };  class B1 : virtual public B  {  public:  int b1;  protected:  private:  };  class B2 :virtual public B  {  public:  int b2;  protected:  private:  };  class C :public B1, public B2  {  public:  protected:  private:  };  void main()  {  C c1;  c1.b1 = 10;  c1.b2 = 20;  c1.i = 100;  system("pause");  } |

# 多态

## 问题引出（赋值兼容性原则遇上函数重写）

|  |
| --- |
| #include "iostream"  using namespace std;  //定义一个子类和一个父类  class Parent  {  public:  Parent(int a = 0)  {  this->a = a;  }  Virtual void print()  {  cout<<"父类a:"<<a<<endl;  }  protected:  private:  int a;  };  class Child : public Parent  {  public:  Child(int b = 0)  {  this->b = b;  }  void print()  {  cout<<"子类b:"<<b<<endl;  }  protected:  private:  int b;  };  //面向对象新需求  //如果我传一个父类对象，执行父类的print函数  //如果我传一个子类对象，执行子类的printf函数  //现象产生的原因  //赋值兼容性原则遇上函数重写 出现的一个现象  //1 没有理由报错  //2 对被调用函数来讲，  //3 在编译器编译期间，我就确定了，这个函数的参数是p，是Parent类型的。。。  //静态链编  void HowToPrint1(Parent \*p)  {  p->print(); //一句话，有多种效果，有多种表现形态把。。。//这个功能的就是多态  }  void HowToPrint2(Parent &myp)  {  myp.print();  }  void main()  {  Parent p1;  Child c1;  /\*  p1.print();  c1.print();    Parent \*p = NULL;  p = &p1;  p->print(); //调用父类的打印函数  ////赋值兼容性原则 遇上 同名函数的时候  p = &c1;  p->print(); //  Parent &myp = c1;  myp.print();  \*/  HowToPrint1(&p1);  HowToPrint1(&c1);  HowToPrint2(p1);  HowToPrint2(c1);  system("pause");  } |
| 多态的理解  角度1：  p->print(); //一句话，有多种效果，有多种表现形态把。。。//这个功能的就是多态  角度2： |
|  |

## 面向对象新需求

|  |
| --- |
| 编译器的做法不是我们期望的  根据实际的对象类型来判断重写函数的调用  如果父类指针指向的是父类对象则调用父类中定义的函数  如果父类指针指向的是子类对象则调用子类中定义的重写函数 |

## C++提供的多态解决方案

* **C++中的多态支持**
* **C++中通过virtual关键字对多态进行支持**
* **使用virtual声明的函数被重写后即可展现多态特性**

## 多态实例

|  |
| --- |
| #include "iostream"  using namespace std;  class HeroFighter  {  public:  virtual int Power()  {  return 10;  }  };  class AdvHeroFighter : public HeroFighter  {  public:  int Power()  {  return 20;  }  protected:  private:  };  class Adv2HeroFighter : public HeroFighter  {  public:  int Power()  {  return 30;  }  protected:  private:  };  class EnemyFighter  {  public:  int attack()  {  return 15;  }  protected:  private:  };  void main01()  {  HeroFighter hf;  EnemyFighter ef;  AdvHeroFighter advHf;    if (hf.Power() < ef.attack())  {  cout<<"英雄挂了。。。"<<endl;  }  else  {  cout<<"英雄win。。。"<<endl;  }  if (advHf.Power() < ef.attack())  {  cout<<"英雄2挂了。。。"<<endl;  }  else  {  cout<<"英雄2win。。。"<<endl;  }  system("pause");  }  //写了一个框架，可以调用  //我的第3代战机代码出现的时间晚于框架出现的时间。。。。  //框架 有使用后来人 写的代码的能力。。。  //面向对象3大概念  /\*  封装  突破了C语言函数的概念。。  继承  代码复用 。。。。我复用原来写好的代码。。。  多态  多态可以使用未来。。。。。80年代写了一个框架。。。。。。90人写的代码  多态是我们软件行业追寻的一个目标。。。  ////  \*/  //间接赋值成立的3个条件  //1 定义两个变量。。。  //2 建立关联 。。。。  //3 \*p  //多态成立的三个条件  //1 要有继承  //2 要有函数重写。。。C 虚函数  //3 要有父类指针（父类引用）指向子类对象  //  //AOP  //相当于你的框架把。。。。  void ObjPlay(HeroFighter \*pBase, EnemyFighter \*peEf)  {  //多态的存在  if (pBase->Power() < peEf->attack())  {  cout<<"英雄挂了。。。"<<endl;  }  else  {  cout<<"英雄win。。。"<<endl;  }  }  void main()  {  HeroFighter hf;  EnemyFighter ef;  AdvHeroFighter advHf;  Adv2HeroFighter adv3Hf;  ObjPlay(&hf, &ef);  ObjPlay(&advHf, &ef);  ObjPlay(&adv3Hf, &ef);  } |

