HZ BOOKS
华章科技

透析C语言中的核心概念、重要知识点、不易理解的知识点，以及容易被理解错的知识点，是修炼C程序设计能力的必读之作

實戰

牟海军 著

Advanced C Programming Language: Focal Points, Difficult Points and Doubtful Points

C语言进阶
重点、难点与疑点解析

机械工业出版社
China Machine Press

實戰

Advanced C Programming Language: Focal Points, Difficult Points and Doubtful Points

C语言进阶
重点、难点与疑点解析

牟海军 著

机械工业出版社
China Machine Press

C 语言是编程语言中的一朵奇葩，虽已垂垂老矣，但却屹立不倒，诞生了数十年，仍然是最流行的编程语言之一。C 语言看似简单，却不易吃透，想要运用好，更是需要积淀。本书是一本修炼 C 程序设计能力的进阶之作，它没有系统地去讲解 C 语言的语法和编程方法，而是只对 C 语言中不容易被初学者理解的重点、难点和疑点进行了细致而深入的解读，揭露了 C 语言中那些鲜为普通开发者所知的秘密，旨在让读者真正掌握 C 语言，从而编写出更高质量的 C 程序代码。

全书一共 11 章：第 1 章重点阐述了 C 语言中不易被理解的多个核心概念，很多初学者在理解这些概念时都会存在误区；第 2~8 章对预处理、选择结构和循环结构的程序设计、数组、指针、数据结构、函数和文件等知识点的核心问题和注意事项进行了讲解；第 9 章介绍了调试和异常处理的方法及注意事项；第 10 章对 C 语言中的若干容易让开发者误解误用的陷阱知识点进行了剖析；第 11 章则对所有程序员必须掌握的几种算法进行了详细的讲解；附录经验性地总结了如何养成良好的编码习惯，这对所有开发者都尤为重要。

图书在版编目（CIP）数据

C 语言进阶：重点、难点与疑点解析 / 牟海军著 . —北京：机械工业出版社，2012.6

ISBN 978-7-111-38861-6

I. C…　II. 牟…　III. C 语言—程序设计　IV. TP312

中国版本图书馆 CIP 数据核字（2012）第 131103 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：王海霞
北京京师印务有限公司印刷
2012 年 7 月第 1 版第 1 次印刷
186mm×240mm • 22.5 印张
标准书号：ISBN 978-7-111-38861-6
定价：59.00 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；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

前　言

为什么要写这本书

或许绝大多数人都有这样的经历，最初学习C语言的目的是为了应付考试，所以对于C语言只能算是一知半解。真正运用C语言进行编程时会出现很多问题，让人措手不及，这时才发现自己只能理解C语言的皮毛，虽能看懂简单的代码，却写不出程序来，对于那些稍微复杂的代码就更是望尘莫及了。

为了摆脱对C语言知其然不知其所以然的状态，本书将带领读者重启C语言学习之旅，这次不再是为了考试，而是出于真正的使用需要，所以有针对性地给出了C语言学习中的重点、难点与疑点解析，希望能够帮助更多的C语言爱好者走出困境，真正理解C语言，真正做到学以致用。

为了让读者能够真正地理解C语言学习中的重点、难点与疑点，以及体现本书学以致用的特色，全书没有采用枯燥的文字描述来讲解C语言相关的知识点，而是采用知识点与代码结合的方式，同时对于代码展开相应的分析，这就避免了部分读者在学习了相关知识点之后仍然不知道如何使用该知识点的弊端，使读者可以通过代码来加深对相关知识点的理解。

全书在结构安排上都是围绕C语言学习中的重点、难点与疑点进行讲解，如第1章并没有从讲解C语言中的基础知识点开始，而是先列举了C语言学习中易混淆的核心概念，使读者清晰地区分这些核心概念后再开始相应知识点的学习。本书对基础知识点也并非概念性地讲解，而是重点讲解了使用中的要点，同时重点讲解了C语言中的一些调试和异常处理的方

