HZ BOOKS
华章科技

大学程序设计课程与ACM竞赛实训教材

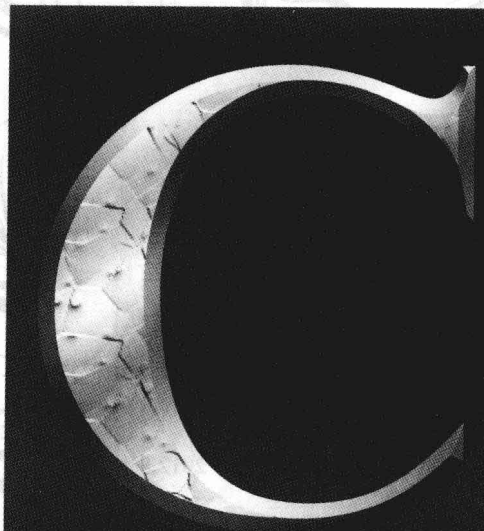
C程序设计竞赛实训教程

C Programming Contest Training Course

刘高军 何丽 编著

机械工业出版社
China Machine Press

大学程序设计课程与ACM竞赛实训教材



C程序设计竞赛实训教程

C Programming Contest Training Course

刘高军 何丽 编著

机械工业出版社
China Machine Press

本书是以大学生程序设计竞赛为应用背景的程序设计综合训练教程，以具有初级 C 语言基础的读者为对象，从 C 语言的深度解析和程序设计基本方法两方面进行分析介绍，使读者达到深入理解 C 语言和全面掌握程序设计基本方法的目的。全书内容按常用程序设计方法划分为不同专题，理论联系实际，强调动手实践，深入浅出，便于读者学习和理解。

本书适合具有一定 C 语言基础的初、中级读者使用，可作为大学程序设计课程或参加 ACM 竞赛的培训教材，也可作为相关专业师生的参考用书。

图书在版编目（CIP）数据

C 程序设计竞赛实训教程/刘高军，何丽编著. -北京：机械工业出版社，2012.8

ISBN 978-7-111-38917-0

I. ①C… II. ①刘… ②何… III. ①C 语言－程序设计－竞赛－高等学校－教材 IV. ①TP312

中国版本图书馆CIP数据核字（2012）第135909号

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：夏非彼　迟振春

中国电影出版社印刷厂印刷

2012年8月第1版第1次印刷

188mm×260mm • 14.75 印张

标准书号：ISBN 978-7-111-38917-0

定价：29.00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）88378991；82728184

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）82728184；88379603

读者信箱：booksaga@126.com

前言

Preface

随着计算机技术的发展，计算机已应用到社会的各个领域，对我们生活、生产、科技等各个方面都产生了很大影响。计算机教育也已成为大学教育的一个重要组成部分，C语言程序设计作为计算机专业的基础课程，是学生必须要掌握的，也是进行思维方法、问题抽象和解决等方面训练的有效工具。在C语言程序设计的教学中，基本上是以语句、语法和程序基本结构为重点，以语言自身体系内容开展教学，对于如何从问题本身入手进行分析、抽象和解决等方面涉及不深，虽然语言中的各种语法现象和程序结构都已基本掌握，但在实际应用中，特别是解决一些有一定难度的问题时，还是感到无从入手。目前，各类学科竞赛，特别是ACM国际大学生程序设计大赛在学生中有很大影响力，学生们在学习了程序设计课程之后，都有参加各类竞赛的愿望，希望自己能在竞赛中有所表现，但仅从程序设计课程中掌握的知识来看，是明显不够的。怎样才能快速提高学生的语言应用能力，提高对实际问题的分析和抽象能力，使学生们在掌握了基本语言后，能够进一步深入了解和掌握语言特性，并能在各种程序设计竞赛中，展示自己在问题抽象和程序设计方面的能力呢？我们通过多年的教学实践，对学生们进一步学习的需求和愿望有较清楚的了解。学生们想通过进一步的学习和实践，更加深刻地了解语言本身的细节特点，循序渐进地学习一些基础算法知识，以达到运用基础算法解决一些实际问题的目的。

