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

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
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。

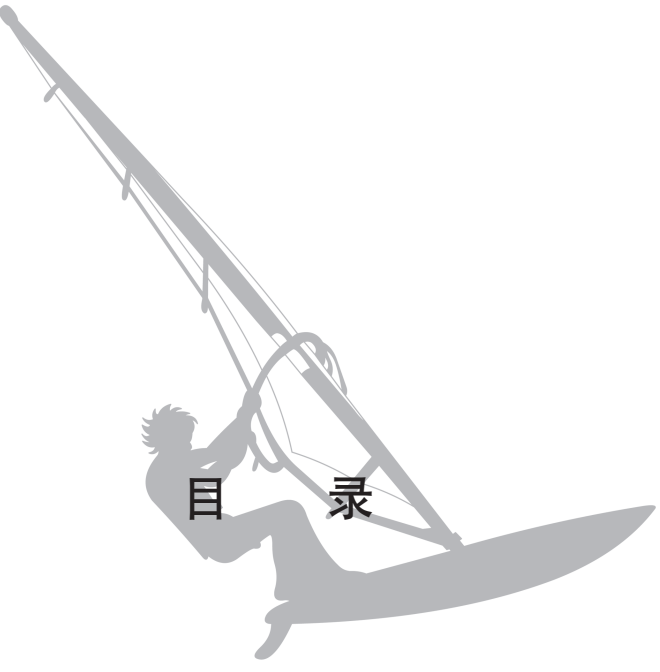
感谢机械工业出版社华章公司的编辑杨福川和姜影，你们在这一年多的时间中始终支持我的写作，你们的鼓励和帮助指引我顺利地完成全部书稿。

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。

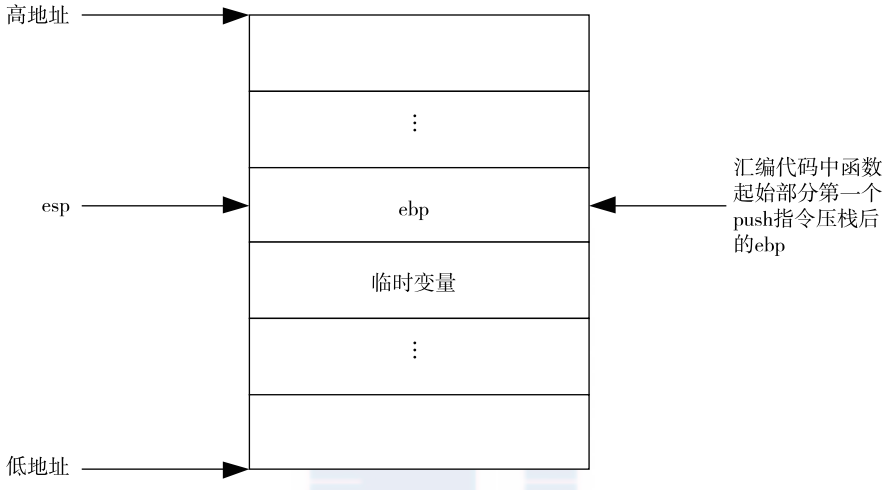


图 1-1　函数调用过程中的压栈流程

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。

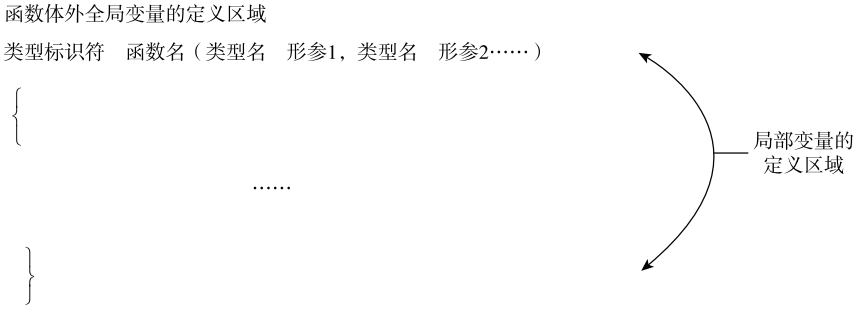


图 1-2　全局变量和局部变量的定义区域

1.3　生存期和作用域

1.3.1　生存期

不少人对于生存期有着一种错误的理解，认为变量离开了它的作用域，其生存期就结束了。产生这种误解的原因，是对于生存期的概念理解不深刻。所谓的生存期，其实是指变量占用内存或者寄存器的时长。根据变量存储类别的不同，在编译的时候，变量将被存放到动态存储区或静态存储区中，所以其生存期是由声明时的存储类别所决定的。

