嵌入式软件工程方法与实践丛书

程序设计与数据结构

周立功　周攀峰　编著

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

嵌入式软件工程方法与实践丛书

程序设计与数据结构

周立功　周攀峰　编著

北京航空航天大学出版社

内 容 简 介

本书是C语言程序设计的进阶书籍，在介绍C语言基础知识的同时，重点强调了软件设计的思想：共性与可变性分析、面向对象的编程思想等，并提供了详尽的范例程序；使读者体会到思想的重要性，面向对象编程并不局限于特定语言，使用C语言同样可以进行面向对象的编程。本书分为4章：第1章，主要介绍C语言的基础知识，并提及了共性与可变性分析；第2章，主要介绍C语言的进阶用法，特别是结构体及函数指针；第3章，主要介绍算法与数据结构，包含链表、哈希表、队列等；第4章，主要介绍面向对象的编程思想，包含面向对象的基础概念、虚函数的妙用、状态机设计等。

本书既可作为高等院校、高职高专电子信息工程、自动化、机电一体化及计算机专业的教材，也可作为电子及计算机编程爱好者的自学用书，还可作为软件开发工程技术人员的参考书。

图书在版编目(CIP)数据

程序设计与数据结构 / 周立功，周攀峰编著. -- 北京 ：北京航空航天大学出版社，2018.11

ISBN 978-7-5124-2870-6

Ⅰ. ①程… Ⅱ. ①周… ②周… Ⅲ. ①程序设计 ②数据结构 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2018)第 257823 号

程序设计与数据结构

周立功　周攀峰　编著

责任编辑　王慕冰

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(邮编 100191)　http：//www.buaapress.com.cn

发行部电话：(010)82317024　传真：(010)82328026

读者信箱：emsbook@buaacm.com.cn　邮购电话：(010)82316936

涿州市新华印刷有限公司印装　各地书店经销

*

开本：710×1 000　1/16　印张：20.5　字数：437 千字

2018 年 11 月第 1 版　2018 年 11 月第 1 次印刷　印数：3 000 册

ISBN 978-7-5124-2870-6　定价：59.00 元

前　言

1. 存在的问题

最近十多年来，软件产业和互联网产业的迅猛发展，给人们提供了用武之地，同时也对软件工程教育提出了巨大的挑战。从教育现状看，通过灌输知识可以让人具有很强的考试能力，却往往经不起用人单位的检验（笔试和机试）。虽然大家都知道，教育的本质在于培养人的创造力、好奇心、独特的思考能力和解决问题的能力，但实际上我们的教学实践背离了教育理念。

代码的优劣不仅直接决定了软件的质量，还将直接影响软件成本。软件成本是由开发成本和维护成本组成的，但维护成本却远高于开发成本，蛮力开发的现象比比皆是，大量来之不易的资金就这样被无声无息地吞没了，造成了社会资源的严重浪费。

2. 核心域和非核心域

一个软件系统封装了若干领域的知识，其中一个领域的知识代表了系统的核心竞争力，这个领域就称为“核心域”，其他领域称为“非核心域”。虽然更通俗的说法是“业务”和“技术”，但使用“核心域”和“非核心域”更严谨。

非核心域就是别人的领域，如底层驱动、操作系统和组件，即便自己有一些优势，那也是暂时的，竞争对手也能通过其他渠道获得。非核心域的改进是必要的，但不充分，还是要在核心域上深入挖掘，让竞争对手无法轻易从第三方获得。只有在核心域中深入挖掘，达到基于核心域的复用，才能获得和保持竞争力。

要达到基于核心域的复用，有必要将核心域和非核心域分开考虑。因为过早地将各个领域的知识混杂，会增加不必要的负担，从而导致开发人员腾不出脑力思考核心域中更深刻的问题，待解决问题的规模一旦变大，而人脑的容量和运算能力又是有限的，就会造成顾此失彼，因此必须分而治之。核心域与非核心域的知识都是独立的，比如，一个计算器要做到没有漏洞，其中的问题也很复杂。如果不使用状态图对