## 多态工程意义及多态成立的三个条件

|  |
| --- |
| 1. 工程中大量应用，可以做框架 2. 可以架构上，解耦合。。。。   //间接赋值成立的3个条件  //1 定义两个变量。。。  //2 建立关联 。。。。  //3 \*p  //多态成立的三个条件  //1 要有继承  //2 要有函数重写。。。C 虚函数  //3 要有父类指针（父类引用）指向子类对象  // |
|  |

## 重写 PK 重载理解

|  |
| --- |
| 函数重载  必须在同一个类中进行  子类无法重载父类的函数，父类同名函数将被名称覆盖  重载是在编译期间根据参数类型和个数决定函数调用  函数重写  必须发生于父类与子类之间  并且父类与子类中的函数必须有完全相同的原型  使用virtual声明之后能够产生多态(如果不使用virtual，那叫重定义)  多态是在运行期间根据具体对象的类型决定函数调用 |

## 请谈谈你对多态的理解

|  |
| --- |
| **多态的实现效果**  多态：同样的调用语句有多种不同的表现形态；  **多态实现的三个条件**  有继承、有virtual重写、有父类指针（引用）指向子类对象。  **多态的C++实现**  virtual关键字，告诉编译器这个函数要支持多态；不要根据指针类型判断如何调用；而是要根据指针所指向的实际对象类型来判断如何调用 **多态的理论基础**  动态联编PK静态联编。根据实际的对象类型来判断重写函数的调用。  **多态的重要意义**  设计模式的基础。  **实现多态的理论基础**  函数指针做函数参数 |

## 多态原理探究（证明VPTR指针的存在）

## 虚函数表指针（VPTR）被编译器初始化的过程

构造函数中调用虚函数能实现多态吗

|  |
| --- |
|  |
|  |

## 为什么要定义虚析构函数

|  |
| --- |
| //在父类中声明虚析构函数的原因  //通过父类指针，把所有的子类析构函数都执行一遍。。。  //  void howtoDel(Parent \*pbase)  {  delete pbase;  }  void mainobj()  {  Parent \*p1 = new Parent();  p1->print();  delete p1;  }  void main()  {  Child \*pc1 = new Child();  howtoDel(pc1);  //mainobj();  system("pause");  } |

## 基类和子类对象指针++混搭风

|  |
| --- |
| #include "iostream"  using namespace std;  class Parent01  {  protected:  int i;  int j;  public:  virtual void f()  {  cout<<"Parent01::f"<<endl;  }  };  //一次偶然的成功，比必然的失败更可怕  class Child01 : public Parent01  {  public:  int k;  public:  Child01(int i, int j)  {  printf("Child01:...do\n");  }  virtual void f()  {  printf("Child01::f()...do\n");  }  };  void howToF(Parent01 \*pBase)  {  pBase->f();    }  int main()  {  int i = 0;  Parent01\* p = NULL;  Child01\* c = NULL;  //可以使用赋值兼容性原则，是用在多态的地方  //不要轻易通过父类指针p++,来执行函数操作  //问题的本质 子类指针 和父类指针 步长可能不一样。。。  Child01 ca[3] = {Child01(1, 2), Child01(3, 4), Child01(5, 6)};  p = ca; //第一个子类对象赋值给p，p是基类指针，  c = ca;  p->f(); //有多态发生  //c->f(); //  p++;  //c++;    p->f();//有多态发生  //c->f();  // for (i=0; i<3; i++)  // {  // howToF(&(ca[i]));  // }  system("pause");  return 0;  } |