本书就是在这种背景下编写的，主要对象是基本掌握了C语言语法知识的读者。书中总结了多年的教学经验，在内容编排上从易到难，学习过程从简单到复杂、再到常用算法的应用，逐步引导学生在实践中掌握科学的思维方式，实现真正的从语言到程序设计的跨越。本书的一部分内容是对C语言的深入解析，这里我们不再讲解基本语法和程序结构，而是对C语言中经常容易混淆和不易理解的知识点进行深入分析，使读者真正掌握C语言的精髓；另一部分内容是程序设计的基本方法，这里总结了程序设计中的常用方法，采用以问题为中心的讲授方式，重点介绍问题的分析和抽象方法，力求理论与实际相结合、算法与程序相统一，突出方法在解决实际问题中的应用。所有问题及方法的叙述均采用统一格式，对每个题目从问题描述、问题分析到程序实现连贯而自成一体，方便读者理解和掌握。

本书结构简明清晰，内容深入浅出，全书共分11章。

第1章：语言解析，主要对C语言中的难点问题进行分析，从更深层次解读C语言中不易理解的知识点。

第2章：输入输出格式，按ACM大学生程序设计竞赛以及一些学科竞赛中常用的数据输入输出格式进行分类，讨论了不同输入输出格式的实现方法。

第3章：简单数据处理问题，主要包括一些常见的简单问题，涉及的理论一般为初等数学，或者是一些现实生活中的常识，通过对问题进行分析和抽象，得到解决问题的方法。

第4章：递推，在介绍常用递推方法的基础上，重点讨论了具有递推关系问题的抽象方法，通过对典型问题求解方法的分析，讲述递推方法在实际问题中的应用。

第5章：进制转换问题，介绍了程序设计中常用的进制转换问题的实现方法，并讨论了各种不同进制之间进行转换的一般方法。

第6章：字符串处理问题，介绍了字符串处理的相关函数，通过实例介绍字符串处理的方法和需要注意的一些问题。

第7章：大数问题，重点讲述了大数的存储和处理方法，通过实例介绍大数进行算术运算时所采用的一般方法。

第8章：枚举，介绍了枚举方法的基本思想和适合采用枚举方法求解的典型问题，重点讨论了在减少枚举量、提高程序效率方面的一般方法。

第9章：模拟，重点讲述了运算模拟和过程模拟，讨论了实际问题中非数值数据的抽象和表示，介绍了模拟方法在一些典型问题求解中的应用。

第10章：回溯与递归，详细介绍回溯与递归的基本思想以及程序实现的基本框架，通过实例介绍回溯与递归在实际问题中的应用。

第11章：搜索，介绍了采用搜索法解决问题的基本思路，重点讨论了深度优先搜索和广度优先搜索的基本方法和程序实现框架，通过实际问题讲述两种方法的应用。

书中所有程序均通过上机调试，部分章节配有例题，方便读者的学习和练习。

由于编者水平有限，书中所述难免有不当之处，恳请广大读者批评指正。

编　者

2012.06

目录 Contents

第1章

语言解析

C 语言作为一种应用广泛的语言，目前在各高校计算机专业课程体系中，都将其列为重要的专业基础课。当我们学习了 C 语言以后，对 C 语言的语法以及程序设计方法有了一定的了解，并且也用 C 语言编写过一些程序，但是对于 C 语言中的一些特点是否能真正把握，是否能够运用自如地利用语言特性来解决问题，写出精练清晰的程序，特别是 C 语言中有一些容易混淆的地方，这些都需要在不断地编程练习过程中逐步掌握和提高。

在这里我们不再讲解 C 语言中的基本语法和结构，而是对语言中一些难于理解和容易出错的内容进行讨论。当然在编写程序时，用法上可以采用其他方法，以避免问题的混淆，但从对 C 语言的深层理解角度，必须把这些问题搞清楚，特别是从概念上一定要清楚。这对于我们用 C 语言来处理复杂的数据结构、编写高效程序将起到至关重要的作用。

1.1 关于变量的存储类别

我们知道在设计程序的时候，肯定会用到变量，变量是我们用来保存数据的，设计程序时会根据要处理数据的类型和大小定义相应变量，C 语言中提供了丰富的数据类型，使我们可以根据需要定义不同类型的变量，如：

```
int a;
```

