HZ BOOKS
华章科技

以C语言中的函数设计作为全书主线，串联与之相关的指针、数组、函数、多文件编程等难点，通过实际问题案例帮助读者克服编程中这些难于掌握的概念和技术。

以“解惑”作为编写宗旨，启迪读者的编程思维方式，帮助读者快速进化为编程高手。

C语言解惑

C Language Demystified

Pointer, Array, Function and Multi-File Programming

指针、数组、函数和多文件编程

刘振安 刘燕君 编著

机械工业出版社
China Machine Press

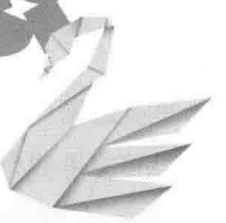
C语言解惑

C Language Demystified

Pointer, Array, Function and Multi-File Programming

指针、数组、函数和多文件编程

刘振安 刘燕君 编著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C 语言解惑：指针、数组、函数和多文件编程 / 刘振安，刘燕君编著．—北京：机械工业出版社，2016.12

ISBN 978-7-111-55406-6

I. C… II. ①刘… ②刘… III. C 语言－程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2016）第 289020 号

C 语言解惑：指针、数组、函数和多文件编程

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：佘 洁　　责任校对：殷 虹

印　　刷：北京诚信伟业印刷有限公司　　版　　次：2017 年 1 月第 1 版第 1 次印刷

开　　本：186mm×240mm 1/16　　印　　张：28.5

书　　号：ISBN 978-7-111-55406-6　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

Preface 前　言

C 语言编程仍然是编程工作者必备的技能。本书的基础版本《C 语言解惑》[⊖]通过比较编程中存在的典型错误，从而实现像雨珠打在久旱的沙滩上一样滴滴入骨的效果，使学习者更容易记住编程的要诀，并通过演示如何将一个能运行的程序优化为更好、更可靠的程序，使读者提高识别坏程序和好程序的能力。尽管如此，那本书仍然要照顾初学者并兼顾知识的完整性，所以讨论的深度有所限制。为此，我们决定推出它的提高版，并将讨论聚焦于函数设计。

本书将集中讨论 C 语言的核心部分——函数设计。函数设计涉及函数类型、函数参数及返回值，这就要求读者熟练掌握指针和数组的知识，此外，还要掌握多文件编程以及多文件之间的参数传递等知识。

因为本书要求读者已经学过 C 语言，所以我们可以完整、系统地论述各个部分的内容，无须赘述基础知识。本书的另一个特点是每一章之间都有知识交叉，进而达到讲透的目的。如果遇到不清楚的知识点，读者可以自行学习相应参考资料，也可以与《C 语言解惑》配合学习。

本书的落脚点是实现 C 语言的结构化程序设计。为实现这一目标，本书专门选择了完整的设计实例。尤其是第 10 章，结合趣味游戏程序，综合讲解函数设计和多文件编程。

本书各个部分论述详细，涉及的知识面广，有些知识是传统教材中所没有的，所以它既可以作为从事教学的老师及工程技术人员的参考书，也可以作为常备手册。其实，它不仅对工程技术人员极有参考价值，也能帮助在校生进行编程训练或作为毕业论文的参考资料。此外，本书对于初学者也大有帮助，他们可以将它作为课外读物，对目前看不懂的地方，可以等具备相关知识之后再来研究，彼时将收获更大。总之，本书能帮助各类人群找到自己需要的知识并有所收获，而这也将拓宽本书的应用范围。

⊖ 此书已于 2014 年由机械工业出版社出版，书号 978-7-111-47985-7。

本书共分 10 章。第 1 章通过例子说明引入指针变量的必要性并简单介绍指针变量的基本性质。第 2 章通过实例解释指针的基本性质。第 3 章介绍数组及数组的边界不对称性。第 4 章介绍 C 语言中两个非常重要的概念——数组和指针。第 5 章介绍如何掌握函数设计和调用的正确方法。第 6 章介绍如何设计合理的函数类型及参数传递方式。第 7 章先讨论函数设计的一般原则，然后结合典型算法，用实例说明设计的具体方法，以便使读者进一步开阔眼界。第 8 章结合具体实例详细介绍头文件的编制、多个 C 语言文件及工程文件的编制等方法，以提高读者的多文件编程能力。第 9 章给出两个典型的多文件编程实例，一个使用链表，另一个使用数组。第 10 章中的游戏程序实例将加深读者对一个完整工程项目的理解。为了学习方便，本书提供全部程序代码。