在讲解存储类别和相应的变量之前，我们先来看看静态存储区和动态存储区。

- 静态存储区，存放全局变量和静态变量，在执行程序前分配存储空间，占据固定的存储单元。
- 动态存储区，存放的是函数里的局部变量、函数的返回值、形参等，它在函数被执行的过程中进行动态分配，在执行完该函数时自动释放。由于这种分配和释放都是每次执行到函数时进行的，因此前后两次调用同一个函数，其临时变量分配到的地址可能是不同的。

了解了动态存储区和静态存储区之后，接下来介绍存储类别和相应的变量。

（1）自动（auto）

非静态变量的局部变量即为自动变量，其类型说明符为 auto，在 C 语言中，将函数内没有存储类别说明的变量均视为自动变量，即自动变量可以省去说明符 auto。如：

```
void print()
{
        int a;
}
```

等价于

```
void print()
{
        auto int a;
}
```

（2）寄存器（register）

指定了 register 存储类别的变量即为寄存器变量。使用寄存器变量是为了提高执行效率，因为频繁地从内存单元存取变量相比于从寄存器中存取变量需要消耗更多的时间，所以使用 register 声明的寄存器类型的变量存放在寄存器中，不会占用内存单元，可以提高程序的执行效率。值得注意的一点是，只有局部变量才可以定义成寄存器变量。为了加深读者的印象，我们通过下面两段代码来对比不使用 register 和使用 register 的程序执行效率。

注意　*以下两段代码均在 Linux 环境下采用 gcc 编译运行。*

不用 register 的程序如下：

```
#include <stdio.h>
#include <sys/time.h>

int main(int argc, char * argv[])
{
    struct timeval start,end;
    gettimeofday( &start, NULL );  /* 测试起始时间 */
    double timeuse;
    double sum;
    int j,k;
    for(j=0;j<1000000000;j++)
          for(k=0;k<10;k++)
               sum=sum+1.0;
    gettimeofday( &end, NULL );   /* 测试终止时间 */
    timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec ;
    timeuse /= 1000000;
    printf("运行时间为：%f\n",timeuse);

    return 0;

}
```

不用 register 的程序的运行结果：

```
root@ubuntu:/home# ./ce
运行时间为：35.608037
```

用 register 的程序如下：

```
#include <stdio.h>
#include <sys/time.h>

int main(int argc, char * argv[])
{
    struct timeval start,end;
    gettimeofday( &start, NULL );  /* 测试起始时间 */
    double timeuse;
    register double sum;
    register int j,k;
    for(j=0;j<1000000000;j++)
          for(k=0;k<10;k++)
              sum=sum+1.0;
    gettimeofday( &end, NULL );   /* 测试终止时间 */
    timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec - start.tv_usec ;
    timeuse /= 1000000;
    printf("运行时间为：%f\n",timeuse);

    return 0;

}
```

用 register 的程序的运行结果：

```
root@ubuntu:/home# ./ce
运行时间为：9.678347
```

对比上面的两个运行结果，我们发现，使用了 register 的程序执行速度提高了近 3 倍，但是读者要注意，虽然可以使用 register 来提高程序的执行速度，但是也不能大量使用 register，因为寄存器的数目是有限的。

（3）静态（static）

关于静态变量，值得注意的一点是，它的生存期是从程序开始运行到程序运行结束。静态变量不属于动态存储，是静态存储。

静态局部变量的生存期虽然是从程序开始运行到程序运行结束，但是它的作用域并不会因此而改变，而且仍然与其作为自动变量的作用域相同。静态全局变量的特点是，它只能在被定义的源程序文件中使用，即它只能被本源程序文件的函数调用，而不能被其他的源程序文件中的函数调用。

静态局部变量和静态全局变量的定义形式都是在数据类型前加上一个静态存储定义符 static。但是值得注意的是，两者的初始化方式不同，静态局部变量在它所在的函数被执行时初始化，之后再次执行该函数时，该静态局部变量不再进行初始化，其中保留的是上一次的运行结果；而静态全局变量的初始化是在执行 main() 函数之前完成的，其静态全局变量的当前值由最近一次对它的赋值操作决定。