领域逻辑显式地建模，再根据模型映射到实现，而是直接下手编程，领域逻辑的知识靠临时去想，最终得到的代码肯定破绽百出。其实有利润的系统，其内部都是很复杂的，千万不要幼稚地认为“我的系统不复杂”。

3. 利润从哪里来

早期创业时，只要抓住一个机会，多参加展会，多做广告，成功的概率就很大了。在互联网时代，突然发现入口多了，聚焦用户的难度也越来越大了。当产品面临竞争时，你会发现“没有最低只有更低”。而且现在已经没有互联网公司了，携程变成了旅行社，新浪变成了新媒体……机会驱动、粗放经营的时代已经过去了。

Apple 之所以能成为全球最赚钱的手机公司，关键在于产品的性能超越了用户的预期，且因为大量可重用的核心领域知识，综合成本做到了极致。Yourdon 和 Constantine 在《结构化设计》一书中，将经济学作为软件设计的底层驱动力，软件设计应该致力于降低整体成本。人们发现软件的维护成本远远高于它的初始成本，因为理解现有代码需要花费时间，而且容易出错。同时改动之后，还要进行测试和部署。

更多的时候，程序员不是在编码，而是在阅读程序。由于阅读程序需要从细节和概念上理解，因此修改程序的投入会远远大于最初编程的投入。基于这样的共识，让我们操心的一系列事情，需要不断地思考和总结使之可以重用，这就是方法论的缘起。

通过财务数据分析可知，由于决策失误，我们开发了一些周期长、技术难度大且回报率极低的产品。由于缺乏科学的软件工程方法，不仅软件难以重用，而且扩展和维护难度很大，从而导致开发成本居高不下。

显而易见，从软件开发来看，软件工程与计算机科学是完全不同的两个领域的知识。其主要区别在于人，因为软件开发是以人为中心的过程。如果考虑人的因素，软件工程更接近经济学，而非计算机科学。如果不改变思维方式，则很难开发出既好卖成本又低的产品。

4. 优秀人才在哪里

学徒模式是过去造就大师、传承技艺的方法，而现在指导和辅导却成为了一项被忽略的活动，团队成员得不到所需的支持。技术领导人不仅要引导整个项目，而且还要为员工提供必需的协助。除此之外，指导和辅导提供了一种增强员工技能的方式，可帮助他们完善自己的职业生涯。这种协助有时是技术性的，有时是软技能的。

可惜的是，在我们的行业里，许多优秀的开发者在转向管理岗位之后，就放弃了对技术的追求，甚至再也不写代码了，因而团队中失去了最有价值的技术领导和导师，导致今天的开发者还会继续重蹈覆辙。很多优秀的导师都消失了，让开发者到哪里去获得经验呢？未来的优秀人才从哪里来？

5. 告知读者

这本书如同培训讲师的教案，是我和同事们的读书笔记及程序设计实践的心得，并不是一本从零开始编写的专著或图书。其中的很多内容并非我们原创，而是重用了一些公开出版物的内容，详见本书的参考文献。

6. 丛书简介

这套丛书命名为《嵌入式软件工程方法与实践丛书》，目前已经完成《程序设计与数据结构》、《面向 AMetal 框架和接口的 C 编程》和《面向 AWorks 框架和接口的 C 编程(上)》，后续还将推出《面向 AWorks 框架和接口的 C 编程(下)》、《面向 AMetal 框架和接口的 LoRa 编程》、《面向 AWorks 框架和接口的 C++编程》、《面向 AWorks 框架和接口的 GUI 编程》、《面向 AWorks 框架和接口的 CAN 编程》、《面向 AWorks 框架和接口的网络编程》、《面向 AWorks 框架和接口的 EtherCAT 编程》和《嵌入式系统应用设计》等系列图书，最新动态详见 www.zlg.cn(致远电子官网)和 www.zlgmcu.com(周立功单片机官网)。

周立功

2018 年 9 月 20 日

目 录

第 1 章

程序设计基础

本章导读

在学习程序设计时，很多初学者常常会陷入这样的误区，他们总将阻碍个人成长的原因归结为缺少机会。其实问题的根源在于缺乏方法，很少有人将“知其然知其所以然”作为自己的学习准则，进而也就谈不上熟练掌握多种编程风格了。