本书的两位作者分别撰写各章的不同小节，然后逐章讨论并独立成章。刘燕君负责第 1 ~ 6 章，刘振安负责第 7 ~ 10 章，最后由刘振安统稿。参与本书工作的还有周淞梅实验师、苏仕华副教授、鲍运律教授、刘大路博士、唐军高级工程师等。

在编写过程中，我们得到了中国科学院院士、中国科学技术大学陈国良教授的大力支持，特此表示感谢！对书中所引用资料的作者及网络作品的作者表示衷心感谢！

作　者

zaliu@ustc.edu.cn

2016 年 6 月

Contents 目　录

第 1 章 Chapter 1

引入指针变量

指针在C语言中具有举足轻重的地位，也是编制C程序的基本功之一。本章将通过例子说明引入指针变量的必要性并简单介绍指针变量的基本性质。

1.1 变量的三要素

一个变量具有3个要素：数据类型、名字和存放变量的内存地址。本节将简要回顾变量的3个要素，以便为引入指针打下基础。

1. 基本数据类型

数据类型是C语言中非常重要的一个概念，它将C语言所处理的对象按其性质不同分为不同的子集，以便对不同类型的数据规定不同的运算。void是无类型标识符，只能声明函数的返回类型，不能声明变量，但可以声明指针。

本节只涉及基本数据类型，C语言的基本数据类型有如下4种。

- ❑ char　字符型
- ❑ int　整数型
- ❑ float　浮点数型（又称为单精度数）
- ❑ double　双精度浮点数型

另外还有用于整型的限定词short、long、signed和unsigned。short和long表示不同长度的整型量；unsigned表示无符号整型数（它的存放值总是正的）；可以省略signed限定词。例如，可以将如下声明

```
short  int      x;
unsigned int    z;
```

中的说明符 int 省略。即它们与如下声明

```
short       x;
unsigned    z;
```

是等效的。上述数据类型的长度及存储的值域也随编译器不同而变化，ANSI C 标准只限定 int 和 short 至少要有 16 位，而 long 至少 32 位，short 不得长于 int，int 不得长于 long。表 1-1 是数据类型的长度及存储的值域表，表 1-1 中 VC 是 Visual C++ 6.0 的缩写。表 1-2 是加了限定词的数据类型及它们的长度和取值范围。

C 语言提供一个关键字 sizeof，用来求出对于一个指定数据类型，编译系统将为它在内存中分配的字节长度。例如，语句“printf("%d",sizeof(double));”的输出结果为 8。

注意在表 1-1 中的标注，在 VC 中 int 使用 4 字节，这是本章计算的依据。

C 语言定义的存储类型有 4 种：auto、extern、static 和 register，分别称为自动型、外部型、静态型和寄存器型。自动型变量可以省略关键字 auto。存储类型在类型之前，即

```
存储类型    类型
```

例如 auto int 和 static float 等。可以省略 auto，其他类型均不可以省略。

表 1-1 数据类型的长度及存储的值域

类 型	位长度	字节数	值 域	备 注
char	8	1	0 ~ 255	
int	16	2	–32 768 ~ 32 767	VC 使用 4 字节
float	32	4	3.4E–38 ~ 3.4E+38	
double	64	8	1.7E–308 ~ 1.7E+308	

表 1-2 加限定词的数据类型及其长度和取值范围

类 型	位长度	字节数	值 域	备 注
short int	16	2	–32 768 ~ 32 767	
long int	32	4	–2 147 483 648 ~ 2 147 483 647	
unsigned int	16	2	0 ~ 65 535	VC 使用 4 字节