法，以及误区和陷阱知识点。最后一章讲解了编程中必须掌握的一些常用算法。总之，本书能够使读者在现有基础上进一步提高自己的C语言编程能力，更清晰地认识和理解C语言。

本书读者对象

本书适合以下读者：

- C语言爱好者
- 嵌入式开发人员
- 初、中级C语言程序员
- 参加C语言培训的学员

如何阅读本书

本书共11章，第1章主要针对C语言学习中一些容易混淆的核心概念进行具体讲解，内容跨度比较大，初学者学起来可能有些吃力，所以建议在遇到不懂的知识点时暂时跳过，待学习了后面的相关知识点后再进行相应的学习；第2~8章有针对性地讲解了C语言中的相应知识点，同时有针对性地对其中的要点部分进行具体讲解，读者可以通过这几章的学习夯实每个知识点的基础；第9章重点讲解了在C语言编程中进行调试和异常处理的一些常见方法和技巧；第10章重点讲解了C语言编程中的一些陷阱知识点，通过本章的学习读者可以知道如何在以后编程时绕开陷阱；第11章讲解了一些编程中的常用算法，这是编程中必然会遇到的，因此读者有必要掌握这些常见的算法。

最后在附录部分给出了养成良好编程习惯的建议。本书针对每个知识点都提供了相应的代码，建议读者在学习的过程中自己动手编写，这样才会发现自己在C语言学习方面的缺陷，进而快速提升自己的编程能力。

勘误和支持

除署名作者外，参与本书材料整理和代码测试工作的还有项俊、马晓路、刘倩、罗艳、胡开云、余路、张涛、张晓咏、时翔、秦萤雪等。由于作者的水平有限，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。读者遇到任何问题都可以发邮件到bigloomy@hotmail.com，我会尽力为读者提供最满意的解答。书中的全部源文件除可以从华章网站（www.hzbook.com）下载外，还可以发邮件向我索取。如果你有更多的宝贵意见，也欢迎发邮件与我交流，期待得到你们的真挚反馈。

致谢

本书得以出版要感谢很多人，首先要感谢我的导师侯建华教授，无论是在科研还是平时

的学习和生活中，都得到您严格的指导和无微不至的关怀，在此向您表示最真诚的敬意和衷心的感谢！

其次要感谢我的好朋友们，他们是刘倩、马晓路、胡开云、时翔、张晓咏、余路、张涛，有你们的陪伴，我每天都过得很开心，感谢你们在生活中给予我的关心和体贴。同时也感谢实验室的项俊、梁娟、左坚、罗艳、严明君、李思，谢谢你们平时给予的帮助。

感谢机械工业出版社华章公司的编辑杨福川和姜影，你们在这一年多的时间中始终支持我的写作，你们的鼓励和帮助指引我顺利地完成全部书稿。

最后要感谢我的家人，没有你们的鼓励和支持，就没有我今天的成绩。在此要特别感谢我的父亲，您多年来对我的悉心教导，我都铭记在心。

谨以此书献给众多热爱 C 语言的朋友们！

牟海军（bigloomy）

2012 年 4 月于中国武汉

目　录

第1章 必须厘清的核心概念

人或多或少都有一点惰性和急功近利，我就是这样，在一开始学习编程的时候不喜欢阅读那些枯燥的文字，喜欢直接去阅读代码。但是渐渐地，我发现一个问题，那就是编程时经常会犯一些低级的错误。通过总结才明白，这些错误源于自己对C语言中的基本概念一知半解，知其然，不知其所以然，发现问题后才意识到那些枯燥的文字对掌握并熟练使用C语言非常重要。为了让读者少走一些弯路，本书的第1章先来介绍C语言中的核心概念。