这是定义了一个整数类型的变量，其变量名为 a，简单理解就是定义了一个变量 a，a 中可以存放一个整型数，但这是不够的，首先要理解到，a 标识了内存的一块空间，所谓 a 中可以存放一个整型数，是指 a 所标识的这块内存空间里能存放一个整型数，那么这块空间有多大，占用多少个字节呢？这是由定义变量时所用的类型标识符决定的，在 32 位系统中用类型标识符 int 定义的变量，系统为其分配 4 个字节的空间，也就是说，一个整数存放在内存

中，系统只允许它占用 4 个字节的空间，这也就决定了我们所说的整型变量只能存放一个某一范围内的整数。理解了 int 类型，对于 short、long、char、float、double 这几个基本类型标识符定义的变量，也就不难理解了，它们和 int 定义的变量一样，所不同的只是各类型定义的变量，系统为其分配的空间大小不同。在 32 位系统中基本类型所占空间和表示数的范围如表 1-1 所示。

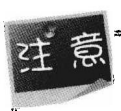
表 1-1 基本数据类型所占空间和表示数的范围

类型	类型标识符	占用字节数	存储数值范围
整型	int	4	-2147483648～2147483647
无符号整型	unsigned int	4	0~4294967295
短整型	short int	2	-32768~32767
无符号短整型	unsigned short int	2	0～65535
长整型	long int	4	-2147483648～2147483647
无符号长整型	unsigned long int	4	0~4294967295
字符型	char	1	-128～127
无符号字符型	unsigned char	1	0~255
浮点型	float	4	3.4×10^{-38}~3.4×10^{38}
双精度型	double	8	1.7×10^{-308}~1.7×10^{308}

变量除了类型属性外，还有另外一个属性就是存储类别，系统把程序中用到的数据区分成静态存储区和动态存储区，如图 1-1 所示。

数据区
静态存储区
动态存储区

图 1-1 内存区域划分



怎样来理解动态和静态这两个概念呢？静态指的是在这个区分配了空间的变量，在程序运行过程中一直存在，即相对是不变的；而动态指的是在这个区分配了空间的变量，可能随时被释放，即相对是不断变化的。

当我们定义一个变量时，不仅要告诉系统，为这个变量分配一个能存放什么类型数据的空间，同时还要告诉系统这个空间应在哪个区分配。怎样告诉系统？也就是怎样定义变量的存储类别，C 语言中提供了存储类别说明符，包括自动的（auto）、静态的（static）、寄存器的（register）和外部的（extern）共 4 种。语法格式为：

```
[存储类别说明符] 类型标识符 变量 [,变量]:
```

在定义变量时，存储类别说明符可以省略，缺省情况下系统会根据变量是局部变量还是全局变量，自动为变量在动态存储区或静态存储区分配空间，分配原则：由于全局变量在程

序运行过程中自始至终都存在，所以在静态区为其分配空间；而局部变量只是在其被定义的函数被执行时才存在，所以在动态区为其分配空间。这种缺省情况是自动的（auto），换句话说，缺省情况和使用存储类别说明符 auto 的实质是一样的，因此，我们就当 auto 这个存储类别说明符不存在，由系统“自动”定义就可以了。

我们能改变变量的存储区吗？实际上，全局变量一定是在静态存储区分配空间，是不能改变的，只有局部变量，既可以在动态区，也可以在静态区。也就是说除了系统自动分配的情况外，只有一种情况，即将局部变量放在静态区的情况，例如局部变量 a 定义成如下形式：

```
static int a;
```

对于局部变量 a，系统应在动态区为其分配空间，但在定义时添加了存储类别说明符 static，则系统就为变量 a 在静态存储区分配空间，既然是在静态存储区分配的空间，局部变量 a 既有局部有效的特性，也有长时间存在的特性，因此当定义了静态局部变量后，该变量自第一次被分配空间到程序运行结束一直存在，但它只是在定义它的函数内有效，这也决定了静态局部变量的一些特殊性，如：只在定义它的函数第一次被调用时，为该变量在静态存储区分配空间，同时若有初始化则进行初始化，并且函数返回，该变量空间并不释放，变量空间一直存在并保留最后对其操作的值，而在函数返回后，在该函数以外不能使用该变量，只有当函数再次被调用时，才可以继续使用该变量。静态局部变量的这种特性有时对一些特殊操作是很方便的。