2. 变量的名字和变量声明

C 语言中大小写字母是具有不同含义的，例如，name 和 NAME 就代表不同的标识符。原来的 C 语言中虽然规定标识符的长度不限，但只有前 8 个字符有效，所以对定义为

```
dwNumberRadio
dwNumberTV
```

的两个变量是无法区别的。

现在流行的为 32 位操作系统配备的 C 编译器已经能识别长文件名，不再受 8 位的限制。另外，在选取时不仅要保证正确性，还要考虑容易区分，不易混淆。例如，数字 1 和字母 i 在一起，就不易辨认。在取名时，还应该使名字有很清楚的含义，例如使用 area 作为求面积函数的名字，area 的英文含义就是“面积”，这就很容易从名字猜出函数的功能。对一个可读性好的程序，必须选择恰当的标识符，取名应统一规范，以便使读者能一目了然。

在现在的编译系统中，内部名字中至少前 31 个字符是有效的，所以应该采用直观的名字。一般可以遵循如下简单规律。

1）使用能代表数据类型的前缀。

2）名称尽量接近变量的作用。

3）如果名称由多个英文单词组成，每个单词的第一个字母大写。

4）由于库函数通常使用下划线开头的名字，因此不要将这类名字用作变量名。

5）局部变量使用比较短的名字，尤其是循环控制变量（又称循环位标）的名字。

6）外部变量使用比较长且贴近所代表变量的含义。

7）函数名字使用动词，如 Get_char(void)。变量使用名词，如 iMen_Number。

变量命名可以参考 Windows API 编程推荐的匈牙利命名法。它是通过在数据和函数名中加入额外的信息，既增进程序员对程序的理解，又方便查错。

所有的变量在使用之前必须声明，所谓声明即指出该变量的数据类型及长度等信息。声明由类型和具有该类型的变量列表组成。如：

```
int  lower, upper;
char c, name[15];
```

变量可按任何方式分布在若干个声明中，上述声明同样可以写成：

```
int  lower;          //整数类型
int  upper;          //整数类型
char  c;             //字符类型
char  name[15];      //字符数组，可连续存放 15 个字符
```

后一种形式会使源程序冗长，但便于给每个声明加注释，也便于修改。

变量的存储类型在变量声明中指定。变量声明的一般形式为：

```
存储类型    类型    变量名列表;
```

应该养成在声明时就为变量赋初值的习惯，但在某些特殊场合则只能声明，如头文件中对外部变量的声明，下面是一些典型的例子。

```
auto int a;
static float b, c;
extern double x;
register int i=0;
extern char szClassame[ ];
```

```
static int size=50;
const double PI=3.14159;
```

3. 变量的地址

内存地址由系统分配，不同机器为变量分配的地址大小虽然可以不一样，但都必须给它分配地址。

在 C 语言中，声明和定义两个概念是有区别的。声明是对一个变量的性质（如构成它的数据类型）加以说明，并不为其分配存储空间；而定义则是既说明一个变量的性质，又为其分配存储空间。定义一个函数，也是为它提供代码。

1.2 变量的操作

从三要素可知，既可以通过名字对变量进行操作，也可以通过地址对存放在该地址的变量进行操作。

1. 左值和右值的概念

变量是一个指名的存储区域，左值是指向某个变量的表达式。“左值”来源于赋值表达式“ A=B”，其中左运算分量“ A”必须能被计算和修改。左值表达式在赋值语句中既可以作为左操作数，也可以作为右操作数，例如“ x=56”和“ y=x”，x 既可以作为左值（x=56），又可以作为右值（y=x）。但右值“56”只能作为右操作数，而不能作为左操作数。由此可见，常量只能作为右值，而普通变量既可以作为左值，也可以作为右值。如下语句

```
const int a = 256;
```