开始本章的学习之前，先向读者交代一下，由于本章涉及的知识范围较广，有些初学者理解起来会有些吃力，因此建议读者有选择地阅读，遇到陌生知识点可以暂时跳过，待学习了后面章节的内容后再回过头来阅读这一章的相关内容。当然，学习代码的最佳方法是动手，所以本章在讲解C语言的一些基本概念的同时，为了便于读者理解，有针对性地列举了一些代码，读者也可以通过这些代码来验证所学的概念，体会学习的乐趣，以避免单纯通过阅读文字来枯燥地学习概念。

1.1 堆栈

不少人可能对堆栈的概念并不清楚，甚至部分从事计算机专业的人也没有理解通常所说的堆栈其实是两种数据结构。那么究竟什么是堆，什么又是栈呢？接下来，我们就来看看它们各自的概念。

栈，是硬件，主要作用表现为一种数据结构，是只能在一端插入和删除数据的特殊线性表。允许进行插入和删除操作的一端称为栈顶，另一端为栈底。栈按照后进先出的原则存储数据，最先进入的数据被压入栈底，最后进入的数据在栈顶，需要读数据时从栈顶开始弹出数据。栈底固定，而栈顶浮动。栈中元素个数为零时称为空栈。插入一般称为进栈（push），删除则称为出栈（pop）。 栈也被称为先进后出表，在函数调用的时候用于存储断点，在递归时也要用到栈。

在计算机系统中，栈则是一个具有以上属性的动态内存区域。程序可以将数据压入栈中，也可以将数据从栈顶弹出。在i386机器中，栈顶由称为esp的寄存器进行定位。压栈的操作使栈顶的地址减小，弹出的操作使栈顶的地址增大。

栈在程序的运行中有着举足轻重的作用。最重要的是，栈保存了一个函数调用时所需要的维护信息，这常常被称为堆栈帧。栈一般包含以下两方面的信息：

1）函数的返回地址和参数。

2）临时变量：包括函数的非静态局部变量及编译器自动生成的其他临时变量。

堆，是一种动态存储结构，实际上就是数据段中的自由存储区，它是C语言中使用的一种名称，常常用于存储、分配动态数据。堆中存入的数据地址向增加方向变动。堆可以不断进行分配直到没有堆空间为止，也可以随时进行释放、再分配，不存在顺序问题。

堆内存的分配常通过malloc()、calloc()、realloc()三个函数来实现。而堆内存的释放则使用free()函数。

堆和栈在使用时“生长”方向相反，栈向低地址方向“生长”，而堆向高地址方向“生长”。

我们对于堆的理解可能要直观些，而仅从概念上理解栈会让读者感到有些模糊。为了加深读者对于栈的理解，我们来看一个 C 语言题目。这个题目要求在不传递参数的情况下，在 print() 函数中打印出 main() 函数中 arr 数组中的各个元素。

```
#include <stdio.h>

void print()
{
    //填充代码
}

int main()
{
   int a=1;
   int b=2;
   char c='c';
   int arr[]={11,12,13,14,15,16,17};

   print();

   return 0;
}
```

注意　如无特殊说明，本书代码均通过 VC++6.0 来编译运行。

看看上面的代码和相关要求，可能会让很多读者束手无策，如果能联系前面的知识点，就应该想到用栈。那么我们该如何来解决问题呢？先别急，在讲解之前，我们先来回顾几个知识点。

1）push 操作先移动栈顶指针，之后将信息入栈。

2）esp 为堆栈指针，栈顶由 esp 寄存器来定位。压栈的操作使栈顶的地址减小，弹出的操作使栈顶的地址增大。

3）ebp 是 32 位的 bp，是基址指针。bp 为基指针寄存器，用它可直接存取堆栈中的数据，它在调用函数时保存 esp，以便函数结束时可以正确返回。

4）默认的函数内部变量的压栈操作为：从上到下、从左向右，采用 4 字节对齐。数组压栈方法略有不同，即从最后一个元素开始，直到起始元素为止，即采用从右向左的方法压栈。

现在看一下以上代码的汇编代码，在 main() 函数的 return 语句处按 F9 键设置一个断点，然后按 F5 键运行代码，代码运行到断点时把光标移动到断点处，右击选择 Go to Disassembly，就可以看到上面那段代码的汇编代码了。我们发现，在 main() 函数和 print() 函数的开头都有如下两句汇编指令：

```
push        ebp
mov         ebp,esp
```