其实，程序设计中的数据结构和算法是围绕各种类型的数据和需求展开的，而完成这些工作的载体便是各种各样的变量，因此只要抓住变量的三要素（即变量的类型、变量的值和变量的地址）并贯穿始终，那么一切问题都将迎刃而解。

1.1 思想的力量

《思想者》是法国雕塑家罗丹创作的雕像，他更多的是在强调其核心的内涵——思想，人类的整合思想。在 20 世纪初，人们把它作为一种改造世界力量的象征。显而易见，思想的力量是伟大而无穷的，无论是做大事还是做小事，无不与思想息息相关。

1.1.1 过程主题

1. 限制与抽象化

结构化编程的“限制”和“抽象化”是人类处理复杂软件的有效方法之一，为了使程序变得简单且容易理解，Edsger Dijkstra 提倡禁止使用 goto，并将程序控制流程限制为“顺序、分支和循环”三种组合。虽然面向结构的编程实现了控制流程的结构化，使程序流程结构化了，但要处理的数据并没有结构化。虽然面向过程的编程降低了程序的复杂性，但随着数据的类型越来越多，分别管理程序处理内容和处理数据对象所带来的程序也越来越复杂。

当需要将一部分计算任务独立实现时，可以将其定义为一个函数，因为这样可以实现计算逻辑的分离，通过使用函数名使代码更清晰，利用函数使得同样的代码在程序中可以多次使用，且减少调试程序的工作量。

由于实际的应用程序中可能会用到成千上万个函数，为了得到正确的结果，必须保持处理和数据的一贯性，人们想到了数据抽象技术。数据抽象是数据和处理方法的结合，对数据的处理和操作，必须通过事先定义好的方法进行。于是面向过程编程引入了较为抽象的模块的概念，因此可以说程序是由模块构成的，而模块又是由函数构成的，一个模块就是一个过程。由于不同结构中的数据是由函数或过程管理的，因此在设计程序时就可以对这些模块分别进行抽象、设计、编码和测试，最后将这些模块有机地组合在一起，形成一个完整的程序。

2. 功能分解法

通常解决复杂问题的方法都是从分析问题开始的，将一个大问题分解为多个小问题，分成多个子模块，解决每个小问题，实现每个子模块；通过主函数按照某种次序调用这些子模块，组织业务逻辑流程，最终解决问题。像这样从问题出发，自顶向下逐步求精，利用算法作为基本构建块构建复杂系统的开发方法称为结构化或面向过程编程。

怎样编写更容易应对需求多变的代码呢？与其编写一个大函数，不如使之更加模块化，即用模块化封装变化。虽然模块化有助于提高代码的可理解性，使代码更容易维护，但模块化也无法包治百病，因为模块化存在两个问题，即低内聚和高耦合。

假设要给 main 中调用的每个子过程增加一个参数，以便传递某个额外的信息。同时每个子过程又要将这个信息传递给自己的子过程，这种现象就是我们熟知的串联改变。所谓串联改变，是指某个过程的变化会传递到其子过程中，并由这些子过程继续往下延续，直到所有的分解层次。显然，在面对软件维护时，包括软件测试、调试和升级等，自顶而下的设计方法存在致命的缺陷。因为面向过程编程强调从软件的功能特性出发思考问题，将系统划分为多个功能模块，同时尽量确保模块之间的耦合度最小。其实这种方式并不能很好地模拟现实世界，其思维方式存在先天的缺陷。

在面向过程编程时，经常会遇到这样的问题，一个 bug 修改好了，另一个地方又出问题了，因为许多 bug 源于修改代码。实际上人们在理清楚代码的运行原理，寻找 bug 和防止出现不良副作用上，花费了大量的时间，而修改 bug 的时间却很短。由不良副作用产生的 bug 是最难发现的，如果让一个函数处理很多不同的数据，一旦需求发生变化，那么出现的问题会更多。需求的变更会对软件开发和维护工作产生极大的影响，因为只关注函数，将会导致一连串难以避免的变化。