例如：

```
int func(void)
{   static c=0;                    //局部静态变量 c，初始化为 0
    c=c+1;
    return(c);
}
void main()
{   int i,a;
    for(i=0;i<5;i++)
    {   a=func();                  //函数调用
        printf("%d   ",a);
    }
    printf("\n");
}
```

程序输出结果为：

```
1  2  3  4  5
```

在函数 func 第一次被调用时，系统为静态局部变量 c 在静态区分配空间，并赋初值 0，函数执行后 c 变为 1，当函数返回时，变量 c 的空间并不释放，仍然保留在静态区，并且保

留其现有数据值 1，当函数再次被调用时，不再为 c 重新分配空间，还是利用原有 c，所以执行加 1 操作后，c 变量值为 2，如此进行，变量 c 在程序结束前一直存在。

在程序设计中有时会利用这一特性，但需控制好函数的每次调用。

我们再看静态全局变量，前面说过，全局变量一定是在静态存储区分配空间，那所谓静态全局变量是什么意思呢？实际上，这里的 static 不是说明全局变量在哪个存储区分配空间，而是用来限制变量作用域的。这里我们就必须提到另一个存储类别说明符 extern，当在编写较大程序时，往往将代码分成不同的几个源文件分别存储，可以分别编译，最终连接到一起形成一个执行程序，这样便于程序的调试。我们知道，程序中定义的全局变量，它的作用域是从定义处开始到文件结束，如果在一个源文件中定义的全局变量（存储类别为 auto 或缺省），需要在另一个源文件中使用这个变量，则只须在另一个源文件中对其说明即可，这时的说明就要用到 extern 存储类别说明符。如果在一个源文件中定义的全局变量不允许其他文件中的代码使用，则将变量定义成静态全局变量，即使在其他文件中使用了 extern 声明也无法使用。例如在图 1-2 中定义的两个文件 file1.c 和 file2.c，file1.c 中定义了自动全局变量 a 和静态全局变量 b，则在文件 file2.c 中可以用 extern 来说明变量 a，在本文件以外的其他文件中已经定义了，这里直接使用即可。利用同样的方法来说明变量 b，但由于在文件 file1.c 中变量 b 被定义成静态全局变量，所以即使在文件 file2.c 中使用了 extern 说明 b 变量，也是无法引用。因此静态全局变量中定义的 static 的作用是限制变量作用域。

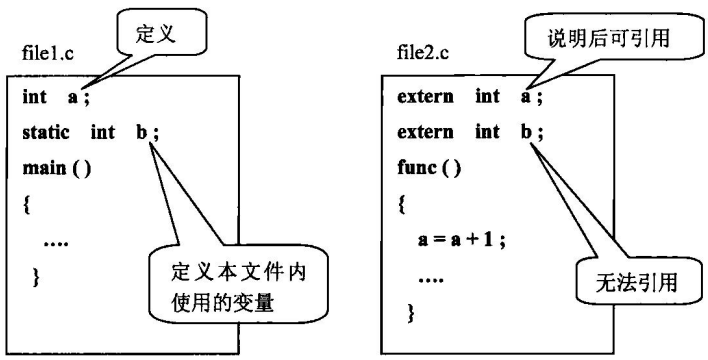


图 1-2 不同文件之间变量的引用

下面我们来看 register，由 register 定义的变量，当涉及到反复进行运算时，将变量存储在 CPU 的内部寄存器中，从而避免每次的内存访问，以达到提高效率的目的。但实际上，寄存器是十分有限的，并且受到数据类型的限制，所以它只是在一些特殊操作中起作用，况且并不是所有的编译器都能真正实现这一功能。