# 纯虚函数抽象类（接口类）

## 语法基础

|  |
| --- |
| #include "iostream"  using namespace std;  //抽象一个类  //抽象类  class figure  {  public:  virtual int showarea() = 0; //纯虚函数  //int showarea2() = 0;  //int showarea3() = 0;  };  class Tri : public figure  {  public:  Tri(int a = 0, int b = 0)  {  this->a = a;  this->b = b;  }  virtual int showarea()  {  cout<<"三角形s"<<a\*b/2<<endl;  return 0;  }  protected:  private:  int a;  int b;  };  class Squre : public figure  {  public:  Squre(int a = 0, int b = 0)  {  this->a = a;  this->b = b;  }  virtual int showarea()  {  cout<<"四边形s"<<a\*b<<endl;  return 0;  }  protected:  private:  int a;  int b;  };  //通过抽象类，定义一套接口。。让别人实现你的抽象类的接口定义；。。。  //框架就可以在不做任何改变的情况下，调用后来人写的子类（抽象类的子类）。。  void printS(figure \*pbase)  {  pbase->showarea();  }  void main81()  {    //figure f;  figure \*pbase = NULL;  Tri tri(10, 2);  tri.showarea();  Squre sq(3, 4);  sq.showarea();  system("pause");  }  void main()  {  //figure f;  figure \*pbase = NULL;  Tri tri(10, 2);  Squre sq(3, 4);  printS(&tri);  printS(&sq);  system("pause");  } |
|  |

## 工程实践

观看例子：mysocket工程实践.rar

## c++中接口的封装和设计和模块界面分工

# 函数指针专题讲座

## 函数指针语法基础

|  |
| --- |
| /定义函数类型  //定义函数指针类型  //定义一个指针函数的指针变量  //test是函数名，函数名代表函数首地址，函数名就是函数指针  int test(int a)  {  printf("a:%d", a);  return 0;  }  void main()  {  //定义一个函数类型。。。  typedef int (MYFuncType)(int);      MYFuncType \*myfun; //函数指针变量  test(1); //指向一个函数调用  {  myfun = test;  //通过函数指针变量，可以找到函数入口，可以执行函数  myfun(2);  }  {  //对函数名字&地址，和不取地址，效果一样。。。。  //历史遗留原因  myfun = &test;  myfun(3);  }  {  int ret = 0;  //定义了一个函数指针类型  typedef int (\*PMYFuncType)(int); //int(\*my)(int , int, char \*) = NULL:  PMYFuncType pMyFuncType;  pMyFuncType = test;  ret = pMyFuncType(1);  }  {  //直接定义一个函数指针变量。。。。  int (\*PmyFunc)(int) ; //告诉编译器分配4个字节  //\*\*\*\* 没有意义  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PmyFunc = test;  PmyFunc(11);  }  //printf("myFunc:%d", sizeof(myFunc));  system("pause");  } |

|  |
| --- |
| //子任务写任务的人  int add(int a, int b)  {  int c ;  c = a + b;  printf("func add:%d \n", c);  return c;  }  int add2(int a, int b)  {  int c ;  c = a + b;  printf("func add:%d \n", c);  return c;  }  int add3(int a, int b)  {  int c ;  c = a + b;  printf("func add:%d \n", c);  return c;  }  //调用任务的人  int MyOP(int (\*MyAdd)(int, int))  {  //执行add函数调用有2种方法  add(1, 2);  MyAdd(3, 4);  return 0;  }  //我在声明一个类型，申明一个函数指针类型。。。  typedef int(\*PMyFuncType)(int , int);  //函数指针做函数参数的本质。  //1）不光是把函数入口地址给传送给被调用函数。。  //2）同时，函数类型，也做了一个接口的约定。。。。。。（任务类型的约定）  //反映到代码上。。。函数三要素 （参数、返回值）统统的做了一个约定。。。。  //这个就函数指针做函数参数的核心思想。。。  //对比：C++  //  int MyOP2(PMyFuncType pMyFuncType)  {  //执行add函数调用有2种方法  add(1, 2);  pMyFuncType(3, 4);  return 0;  }  //调用任务的人，，分开了。。。。。。  void main()  {  //定义一个指针变量，函数指针变量  int (\*pAdd)(int , int);  //把函数的入口地址复制给。。。。pAdd  pAdd = add;  //调用MyOP，要求传送一个函数指针类型。。的变量。。。  //MyOP(pAdd);  MyOP2(add3);  system("pause");  } |