因为用户的需求总是在变化之中，所以我们将无法阻止变化。与其抱怨变化，不如改变开发过程，从而更有效地应对变化，面向对象编程就是这样作为对抗软件复杂性的手段出现的。

1.1.2 思维差异

学习的最高境界是“知其然知其所以然”，但真正达到这个层次的人并不多，这是

每个人梦寐以求的人生目标。如果你已经进入了这样的化境，则一切问题迎刃而解。将不再受限于年龄，且与性别无关。

其实，牛人和普通人的差距并非知识和经验的多少，而是思维方式的不同。为何编程语言、操作系统、控制论等都是美国人发明的呢？他们似乎天生具备自上而下分析问题的直觉和从特殊到一般的泛化思维。美国之所以在 IT 领域处于绝对领先的地位，是与他们的教育理念和方法密切相关的。

我们时常以美国白领计算能力太差作为笑料，认为中国教育重在练“基本功”，掩盖、忽略了培养学生“创造力和思维能力”方面的欠缺。而美国教育极其重视“创造力”的培养，让学生根据个人的兴趣而学。例如，选 6 个学期的物理和数学，只选 2 个学期的化学和生物。不仅可以在校选修大学课程，而且大学还承认学生在高中阶段学习大学课程的学分，甚至还可以利用假期到任何大学学习自己感兴趣的内容，如哲学、Java。

由于创造力的不足和思维能力的差异，导致我们在很多问题的认识上出现大量的知识盲点或黑洞，从而将严密的科学知识割裂开来成为了知识孤岛。即便你非常努力学习，甚至花费更多的时间到企业实习，但依然难以获得更大的突破。

在自我训练的过程中，我深刻地体会到，对问题的研究经常会受到传统思维的影响。而《异类》一书指出，“人们眼中的天才之所以卓越非凡，并非天资超人一等，而是付出了持续不断的努力，一万小时的锤炼是任何人从平凡变成超凡的必要条件。”我开始感到迷茫，因为我花费的时间，又何止一万小时呢？

1.1.3 语言的鸿沟

开发人员对问题域的认识是一种思维活动，而人类的任何思维活动都是借助于他们熟悉的某种自然语言进行的。而软件系统的最终实现必须用一种计算机能够阅读和理解的语言描述系统，这种语言就是编程语言。

人们所习惯使用的自然语言和计算机能够理解及执行的编程语言之间存在很大的差距，这种差距被称为“语言的鸿沟”，实际上也是认识和描述之间的鸿沟。也就意味着，一方面人们借助自然语言对问题域所产生的认识远远不能被机器理解和执行，另一方面机器能够理解的编程语言又很不符合人们的思维方式。因此开发人员需要跨越两种语言之间的这条鸿沟，即从思维语言过渡到描述语言。

之所以学习和开发的成本很高，主要是理解困难所造成的，所以必须建立一种用于沟通的通用语言，构建通用语言的过程就是自我思维训练和建立逻辑推理的过程。

1. 变量和指针

首先从变量的三要素开始学起，一起建立一套通用语言。例如：

```
int iNum = 0x64;
```

最初的词汇有变量的类型 int、变量名 iNum 和变量的值 0x64，接着编写第一个只有几条语句的简单程序，详见程序清单 1.1。

程序清单 1.1　输出变量的地址和变量的值

```
1   #include<stdio.h>
2   int main(int argc, char *argv[])
3   {
4       int iNum = 0x64;
5
6       printf("%x, %x\n", &iNum, iNum);
7       return 0;
8   }
```

通过运行结果可以清楚地看到，0x64 存储在 0x22FF74 内存单元中。虽然使用“&”运算符可以获取变量 iNum 在内存中的地址，但 &iNum 是一个孤立的概念。

当将地址形象化地称为“指针”时，即可通过该指针找到以它为地址的内存单元。于是词汇表中又多了一个新的成员——指针，同时还多了一条新的语句——&iNum 是指向 int 变量 iNum 的指针，该语句标识了“指针与变量”之间的关联关系。理解指针的最好方法之一是绘制图表，变量与指针的关系详见图 1.1。