为了使读者易于理解，在此通过图 1-1 来分析说明。根据图上的标注，函数开头部分的

第一个 push 指令的操作步骤是，首先移动栈顶指针 esp，然后将 ebp 内容压栈，注意此时压栈的 ebp 的值为上一个函数的 esp 的值，而 esp 恰好就是上一个函数的栈底，所以每个函数一开始的 push 指令就是保存上一个函数的栈底。那么接下来的 mov 指令有什么作用呢？由于 esp 是当前的栈顶指针，所以该指令的作用就是保存当前栈顶指针的值。由此就可以分析出，ebp 存放的是此刻栈顶的地址，就是说，ebp 是一个指针，指向栈顶，而栈顶存放的数据其实是上一个函数的 ebp 的值，即上一个函数的栈底。

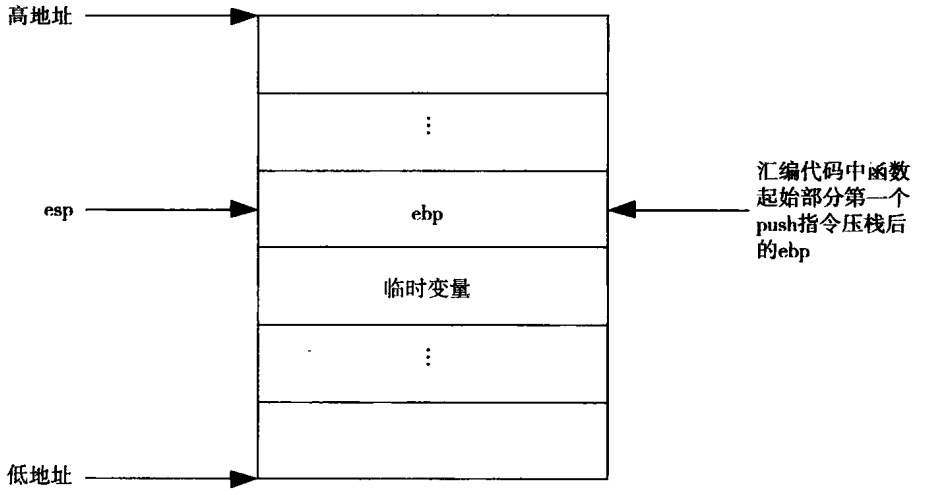

图 1-1　函数调用过程中的压栈流程

通过上面的分析可知，ebp 压栈后，接着就是函数中临时变量的压栈操作，由此可知，我们只需要在 print() 函数中得到 main() 函数的栈底，就可以取出数组中的每个元素了，看看下面的实现方法。

```
#include <stdio.h>

void print()
{
   unsigned int _ebp;
    __asm{
          mov _ebp,ebp
   }
    int *p=(int *)(*(int *)_ebp-4-4-4-7*4);
    for(int i=0;i<7;i++)
          printf("%d\t",p[i]);
}

int main()
{
    int a=1;
    int b=2;
    char c='a';
    int arr[]={11,12,13,14,15,16,17};
```

```
    print();

    return 0;
}
```