## 正向调用

|  |
| --- |
| //客户端初始化 获取handle上下  typedef int (\*CltSocketInit)(void \*\*handle /\*out\*/);  typedef int (\*CltSocketSend)(void \*handle /\*in\*/, unsigned char \*buf /\*in\*/, int buflen /\*in\*/);  //客户端收报文  typedef int (\*CltSocketRev)(void \*handle /\*in\*/, unsigned char \*buf /\*in\*/, int \*buflen /\*in out\*/);  typedef int (\*CltSocketDestory)(void \*handle/\*in\*/);  void C动态库函数调用Dlg::OnBnClickedButton1()  {  // TODO: 在此添加控件通知处理程序代码  int ret = 0;  void \*handle = NULL;  HINSTANCE hInstance;  hInstance = LoadLibrary("c:/socketclient.dll");  if (hInstance == NULL)  {  ;  }  CltSocketInit cltSocketInit = (CltSocketInit)GetProcAddress(hInstance, "cltSocketInit");  if (cltSocketInit == NULL)  {  return ;  }  CltSocketSend cltSocketSend = (CltSocketSend)GetProcAddress(hInstance,"cltSocketSend");  if (cltSocketInit == NULL)  {  return ;  }  CltSocketRev cltSocketRev = (CltSocketRev)GetProcAddress(hInstance,"cltSocketRev");  if (cltSocketInit == NULL)  {  return ;  }  CltSocketDestory cltSocketDestory = (CltSocketDestory)GetProcAddress(hInstance, "cltSocketDestory");  if (cltSocketInit == NULL)  {  return ;  }    char buf[1024] = {0};  strcpy(buf, "dddddddddddsssssssssssssssssssss");  int buflen = 10;  char outbuf[1000] = {0};  int outbuflen;  ret = cltSocketInit(&handle);  if (ret != 0)  {  return ;  }  ret = cltSocketSend(handle, (unsigned char \*)buf, buflen );  ret = cltSocketRev(handle, (unsigned char \*)outbuf, &outbuflen);  ret = cltSocketDestory(handle);  } |

## 反向调用

|  |
| --- |
| 利用函数指针实现的一种调用机制 |
| 具体任务的实现者，可以不知道什么时候被调用 |
|  |
| 回调函数是利用函数指针实现的一种调用机制  回调机制原理  当具体事件发生时，调用者通过函数指针调用具体函数  回调机制的将调用者和被调函数分开，两者互不依赖  任务的实现 和 任务的调用 可以耦合 （提前进行接口的封装和设计） |
| 需要重新审视 dll和其他第三方厂商 接口之间的关系  动态库可以集成第三方厂商的业务功能  动态库（框架）不经常改动，能兼容后来的业务模型 2010 能调用 2020写 多态  动态库和第三方业务模型 模块之间 松耦合 |
| //动态库变成框架 需要做的工作  //1 第三方业务入口传进来（回调函数的入口地址传进来）  //2 加密 解密 业务模型抽象  //实现 动态库 加密解密业务模型抽象  通过定义函数指针类型；函数指针做函数参数实现 |
|  |