定义的 a，显然不能作为左值，只能作为右值。

由此可见，值可以作为右值，如整数、浮点数、字符串、数组的一个元素等。在 C 语言中，右值以单一值的形式出现。假设有字符数组 a 和 b，则这两个字符数组的每个元素均可以作为右值，即“a[0]=b[0]”是正确的，“b[0]=a[0]”也是正确的。需要注意的是，它们在“=”号左右两边的含义是不同的。以 a[0] 为例，在“ b[0]=a[0]”中，它是作为值出现的，即 a[0] 是数组第 1 个元素的值；而在“ a[0]=b[0]”中，它是作为变量出现的，即 a[0] 是数组的第 1 个元素的变量名，所以 a[0] 可以作为左值。即可以使用数组的具体元素作为左值和右值。

a 和 b 都不是字符串的单个元素，所以都不能作为右值。而因为 a 和 b 可以作为数组首地址的值赋给指针变量，所以在这种情况下它们又都可以作为右值。

由此可见，在 C 语言中，左值是一个具体的变量，右值一定是一个具体类型的值，所以有些可以既可以作为左值，也可以作为右值，但有些只能作为右值。

2. 对变量的基本操作

C 语言使用地址运算符“ &”来取变量存储在内存中的首地址。假设变量 a=55，但不

同机器和系统为它分配的地址是不一样的，这里也假设分配的十六进制地址是 0x0012FF7C。如何从这个地址取出“55”呢?

C 语言提供了“*”运算符，用来取出地址里的值。“&a”代表地址，显然“*&a”可以取出 55。使用下面语句

```
printf("%d,%d\n",a,*&a,);
```

可以得到输出结果为“55,55”，即证明 a 和 *&a 是等价的。

1.3　指针变量

1.2 节介绍了“*”和“&”运算符，本节将通过具体的例子说明它们的用途，从而引入指针变量。

1. 对有效地址进行操作

【例 1.1】取地址里的值和取地址里存放的地址值的例子。

```
#include <stdio.h>
int main()
{
    int a = 65;
    int addr;

    addr = 0x0012ff7c;                                  //将 a 的首地址存入变量 addr
    printf("0x%p, 0x%p, 0x%p\n", &a, addr, &addr);      //输出 3 个地址
    printf("%d,%d,0x%p\n", a, *&a, *&addr);             //输出变量及地址里的值

    return 0;
}
```

语句“int a=65；”定义了整型变量 a 的值为 65，VC 使用 4 字节存储 65。假设存放它的内存首地址为十六进制的“0x0012ff7c”，则可以使用输出格式“%p”来输出这个地址。“0x”是标注它为十六进制地址，也可以简单地使用“%#p”输出地址。

一个变量具有地址和值，& 是取地址值运算符。系统为整型变量 a 和 adder 分别分配地址“0x0012ff7c”和“0x0012ff78”。给整型变量 addr 赋十六进制数，这个数可以代表地址，但不一定是有效的地址（将计算机可以存取的地址称为有效地址）。已经验证 0x0012ff7c 是分配给变量 a 的地址，所以 addr 是被赋给一个有效地址。

将“0x0012ff7c”赋给变量 addr，&addr 是系统分给它的地址“0x0012ff78”，这个地址与 a 的地址相差 4 字节，证明它们是连续存放的。现在这个地址里存放的是地址 0x0012ff7c，也就是变量 a 的地址。因为 *&addr 应该输出 a 的地址而不是 a 的值，所以要使用 %p 格式。程序输出结果也验证了如上分析。即输出为

```
0x0012FF7C,0x0012FF7C,0x0012FF78
65,65,0x0012FF7C
```

既然 addr 存放的是有效地址，*addr 也应该能输出这个地址里的值，也就是变量 a 的地址。不过，&a 虽然和 addr 的值是一个值，但它们对运算的反应并不一样。a 是变量，其值为 65，&a 是存储地址，所以 *&a 是取地址里的值。类似的，*addr 应该输出它的存储内容，即地址“0x0012ff7c”，而 **addr 应该输出地址“0x0012ff7c”里的内容 65。其实这是不行的，因为编译系统并不知道 *addr 存储的“0x0012ff7c”是地址，所以将它作为整数，因此编译系统会报错，当然使用 **addr 也要出错。