在此，我们重点来看看静态局部变量的使用。

```
#include <stdio.h>

void print(void)
{
   static int a=0;
   printf("静态局部变量 a=%d\n", a++);
}

int main(void)
{
   print();
   print();

   return 0;
}
```

运行结果：

```
静态局部变量 a=0
静态局部变量 a=1
```

分析运行结果可以得知，静态局部变量在初始化以后，再次执行该函数时静态局部变量保存的是上一次的运行结果。

（4）外部（extern）

外部存储类别定义方式为在全局变量类型前面加上关键字 extern，如果没有指定全局变

量的存储类别，则默认为 extern。

1.3.2 作用域

不少人在编程中并不重视作用域的问题，实际上，它是 C 语言程序设计中的一个要点。通常来说，一段程序代码中所用到的名字并不总是有效或可用的，而限定这个名字的可用性的代码范围就是这个名字的作用域。现在，我们通过如下代码来分析作用域。

```
#include <stdio.h>

void fun()
{
   int a=3,b;
   printf("fun() 函数里面的 a 值为：%d\n",a);

   return ;
}

int main(void)
{
   int a=0,b;
   {
           int a=1;
           printf("main() 函数里面被大括号封装的 a 值为：%d\n",a);
   }
   fun();
   printf("main() 函数里面的 a 值为：%d\n",a);

   return 0;
}
```

运行结果：

```
main() 函数里面被大括号封装的 a 值为：1
fun() 函数里面的 a 值为：3
main() 函数里面的 a 值为：0
```

分析上面的代码后发现，在 main() 函数中定义的变量 a 和 b 仍然可以在 fun() 函数中定义和使用，这是因为局部变量的作用域仅在该函数中有效，所以可以在一个函数中定义与另一个函数中的变量同名的变量。再看 main() 函数，我们发现，居然可以在 main() 函数中定义两次变量 a。这是由于在函数体内可以进一步限制变量的作用域，通常的方法是采用大括号封装来限制变量的作用域，这就可以再次使用 a 来定义变量了。但是，值得注意的是，此时 a 的作用域为大括号的封装范围，在大括号的封装范围之外再次使用 printf 打印语句打印 a 的值时，a 的值为大括号封装外面的 a 值，所以，当在函数体中定义和使用变量名的时候一定要注意其作用域。图 1-3 说明了同名变量的不同作用域的处理方法。

如果在一个区域中出现了同名的变量，那么以在该区域有效且定义最接近该区域的变量为准。

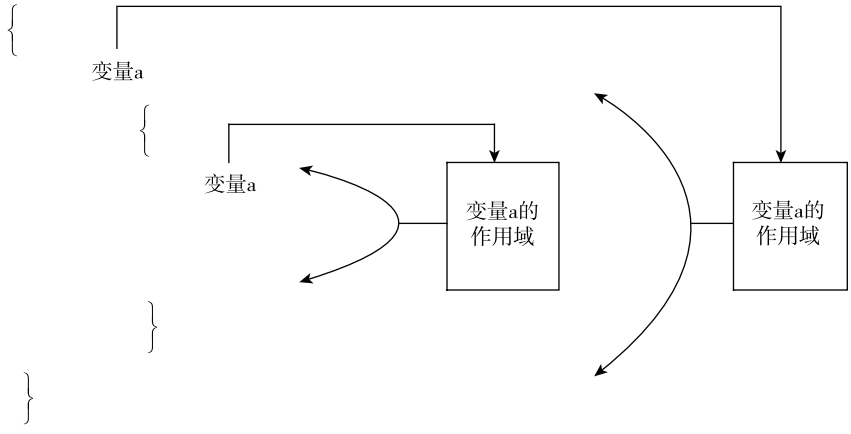


图 1-3　同名变量的不同作用域

1.4　内部函数和外部函数

前面讲解了变量的作用域，那么函数是否有作用域呢？回答是肯定的，函数同样也存在作用域。如果在一个源文件中定义的函数只能被该文件中的函数所调用，而不能被同一程序其他文件中的函数调用，那么我们称之为内部函数，其定义的一般形式为：

```
static 函数类型　函数名（参数表）
```