1.2 关于数值在内存中的表示

程序中所处理的数据，是存放在变量所标识的内存中，我们使用数据时是通过变量对内

存数据进行读写操作，当需要保存数据时，将数据值赋值给变量，当需要读取数据值时，通过变量即可得到数据值。这些对于我们来说都不是问题，编写程序时一直这样用。但当内存中的一个数据赋值给不同类型变量（或以不同类型呈现）时，从变量值上可能反映出不同的结果，或者是不同类型变量之间赋值，赋值后变量值并不是你所希望的值。这些都需要我们对数据在内存中的表示有深入的理解，不能只看表面现象，而是要看到问题的本质。

1.2.1 字符类型和整数类型

1. 字符类型变量

在 C 中，如果把字符类型变量的值当成整数值看待，其操作和整数是一样的。在使用 char 类型变量时，对于同一个变量的值，以不同类型输出时，我们看到的结果是不同的。

例如：

```
char ch;
ch='a';
printf("%c,%d\n",ch,ch);
```

其输出结果是：

```
a  97
```

同一个变量以不同形式输出，其结果为什么会不同呢？是内部存储的值不同吗？回答是否定的，在变量 ch 所标识的内存中，存放的数据是同一个值，就是 0110 0001，当按字符输出时，是将其看成字符的 ASCII 码，字符'a'的 ASCII 码是 97，所以输出为字符'a'；同样是这个值，将其看成是一个整数，则就是正整数 97。这种操作比较常用，也比较容易理解。

2. 字符类型的符号

C 中的字符类型标识符包括：char 和 unsigned char，char 常用，而 unsigned char 怎样用呢？无符号字符是什么意思？当将字符类型变量按整数进行操作，或需要将字符类型转换成整数类型时，这时的转换规则就是根据 char 类型数据是有符号还是无符号而确定的。对于有符号类型，需要对符号位进行扩充，而对于无符号类型，则需要向高位补 0。同样，在进行比较运算时，是否有符号也是不同的，请看下面的例子：

```
char i;
for(i=5;i>=0;i--)
    printf("%u\n",i);
```

程序段循环体执行 6 次，结束循环。

若将“char i;”改为“unsigned char i;”，则代码如下：

```
unsigned char i;
```

```
for(i=5;i>=0;i--)
    printf("%u\n",i);
```

该循环体执行多少次？是 6 次吗？不是，而是无限多次。因为 i 为无符号字符，其最小值为 0，永远大于等于 0，所以字符是有符号的。

3. 整数的截断与扩充

而对于字符类型和整数类型变量之间的相互赋值，或者是不同的整数类型（如 short int、int、long int）变量之间的相互赋值，由于变量在内存中所占用的存储空间大小不同，赋值操作会进行截断或扩充处理，使得赋值后呈现出并不是你所希望的结果，这一点需要特别注意，例如下列程序段：

```
char a,b,c,d,e;
a=127;
b=-128;
c=129;
d=128;
e=-129;
printf("%d,%d,%d,%d,%d\n",a,b,c,d,e);
```

输出结果是：

```
127  -128  -127  -128  127
```

程序段的操作就是对变量赋值，然后再将变量的值打印出来，但从输出结果上看，有的变量的输出结果和其赋值是相同的，而有的却不同。具体来说，变量 a、b 的输出结果和所赋值相同，而变量 c、d、e 的输出结果和所赋值不同，为什么会出现这种问题？是产生错误了吗？不是，程序的结果是正确的，什么原因我们来分析一下。

大家知道，数值在内存中是用补码表示的。正数的补码就等于原码，而负数的补码，其符号位为 1，其余各位为该数绝对值的原码按位取反后整个数加 1。系统中使用补码的好处是可以将符号位和数据位统一处理，并且减法可按加法来处理，当两个用补码表示的数相加时，如果最高位（符号位）有进位，则进位直接舍弃。这样对于赋值表达式 a=127，由于是将一个正整数 127 赋给变量 a，127 的二进制表示为 0000 0000 0111 1111（为了描述方便我们用 16 位来表示），而正数在内存中是用原码表示，所以直接赋值，但变量 a 所占空间为 8 位，所以只能做截断处理，只保留低 8 位，赋值后结果为：

a

0	1	1	1	1	1	1	1

当按整数类型("%d")输出时，得到的结果自然是 127。

对于赋值表达式 b=-128，整型常量-128 的补码表示为 1111 1111 1000 0000，而变量 b 所占空间为 8 位，所以也是做截断处理，只取低 8 位，赋值后结果为：