运行结果为：

```
11      12      13      14      15      16      17
```

在没有传递任何参数的情况下，成功地在 print() 函数中打印出了 main() 函数中 arr 数组内的每个元素。现在来看看上面代码的实现方法，在 print() 函数中定义了一个 _ebp 无符号整型变量，通过 VC++ 6.0 内嵌汇编把 ebp 的值保持到 _ebp 中，按照上面的分析，可以将在函数 print() 中通过这条内嵌汇编语句得到的 ebp 看成一个指针，指针所指向的单元存放的就是 print() 函数的上一个函数的栈底，在此是 main() 函数的栈底。知道了 _ebp 的作用后，我们来分析下代码，通过 (int*)_ebp 将 _ebp 转换为一个整型指针，然后通过 *(int*)_ebp 即可得到 main() 函数的栈底地址。由于栈的压栈操作是从上到下、从右到左的，所以 main() 函数中的变量 a 先压栈，然后是 b、c，最后是 arr 数组，数组的压栈顺序是从右到左。通过“int *p=(int *)(*(int *)_ebp-4-4-4-7*4);”即可得到数组元素的首地址。接下来，根据首地址就可以取出数组中的每个元素了。有的读者可能会有一个疑惑，main() 函数中有一个字符型变量，是不是在求数组元素的首地址时应该把其中的减 4 改为减 1 呢？因为它只占用了一个字节！即将“int *p=(int *)(*(int *)_ebp-4-4-4-7*4);”修改为“int *p=(int *)(*(int *)_ebp-4-4-1-7*4)”。我们暂且不说其对与错，先来看看修改后的运行结果：

```
3072    3328    3584    3840    4096    4352    -859021056
```

我们发现这样的运行结果是错误的，为什么呢？细心的读者可能发现了本章一开始回顾的知识点中有一点是很重要的，那就是压栈操作为 4 字节对齐。所以这里必须减 4，而不是减 1。

通过上面的分析，希望读者能够对栈有更加深入的理解，而对于堆的使用我们会在后续章节详细讲解。

1.2　全局变量和局部变量

全局变量，也称外部变量，在函数体外定义，不是哪一个函数所特有的。全局变量又可以分为外部全局变量和静态全局变量，它们之间的最大区别在于，使用 static 存储类别的全局变量只能在被定义的源程序文件中使用，而使用 extern 存储类别的全局变量不仅可以在被定义的源程序文件中使用，还可以被其他源文件中的函数引用。如果要在函数中使用全局变量，那么通常需要作全局变量说明。只有在函数内经过说明的全局变量才能使用。但在一个函数之前定义的全局变量，在该函数内使用可不再加以说明。 例如：

```
#include <stdio.h>
```

```
int a=0;

void print(void)
{
   printf("global variable a=%d\n", a);
}

int main(void)
{
   print();

   return 0;
}
```

因为全局变量 a 在 print() 函数之前定义，所以在 print() 函数中使用 a 时无需说明，但是下面的代码在运行时会出错。

```
#include <stdio.h>

void print(void)
{
   printf("global variable a=%d\n", a);
}

int a=0;

int main(void)
{
   print();

   return 0;
}
```

因为全局变量 a 的定义出现在 print() 函数之后，所以在 print() 函数中使用 a 时需要说明，应该在 print() 函数的 printf 语句上面加一句“extern int a;”来说明，这样才可以使用全局变量。

局部变量是相对于全局变量而言的，即在函数中定义的变量称为局部变量。当然，由于形参相当于在函数中定义的变量，所以形参也是一种局部变量。我们可以通过图 1-2 来说明全局变量和局部变量的定义区域。

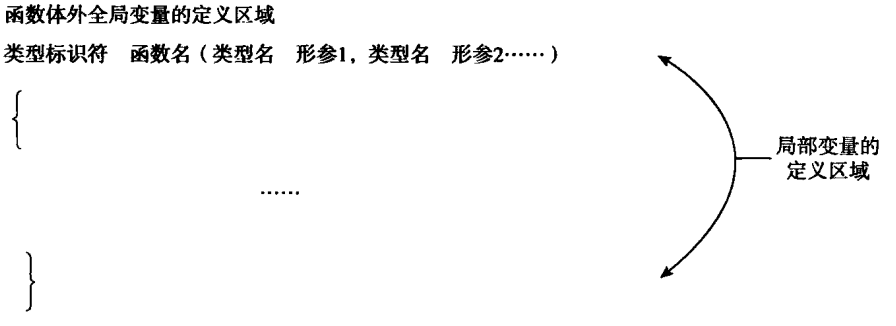

图 1-2 全局变量和局部变量的定义区域