如果一个函数既可以被同一个源文件中的函数调用，又可以被同一程序其他文件中的函数调用，我们称之为外部函数。如果定义函数时没有加关键字 static 或者 extern，那么这种函数也是外部函数。其定义的一般形式为：

```
extern 函数类型 函数名（参数表）
```

从上面的描述中可以看出，外部函数和内部函数之间的最大区别莫过于它们的作用范围不同，内部函数的作用范围是它所在的源文件，而外部函数的作用范围则不局限于它所在的源文件。接下来看看下面的代码，通过对下面的代码进行分析来加深对内部函数和外部函数的理解。

```
/******* 以下代码存放于 file.h 中 *******/
#include <stdio.h>

typedef struct _stu
{
   char name[10];
   int score;
}stu;
/******* 以下代码存放于 file1.cpp 中 *******/
#include "file.h"
```

```
static void input(stu student[],int n)
{
   int i;
   for(i=0;i<n;i++)
   {
          printf(" 请输入学生的姓名：");
          scanf("%s",&student[i].name);
          printf(" 请输入学生的总成绩：");
          scanf("%d",&student[i].score);
   }

   return ;
}

int main(void)
{
   stu student[4];
   extern void sort(stu student[],int n);
   extern void bubble_sort(stu student[],int n);
   extern void print(stu student[],int n);
   input(student,4);

   sort(student,4);
   print(student,4);

   bubble_sort(student,4);
   print(student,4);

   return 0;
}
/******* 以下代码存放于 file2.cpp 中 *******/
#include "file.h"

extern void sort(stu student[],int n)
{
   int i,j,k;
   stu temp;

   for(i=0;i<n-1;i++)
   {
          k=i;
          for(j=i+1;j<n;j++)
          {
                 if (student[j].score<student[k].score)
                        k=j ;
          }
          if(k!=i)
          {
                 temp=student[i];
                 student[i]=student[k];
                 student[k]=temp;
          }
   }

   printf(" 使用选择法升序排列的结果为：\n");
```

```
    return ;
}
/******* 以下代码存放于 file3.cpp 中 *******/
#include "file.h"

void bubble_sort(stu student[],int n)
{
    int i,j,flag;
    stu temp;

    for(i = 0; i < n-1; i++)
    {
            flag = 1;

            for(j = 0; j < n-i-1; j++)
            {
                    if(student[j].score < student[j+1].score)
                    {
                            temp = student[j];
                            student[j] = student[j+1];
                            student[j+1] = temp;
                            flag = 0;
                    }
            }
            if(1 == flag)
                    break;
    }

    printf(" 使用冒泡法降序排列的结果为：\n");

    return;
}
/******* 以下代码存放于 file4.cpp 中 *******/
#include "file.h"

void print(stu student[],int n)
{
    int j;
    for(j=0;j<n;j++)
    {
            printf(" 学生 %s 的总成绩为：%d\n",student[j].name,student[j].score);
    }

    return ;
}
```

总共有 5 个文件：1 个头文件 file.h，4 个源文件 file1.cpp、file2.cpp、file3.cpp 和 file4.cpp。看看运行结果：

```
请输入学生的姓名：小王
请输入学生的总成绩：32
请输入学生的姓名：小李
请输入学生的总成绩：56
```

```
请输入学生的姓名：小刚
请输入学生的总成绩：34
请输入学生的姓名：小张
请输入学生的总成绩：43
使用选择法升序排列的结果为：
学生小王的总成绩为：32
学生小刚的总成绩为：34
学生小张的总成绩为：43
学生小李的总成绩为：56
使用冒泡法降序排列的结果为：
学生小李的总成绩为：56
学生小张的总成绩为：43
学生小刚的总成绩为：34
学生小王的总成绩为：32
```

分析上面的代码，由于每个源文件都用到了定义的结构体，所以在此把结构体放到一个头文件中去。在上面的代码中，使用 static 定义了一个内部函数“static void input(stu student[],int n)”，其功能是输入学生的相关信息；使用 extern 关键字定义了一个外部函数“extern void sort(stu student[],int n)”，其功能是选择法升序排序；不使用任何关键字定义了一个外部函数“void bubble_sort(stu student[],int n)”，其功能为冒泡法降序排序。同时，因为对两种排序方式的结果都进行了打印，所以在 file4.cpp 文件中编写了一个外部函数“void print(stu student[],int n)”，然后在 main() 函数中对进行了排序的结果调用 print() 函数进行打印。读者在使用 VC++ 6.0 进行编译的过程中需要在建立的工程中添加上面 4 个源文件和 1 个头文件。