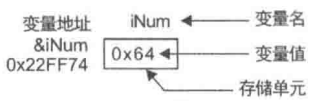

图 1.1　变量与指针

既然通过 &iNum 就能找到变量 iNum 的值 0x64，那么如何存放 &iNum 呢？定义一个存放指针 &iNum 的(指针)变量。例如：

```
int iNum = 0x64;
int *ptr = &iNum;
```

现在词汇表中又多了一个新的成员——指针变量，即 ptr 是指向 int 的指针(变量)，“int *”类型名是指向 int 的指针类型，详见图 1.2。虽然有时也将指针变量泛化为指针，但要根据当前所处的环境而定。

此时，大家不约而同地说出了——&ptr 是指向指针变量 ptr 的指针，那么谁指向 &ptr 呢？指针的指针，或双重指针，于是词汇表中又多了一条语句。如果有以下定义：

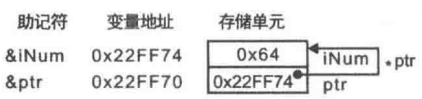

图 1.2　变量的存储与引用

```
int iNum = 0x64;
int * ptr = &iNum;
```

如果在“int * ptr”定义前添加 typedef，即

```
typedef int * ptr;
```

此时，ptr 等同于 int * 。为了便于理解，将类型名 ptr 替换为 PTR_INT。例如：

```
typedef int * PTR_INT;
```

有了 PTR_INT 类型，即可构造指向 PTR_INT 类型的指针变量 pPtr，即

```
PTR_INT  * pPtr = &ptr;
```

其中，pPtr 是指向 PTR_INT * 的指针变量，PTR_INT 的类型为 int * ，pPtr 是指向 int * * 的指针变量，那么 pPtr 就成了保存 int 型 ptr 地址的双重指针。其定义方式简写如下：

```
int * * pPtr = &ptr;
```

以下关系恒成立：

```
* pPtr == * (&ptr) == ptr
* * pPtr == * ptr == * (&iNum) == iNum
```

当读者读到这里时，我相信你对指针已经没有什么恐惧感了。

2. 数　组

如果有以下声明：

```
int a[2];
```

那么你是否想过在声明“int a[2];”时，a、&a[0]和 &a 是什么类型？大多数人可能认为，a 不是指向该数组首元素的指针吗？可以说“是”，也可以说“不是”。为什么一个看起来似乎很简单的问题，却会让人感到很迷茫呢？

由于数组类型属于构造类型，因此通过逻辑推理可以搞清楚其中的来龙去脉，基于此，我们需要以全新的理念看待数组，从变量的类型、变量的地址和变量的值这三个不同的视角出发进行分析。

从概念层次上来看，数组名是概念也是符号，由于它在声明、表达式、作为 & 和 sizeof 的操作数时具有一定的差异，因此需要区别对待。

按照变量的声明规则：a 是由 2 个 int 值组成的数组，取出标识符 a，剩下的 int [2]就是 a 的类型。这是在开发编译器时约定的声明规则，不需要特别做出解释。通常将 int [2]解读为由 2 个 int 值组成的数组类型，简称数组类型。而大多数 C 语言

教材几乎都不提及，从而为构造二维数组带来了很大的障碍。事实上，C 语言中并不存在二维数组，且 C 语言语法也不支持二维数组，仅支持由一维数组构造的数组的数组。

从规约层次上来看，除了在声明中或数组名当作 sizeof 或 & 的操作数之外，表达式中的数组(变量)名 a 被解释为指向该数组首元素 a[0]的指针，因而可将这个原则标识为“a==&a[0]”或等价于“*a==*(&a[0])==a[0]”。当 a[0]作为 & 的操作数时，&a[0]是指向 a[0]的指针，a[0]的类型为 int，&a[0]的类型为 int * const，即指向常量的指针，简称常量指针，其指向的值不可修改。当 a 作为 & 的操作数时，&a 是指向 a 的指针。

使用指针运算规则，当把一个整型变量 i 和一个数组名相加时，将得到指向第 i 个元素的指针，即“a+i==&a[i]”或“*(a+i)==*(&a[i])==a[i]”，因为 a 在编译期和运行时的语义完全不同。

