用C语言实现程序的多态性 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/646/2021_2022__E7_94_A8C_E8_AF_AD_E8_A8_80_c97_646592.htm 导读:使用面向对象的语言可以实现多态，并且在很大程度上降低了代码的复杂性。对于面向过程的 C 语言同样可以实现多态，本文将着重介绍 C 语言是如何实现多态的。 前言：关于多态，关于 C多态 (polymorphism) 一词最初来源于希腊语 polumorphos，含义是具有多种形式或形态的情形。在程序设计领域，一个广泛认可的定义是“ 一种将不同的特殊行为和单个泛化记号相关联的能力 ”。然而在人们的直观感觉中，多态的含义大约等同于“ 同一个方法对于不同类型的输入参数均能做出正确的处理过程，并给出人们所期望获得的结果 ”，也许这正体现了人们对于多态性所能达到的效果所寄予的期望：使程序能够做到越来越智能化，越来越易于使用，越来越能够使设计者透过形形色色的表象看到代码所要触及到的问题本质。 作为读者的你或许对于面向对象编程已有着精深的见解，或许对于多态的方便与神奇你也有了深入的认识。这时候你讶异的开始质疑了：“ 多态，那是面向对象编程才有的技术，C 语言是面向过程的啊！ ”而我想说的是，C 语言作为一种编程语言，也许并不是为了面向对象编程而设计，但这并不意味着它不能实现面向对象编程所能实现的功能，就比如说，多态性。 在本文中我们使用一个简单的单链表作为例子，展示 C 语言是如何体现多态性的。 结构体：不得不说的故事 许多从写 C 代码开始，逐渐走向 C 的程序员都知道，其实 C 里面的 class，其前身正是 C 语言中的 structure。很多基于 C 语言

背景介绍 C 的书籍，在介绍到 class 这一章的时候都会向读者清晰地展示，一个 C 语言里的 structure 是怎样逐渐变成一个典型的 C class 的，甚至最后得出结论：“ structure 就是一个所有成员都公有的类 ”，当然了，class 还是 class，不能简单的把它看做一个复杂化了的 structure 而已。 下面我们来看看在 C 语言中定义一个简单的存储整型数据的单链表节点是怎么做的，当然是用结构体。大部分人会像我一样，在 linkList.h 文件里定义： typedef struct Node* linkList. struct Node // 链表节点 { int data. // 存储的整型数据 linkList next. // 指向下一个链表节点 }. 链表有了，下面就是你想要实现的一些链表的功能，当然是定义成函数。我们只举几个常用功能： linkList initialLinklist(). // 初始化链表 link newLinkList (int data). // 建立新节点 void insertFirst(linkList h,int data). // 在已有链表的表头进行插入节点操作 void linkListOutput(linkList h). // 输出链表中数据到控制台 这些都是再自然不过的 C 语言的编程过程，然后我们就可以在 linkList.c 文件中实现上述两个函数，继而在 main.c 中调用它们了。 然而上面我们定义的链表还只能对整型数据进行操作。如果下次你要用到一个存储字符串类型的链表，就只好把上面的过程重新来过。也许你觉得这个在原有代码基础上做略微修改的过程并不复杂，可是也许我们会不断的增加对于链表这个数据结构的操作，而需要用链表来存储的数据类型也越来越多，这些都意味着海量的代码和繁琐的后期维护工作。当你有了上百个存储不同数据类型的链表结构，每当你要增加一个操作，或者修改某个操作的传入参数，工作量会变大到像一场灾难。 但是我们可以改造上述代码，让它能够处理你所想要让它处理的任何数据类

型：实行，字符型，乃至任何你自己定义的 structure 类型。Void*：万能的指针“ 挂钩 ”几乎所有讲授 C 语言课程的老师都会告诉你：“ 指针是整个 C 语言的精髓所在。 ”而你也一直敬畏着指针，又爱又恨地使用着它。许多教材都告诉你，int * 叫做指向整型的指针，而 char * 是指向字符型的指针，等等不一而足。然而这里有一个另类的指针家族成员 void *。不要按照通常的命名方式叫它做指向 void 类型的指针，它的正式的名字叫做：可以指向任意类型的指针。你一定注意到了“ 任意类型 ”这四个字，没错，实现多态，我们靠的就是它。 下面来改造我们的链表代码，在 linkList.h 里，如下：typedef struct Node* linkList. struct Node // 链表节点 { void *data. // 存储的数据指针 linkList next. // 指向下一个链表节点 }. linkList initialLinklist(). // 初始化链表 link newLinkList (void *data). // 建立新节点 void insertFirst(linkList h, void *data). // 在已有链表的表头进行插入节点操作 void linkListOutput(linkList h). // 输出链表中数据到控制台 我们来看看现在这个链表和刚才那个只能存储整型数据的链表的区别。 当你把 Node 结构体里面的成员定义为一个整型数据，就好像把这个链表节点打造成了一个大小形状固定的盒子，你定义一个链表节点，程序进行编译的时候编译器就为你打造一个这样的盒子：装一个 int 类型的数据，然后装一个 linkList 类型的指针。如果你想强行在这个盒子里装别的东西，编译器会告诉你，对不起，盒子的大小形状并不合适。所以你必须为了装各种各样类型的数据打造出不同的生产盒子的流水线，想要装哪种类型数据的盒子，就开启对应的流水线来生产。 但是当你把结构体成员定义为 void *，一切都变得不同了。这时的链表节点

不再像个大小形状固定的盒子，而更像一个挂钩，它可以挂上一个任意类型的数据。不管你需要存储什么类型的数据，你只要传递一个指针，把它存储到 Node 节点中去，就相当于把这个数据“ 挂 ”了上去，无论何时你都可以根据指针找到它。这时的链表仿佛变成了一排粘贴在墙上的衣帽钩，你可以挂一排大衣，可以挂一排帽子，可以挂一排围巾，甚至，你可以并排挂一件大衣一顶帽子一条围巾在墙上。void * 初露狰狞，多态离 C 语言并不遥远。 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com