我们发现，使用内部函数的优点是：不同的人编写不同的函数时，不用担心自己定义的函数是否会与其他文件中的函数同名，因为作用域的关系，同名也不会产生影响。所以，在编程的过程中，对于那些只需要在一个源文件中使用的函数，我们要养成加上 static 的习惯。当然，对于那些不仅仅在一个源文件中使用的函数，我们需要将其定义为外部函数。

对于内部函数和外部函数的讲解到这里就结束了，相信读者应该掌握了内部函数和外部函数的使用。

1.5 指针变量

懂得 C 语言的人都知道，C 语言之所以强大且具有自由性，主要体现在对指针的灵活运用上。因此，说指针是 C 语言的灵魂一点都不为过。既然指针如此重要，那么指针究竟是什么呢？在回答这个问题之前，我们先通过下面一段代码来看看指针的使用。

```
#include <stdio.h>

int main()
{
   int a =2;
   int *pa;
   char b='t';
   char *pb;
```

```
    pa=&a;
    pb=&b;

    printf("整型指针 pa 占用内存大小为 :%d 字节 \n", sizeof(pa));
    printf("整型指针 pb 占用内存大小为 :%d 字节 \n", sizeof(pb));

    printf("整型变量 a 的地址为 :\t%d\n", &a);
    printf("整型变量 b 的地址为 :\t%d\n", &b);

    printf("整型指针 pa 的值为 :\t%d\n", pa);
    printf("整型指针 pb 的值为 :\t%d\n", pb);

    printf("整型指针 pa+1 的值为 :\t%d\n", pa+1);
    printf("整型指针 pb+1 的值为 :\t%d\n", pb+1);

        return 0;
}
```

运行结果：

```
整型指针 pa 占用内存大小为 :4 字节
整型指针 pb 占用内存大小为 :4 字节

整型变量 a 的地址为 :       1245056
整型变量 b 的地址为 :       1245055

整型指针 pa 的值为 :        1245056
整型指针 pb 的值为 :        1245055

整型指针 pa+1 的值为 :      1245060
整型指针 pb+1 的值为 :      1245056
```

现在逐一分析上面的运行结果，为什么指针变量的大小都是 4 字节呢？这是因为我们使用的是 32 位的计算机，内存地址都是 32 位的整数，而指针变量的实质就是内存地址。再看看整型指针变量 pa 和字符型指针变量 pb，它们分别用于存放整型变量 a 和字符型变量 b 的地址，在之后使用 printf 打印语句打印出来的结果中也可以看出，pa 和 pb 中存放的分别是整型变量 a 和字符型变量 b 的地址。那么，什么是指针变量呢？存放地址的变量称为指针变量。指针变量是一种特殊的变量，它不同于一般的变量，一般变量存放的是数据本身，而指针变量存放的是地址。再看两种类型的指针的运算结果，对比运算前后的结果发现，两种类型指针加 1 后的变化值并不相同，如果按照一般的加法来理解，加 1 以后它们的值都应该增加 1，为什么整型指针的值增加的是 4，而字符型指针增加的是 1 呢？下面用图 1-4 来展示不同类型的变量在内存中是如何分配存储区域的。

在图 1-4 中，字符变量在内存中占用一个字节的大小，而整型变量在内存中占用 4 个字节的大小，但是我们发现，指针变量 pa 指向变量 a 的地址时取的是存储变量 a 在内存中的最小存储地址，而所指向的却是占用 4 个字节大小的内存区域，所以从这里可以看出，我们不能简简单单地将指针理解为地址，而应该把指针理解为指向一块内存区域的起始地址，指向区域的大小视所指变量的类型而定。而指针变量与一般变量的区别就在于，指针变量存放的是地址，看看下面一段代码。