由于 a 的类型为 int [2]，因此 &a 是指向“int [2]数组类型”变量 a 的指针，简称数组指针。其类型为 int (*)[2]，即指向 int [2]的指针类型。为何要用“()”将“*”括起来？如果不用括号将星号括起来，那么“int (*)[2]”就变成了“int *[2]”，而 int *[2]类型名为指向 int 的指针的数组(元素个数 2)类型，这是设计编译器时约定的语法规则。

问题又来了，C 语言的发明者 K&R 编写的 C 语言教材是这样解释“(*)”的：因为“*”是前置运算符，它的优先级低于“()”。为了让连接正确地运行，有必要加上括号。其实这样解读未免有些牵强附会了，因为声明中的“*、()、[]”都不是运算符，运算符的优先顺序在语法规则中是在表达式中定义的。回归本质“int *[2]”中的“*”不加括号，那就是指针类型数组。如果事先不是这样约定，则无法开发编译器。

从实现层次上来看，最重要的不是就事论事地实现，而是要从特殊到一般地寻找问题的通解——泛化。例如，在一个数组中寻找一个特定的值，常见的函数原型如下：

```
int *findValue(int *arrayHead, size_t arraySize, int value);
```

显然，findValue()函数不仅暴露了数组的实现细节，如 arraySize，而且太过于依赖特定的数据结构。那么，如何设计一个算法，使它适用于大多数数据结构呢？或者如何在即将处理的未知的数据结构上，正确地实现所有操作呢？实际上，一个序列有开始和结尾，即可使用++得到下一个元素，也可以使用*得到当前元素的值。

为了让 findValue()函数适用于所有类型的数据结构，其操作应该更抽象化，让 findValue()函数接受两个指针作为参数，表示一个操作范围，详见程序清单 1.2。

程序清单 1.2　findValue()查找函数

```
1    int * findValue(int * begin, int * end, int value)
2    {
3        while(begin != end && * begin != value)
4            ++begin;
5        return begin;
6    }
```

显而易见，这样的循环结构重复做某件事就是一种迭代操作，在每次迭代操作中，对迭代器 begin 的修改等价于修改循环控制标志或计数器。当将 begin 的作用抽象化和通用化后，begin 和 end 就成为了迭代器。其基本思想是，迭代器变量存储了数组的某个元素的位置，因此能够遍历该位置的元素。通过迭代器提供的方法，可以持续遍历数组的下一个元素。

“++”将迭代器移动到下一个数据项，其意图是利用递增操作完成赋值。“*”检索迭代器的当前元素，其意图是利用递增操作检索元素。为了检测输入迭代器元素的结尾，通常将迭代器与另一个恰好位于输入区间末尾之后的已知迭代器 end 进行比较，测试迭代器是否相等。

这个函数在“前闭后开”范围[begin, end)内(不含 end，end 指向 array 最后元素的下一个位置)查找 value，并返回一个指针，指向它所找到的第一个符合条件的元素，如果没有找到就返回 end。这里之所以用“!=”，而不是用“<”判断是否到达数组的尾部，是因为这样处理更精确。永远也不要从“*end”读取数据，更不要向它写入数据。

由此可见，findValue()函数中并无任何操作是针对特定的整数 array 的，即只要将操作对象的类型加以抽象化，且将操作对象的表示法和范围目标的移动行为抽象化，则整个算法就可以工作在同一个抽象层面上了。通常将整个算法的过程称为算法的泛型化。泛化的目的旨在使用同一个 findValue()函数处理各种数据结构，通过抽象创建可重用代码。

3. 二维数组

假设有以下声明：

```
int data0[2] = {1, 2};
int data1[2] = {3, 4};
int data2[2] = {5, 6};
```

当去掉声明中的数组名时，则 data0、data1 和 data2 类型都是“int [2]”。实际上，数组本身也是一种数据类型，当数组的元素为一维数组时，该数组为数组的数组，因此可以通过“int [2]”数组类型构造数组的数组，即

```
typedef int data0[2];
```