# 泛型编程

## 语法基础

|  |
| --- |
| //template关键字告诉c++编译器，现在我要进行泛型编程，  //typename 告诉c++编译器，T为数据类型，，，请你不要乱报错。。。。。  //T为数据类型，T为数据类型参数化，而已。。。。。  //int char double  template<typename T>  void myswap3(T &a, T &b)  {  T t = a;  a = b;  b = t;  };  void main()  {  //泛型编程函数的使用方法有2种  int x = 1;  int y = 2;  //自动类型推导，，  //myswap3(x, y);  //具体类型调用  myswap3<int>(x, y);  char c1 = 'a';  char c2 = 'b';  myswap3<char>(c1, c2);  system("pause");  } |
| /\*  1 函数模板可以像普通函数一样被重载  2 C++编译器优先考虑普通函数  3 如果函数模板可以产生一个更好的匹配，那么选择模板  4 可以通过空模板实参列表的语法限定编译器只通过模板匹配  \*/  /\*  函数模板不允许自动类型转化  普通函数能够进行自动类型转换  \*/  #include<stdio.h>  int main()  {    return  }  #include "iostream"  using namespace std;  void main()  {    system("pause");  }  #include <iostream>  using namespace std;  int Max(int a, int b)  {  cout<<"int Max(int a, int b)"<<endl;  return a > b ? a : b;  }  template<typename T>  T Max(T a, T b)  {  cout<<"T Max(T a, T b)"<<endl;  return a > b ? a : b;  }  template<typename T>  T Max(T a, T b, T c)  {  cout<<"T Max(T a, T b, T c)"<<endl;  return Max(Max(a, b), c);  }  void main()  {  int a = 1;  int b = 2;  cout<<Max(a, b)<<endl;  cout<<Max<>(a, b)<<endl;  cout<<Max(3.0, 4.0)<<endl;  cout<<Max(5.0, 6.0, 7.0)<<endl;  cout<<Max('a', 100)<<endl;  system("pause");  return ;  } |
| 类模板语法基础 |
| //类模板的创建  //类中的属性，，类型参数化  //我们把数据类型，进一步 抽象化。。。  //抽象化的数据类型，不能直接使用。。  template<typename T>  class A  {  public:  //注意函数函数声明和函数实现的区别  A(int a) //函数申明  {  this->a = a;  }  ~A()  {  }  T getA()  {  return a;  }  void setA(T a)  {  this->a = a;  }  protected:  private:  T a;  }; |