b

1	0	0	0	0	0	0	0

当按整数类型("%d")输出时，得到的结果也是-128。

对于赋值表达式 c=129，由于正整数 129 的二进制表示为 0000 0000 1000 0001，同样取低 8 位赋值到变量 c 所标识的空间，其结果为：

c

1	0	0	0	0	0	0	1

此时，当按整数类型("%d")输出时，其高位为 1，所以解释为负数，去掉符号位后其余各位为原数的补码，再把它还原成原码，方法与转换成补码一样，各位取反后加 1，得到原码为 111 1111，即 127，所以得到的结果为-127。

对于赋值表达式 d=128，由于正整数 128 的二进制表示为 0000 0000 1000 0000，同样取低 8 位赋值到变量 d 所标识的空间，其结果为：

d

1	0	0	0	0	0	0	0

在内存中的表示和 b 相同，所以按整数类型("%d")输出时，结果也一样为-128。

赋值表达式 e=-129，由于负数-129 的补码高位为 1，而其他位为 129 的二进制表示 0000 0000 1000 0001 的各位取反加 1，得到补码为：1111 1111 0111 1111，同样取低 8 位赋值到变量 e 所标识的空间，其结果为：

e

0	1	1	1	1	1	1	1

所以按整数类型("%d")输出时，由于其高位为 0，所以结果是 127。

我们这里用的是 char 类型变量，对于其他类型变量实质上是一样的，因为 char 类型只占一个字节的空间，存放的数值比较小，说明问题比较方便。清楚了 char 类型后，推广到其他类型就不难了。

因此，在我们使用字符或整型变量时，在不同长度变量（如 short、int、long、char）之间赋值时，要根据其内部表示的特点对其进行操作，基本原则有两个：

- 将表示数范围小的类型变量值赋值给表示数范围大的类型变量时，高位做扩充处理，原数数值保持不变。
- 将表示数范围大的类型变量值赋值给表示数范围小的类型变量时，做截断处理，只取对应的低位，这时有些数有意义，有些数就没有意义了。

再看下面的程序：

```
void main()
{
    int a;
    char m,n;
    a=0x123456;
    m= a;
    n=(char)0x123486;
    printf("%d(%#x) , %d(%#x) , %d(%#x)\n",a,a,m,m,n,n);
}
```

程序的运行结果是：

```
1193046(0x123456)  86(0x56)  -122(0xffffff86)
```

由结果可以看出，无论是变量之间赋值，还是常量赋值给变量，对长度超过被赋值变量长度的数据，超出部分的高位被截掉，只保留了低位字节，此时变量中的数据值已经发生了变化，而且甚至连符号都可能改变，如上例中的 n 值已变为负数。

当被赋值变量长度大于数据长度时，无论是有符号数还是无符号数，数据值扩充后均保持不变。对于无符号数高位字节直接补 0；而对于有符号数扩充时是对数据的符号位进行扩充，即正数高位补 0，负数高位补 1，以保证扩充后数值的正确性。例如以下程序：

```
void main()
{
    int m,n;
    m=(char)0x82;                          //高位为 1
    n=0x82;
    printf("%d(%#x) , %d(%#x)\n",m,m,n,n);
}
```

运行结果为：

```
-126(0xffffff82)   130(0x82)
```

可以看出，对于数值 0x82，当作为有符号整数进行扩充时，高位补 0；而将其看作为单字节数据（(char)0x82），其最高位为 1，视为负数，在进行扩充时，高位补 1。

4. 整除运算问题

C 语言中的除“/”运算，当被除数和除数都是整数类型时，除运算为整除。在整数运算表达式中，不同运算顺序的等价表达式所产生的计算结果可能不同。例如，设 x、y、z 均为 int 类型变量，而且被赋了初值，表达式 x * y / z 所产生的结果与表达式 x * (y / z)的结果可能不一样。是否相同，取决于变量 x、y、z 的值，例如，当 x = 6，y = 5，z = 3 时，表达式 x * y / z 的值为 10，而表达式 x * (y / z)的值为 6。当 x = 5，y =6，z = 3 时，两个表达式的值