但 addr 里面确实装的是地址，所以可以将这个整数强制转为地址。*addr 加上强制转换，应该是“(int *)addr”，它的内容是变量 a 的地址“0x0012ff7c”。

再对它使用 * 运算符，即 *(int *)addr，输出结果应该是存在这个地址里的变量 a 的值 65。下面的例子验证了如上分析。

【例 1.2】取地址里的整数值和取地址里存放的地址值。

```
#include <stdio.h>
int main()
{
    int a=65;
    int addr;
    addr=0x0012ff7c;

    printf("0x%p, 0x%p, 0x%p\n", &a, addr, (int*)addr);
    printf("%d,%d,%d\n",a,*&a, *(int*)addr);

    return 0;
}
```

输出结果如下：

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C
65,65,65
```

2. 引入指针的概念

在【例 1.2】中，要使用 a 的地址直接给 addr 赋值，必须事先知道这个地址。为了避免这个麻烦，可以直接将地址表达式“&a”赋给变量。即

```
addr=&a;
```

因为 &a 是地址值，addr 是整型变量，所以会给出警告信息。不过，可以使用强制转换让警告信息“闭嘴”。即

```
addr=(int)&a;
```

这样一来，使用起来就方便多了。

【例 1.3】直接将变量地址赋给另一个变量的例子。

```
#include <stdio.h>
int main()
{
    int a=65;
    int addr;
    addr=(int)&a;
    printf("0x%p, 0x%p, 0x%p\n", &a, addr, (int*)addr);
    printf("%d,%d,%d,\n",a,*&a, *(int*)addr);

    return 0;
}
```

程序运行结果如下：

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C
65,65,65,
```

运行结果完全吻合。如果想像对待变量一样对待 addr，即使用“ *”和“ &”运算符的结果与变量 a 一样，就必须定义新的变量类型。分析下述表达式：

```
addr=(int)&a;
(int*)addr
*(int*)addr
```

由此可见，如果定义一种变量，使它存储的数据类型是地址，问题就可以迎刃而解了。要使用 (int*)addr 的 addr 存储地址，那么就要用“int *”声明“addr”，即

```
int * addr;
```

这时“addr=&a”就无需转换，赋值顺理成章了。

“addr”输出地址，则“ *addr”输出地址里的值。这就与普通变量的使用方法完全一样了。

暂且将使用“int *”定义的变量称为指针变量，下面编程验证一下这个设想。

【例 1.4】使用新的数据类型（指针）的例子。

```
#include <stdio.h>
int main()
{
    int a=65;
    int *p;
    p=&a;

    printf("0x%p, 0x%p, 0x%p\n", &a, &p, p);
    printf("%d,%d,%d\n",a,*&a, *p);

    return 0;
}
```

输出结果如下：

```
0x0012FF7C, 0x0012FF78, 0x0012FF7C
65,65,65,
```

输出结果与【例 1.3】的完全一样。

这种类型称为指针类型，指针类型存储的是地址值。因为这里使用的地址值是另外一个变量的地址，所以是有效地址。要明确的是，地址值不一定是有效地址，所以说从指针的引入开始，也就暗示着它存在着无法预防的错误。

3. 引入字符指针再次验证

下面再使用字符来验证一下，看是否与整数的结论相同。

【例 1.5】使用字符的例子。

```
#include <stdio.h>
int main()
{
    char a='B';
    int addr;

    addr=(int)&a;

    printf("0x%p, 0x%p, 0x%p\n", &a, addr, (char*)addr);
    printf("%c, %c, %c\n", a,*&a, *(char*)addr);

    return 0;
}
```

程序输出结果如下：

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C
B, B, B
```

程序验证了“(char*)addr”和“*(char*)addr”的作用，从而推知，可以定义字符类型的指针。

【例 1.6】使用字符指针的例子。

```
#include <stdio.h>
int main()
{
    char c='B';
    char *p;
    p=&c;

    printf("0x%p, 0x%p, 0x%p\n", &c, &p, p);
    printf("%c, %c, %c\n", c, *&c, *p);

    return 0;
}
```