## 类模板遇上友元函数

参考项目：类模板的友元函数优化.rar

## 类模板遇上static

每一个具体的类的类型，是一个static

# STL编程入门

vector

|  |
| --- |
| / vector<int> &c 和数组做函数参数区别  //数组做函数 会退化为指针  //类，类有封装概念，突破了函数的概念。。。（封装了属性和方法）  void printfV(vector<int> &c)  {  int size = c.size();  for (int i=0; i<size; i++)  {  printf("%d ", c[i]);  }  }  void main01()  {  //vector是一个模板类。。在使用模板类的时候，需要指明具体的类型  //  vector<int> v1(5); //相当于 int v1[5];  //vector<char> v1(5); //相当于 int v1[5];  for (int i=0; i<5; i++)  {  v1[i] = i+1;    }  for (int i=0; i<5; i++)  {  cout<<v1[i];  }  vector<int>v2(20);  //执行=号操作  v2 = v1;  printfV(v2);  //vector会把20个大小全部初始化。。。。  vector<int>v3(20);  v3.push\_back(100);  v3.push\_back(101);  printfV(v3);  system("pause");  }  struct Teacher  {  char name[10];  int age;  };  void main02()  {  //vector是一个模板类。。在使用模板类的时候，需要指明具体的类型  //  Teacher t1, t2, t3;  t1.age = 11;  t2.age = 22;  t3.age = 33;  vector<Teacher> v1(5); //相当于 int v1[5];  v1[0] = t1;  v1[1] = t2;  v1[2] = t3;  printf("age:%d \n", v1[0].age);  system("pause");  }  void main()  {  //vector是一个模板类。。在使用模板类的时候，需要指明具体的类型  //  Teacher t1, t2, t3;  t1.age = 11;  t2.age = 22;  t3.age = 33;  vector<Teacher \*> v1(5); //相当于 int v1[5];  v1[0] = &t1;  v1[1] = &t2;  v1[2] = &t3;  for (int i=0; i<v1.size(); i++)  {  Teacher \*tmp = v1[i];  if (tmp != NULL)  {  printf("age:%d", tmp->age);  }  }  system("pause");  } |
| void main()  {  //定义了容器类，具体类型int  stack<Teacher \*> s;  Teacher t1, t2, t3;  t1.age = 31;  t2.age = 32;  t3.age = 33;  s.push(&t1);  s.push(&t2);  s.push(&t3);  //遍历栈的所有元素，必须要一个一个的弹出元素  while(!s.empty())  {  //获取栈顶元素  Teacher \*tmp = s.top();  //弹出栈元素  s.pop();  printf("tmp:%d ", tmp->age);  }  printf("\n");  //printStack(s);  system("pause");  } |
|  |
| struct Teacher  {  char name[64];  int age ;  };  void printQ(queue<Teacher \*> &myq)  {  while (!myq.empty())  {  Teacher \*tmp = myq.front();  if (tmp != NULL)  {  printf("tmp:age : %d ", tmp->age);  }  myq.pop();  }  }  void main()  {  queue<Teacher \*> q;  Teacher t1, t2, t3;  t1.age = 32;  t2.age = 33;  t3.age = 34;  q.push(&t1);  q.push(&t2);  q.push(&t3);  printQ(q);  system("pause");  } |
|  |
| #include "iostream"  #include "list"  using namespace std;  void main01()  {  list<int> l;  cout<<l.size()<<endl;  for (int i=0; i<5; i++)  {  l.push\_back(i+1);  }  cout<<l.size()<<endl;    list<int>::iterator current = l.begin();  while (current != l.end())  {  cout<<\*current<<endl;  current ++;  }  //获取链表的开头，赋值给迭代器  current = l.begin(); //0  current ++; //1  current ++; //2  current ++; //3  l.insert(current, 100);  printf("\n插入位置测试，请你判断是从0位置开始，还是从1位置开始\n");  for (list<int>::iterator p = l.begin(); p!= l.end(); p++)  {  cout<<\*p<<" ";    }  system("pause");  }  struct Teacher  {  char name[64];  int age;  };  void main02()  {  list<Teacher> l;  Teacher t1, t2, t3;  t1.age = 11;  t2.age = 12;  t3.age = 13;  l.push\_back(t1);  l.push\_back(t2);  l.push\_back(t3);  for (list<Teacher>::iterator p = l.begin(); p!=l.end();p++)  {  printf("tmp:%d ", p->age);  }  system("pause");  }  void main()  {  list<Teacher \*> l;  Teacher t1, t2, t3;  t1.age = 11;  t2.age = 12;  t3.age = 13;  l.push\_back(&t1);  l.push\_back(&t2);  l.push\_back(&t3);  for (list<Teacher \*>::iterator p = l.begin(); p!=l.end();p++)  {  Teacher \* tmp = \*p;  printf("tmp:%d ", tmp->age);  }  system("pause");  } |
|  |
| #include "iostream"  #include "vector"  #include "cmath"  #include "algorithm"  using namespace std;  /\*  void sort(\_RanIt \_First, \_RanIt \_Last, \_Pr \_Pred)  void \_Sort(\_RanIt \_First, \_RanIt \_Last, \_Diff \_Ideal, \_Pr \_Pred)  void \_Insertion\_sort(\_BidIt \_First, \_BidIt \_Last, \_Pr \_Pred)  void \_Insertion\_sort1(\_BidIt \_First, \_BidIt \_Last, \_Pr \_Pred, \_Ty \*)  \_Debug\_lt\_pred \_DEBUG\_LT\_PRED(\_Pred, \_Val, \*\_First))  \_Pred（比较函数）  \*/  void myPrintFunc(int &v)  {  cout<<v<<endl;  }  int mycompare(const int &a, const int &b)  {  return a<b;  }  struct Teacher  {  char name[100];  int age;  };  void main()  {  vector<int> v(10);  for (int i=0; i<10; i++)  {  v[i] = rand() %10;  }  for (int i=0; i<10; i++)  {  printf("%d ", v[i]);  }  for\_each(v.begin(), v.end(), myPrintFunc);  //stl对数组进行排序的时候，stl实现了排序算法，但是排序的标准，可以有用户自己来定义  sort(v.begin(),v.end(),mycompare);  for (int i=0; i<10; i++)  {  printf("%d ", v[i]);  }  system("pause");  } |