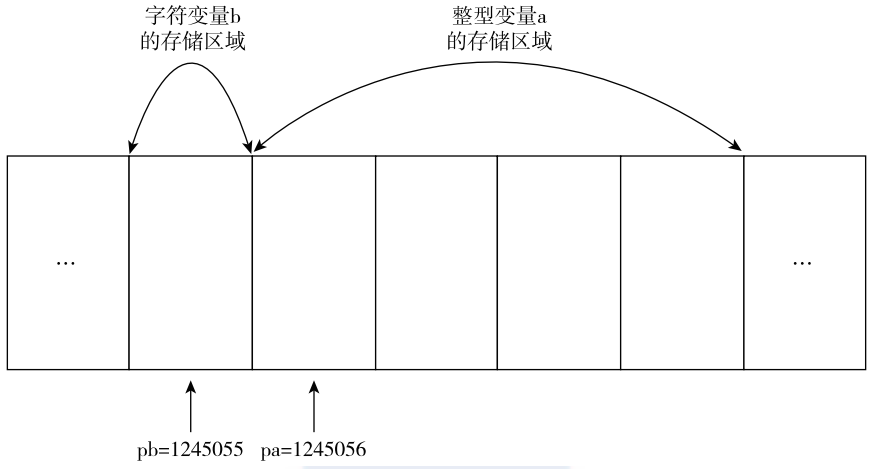


图 1-4　不同类型的指针变量在内存中的存储区域分配

```
#include <stdio.h>

void main(int argc,char *argv[])
{
   int a[10];

   printf("a 的值为 :\t%d\n",a);
   printf("&a 的值为 :\t%d\n\n",&a);

   printf("a+1 的值为 :\t%d\n",a+1);
   printf("&a+1 的值为 :\t%d\n",&a+1);

   return ;
}
```

运行结果：

```
a 的值为 :          1245020
&a 的值为 :         1245020

a+1 的值为 :        1245024
&a+1 的值为 :       1245060
```

很多读者看了上面的运行结果会觉得不可思议，a 和 &a 都表示数组 a 的起始地址，打印出来的结果相同是显而易见的，为什么 a+1 和 & a+1 打印出来的结果却相差如此之大呢？回想前面讲述的内容，出现这种情况的原因是它们是不同类型的指针变量。代码中的 a 其实相当于一个整型指针变量，所以它加 1 的结果就和之前的分析一样，那么 &a 又意味着什么呢？别急，我们先把“int a[10];”变形为“int *（&a）[10];”，这样就可以很直观地看出来，&a 就相当于指向一个 int [10] 类型的指针变量，于是上面的运行结果就很容易理解了，a 到 a+1 的变化就是它指向的变量所占用的内存单元的大小 4 字节，而 &a 到 &a+1 的变化就是它指向的变量所占用的内存单元的大小 4×10 字节 =40 字节。

通过前面两段代码的分析，读者对指针变量应该有了更进一步的认识，但是我们不可能就用这么一点内容来讲解指针，后面我们会通过一章的内容来具体讲解指针，这里只是想让读者对于指针变量有一个初步的认识。

1.6　指针数组和数组指针

对于指针数组和数组指针，单从字面上似乎很难分清它们是什么，先来看看指针数组和数组指针各自的定义形式。

指针数组的定义形式为：

```
类型名 * 数组名 [ 数组长度 ];
```

如：

```
int *p[8];
```

数组指针的定义形式为：

```
类型名 (* 指针名 )[ 数组长度 ];
```

如：

```
int (*p)[8];
```

现在来分析上述两种定义形式，通过“int *p[8];”这条定义语句可以定义一个指针数组。因为优先级的关系，所以 p 先与 [] 结合，说明 p 是一个数组，然后再与 * 结合说明数组 p 的元素是指向整型数据的指针。元素分别为 p[0]，p[1]，p[2]，...，p[7]，相当于定义了 8 个整型指针变量，用于存放地址单元，在此，p 就是数组元素为指针的数组，本质为数组。如果使用的定义方式为“int (*p)[8];”，p 先与 * 号结合，形成一个指针，该指针指向的是有 8 个整型元素数组，p 即为指向数组首元素地址的指针，其本质为指针。介绍了指针数组和数组指针的含义，接下来，我们通过下面一段代码来看看指针数组和数组指针如何访问二维数组。

```
#include <stdio.h>

void main(int argc,char *argv[])
{
   int arr[4][4]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
   int (*p1)[4];
   int *p2[4];
   int i,j,k;

   p1=arr;

   printf(" 使用数组指针的方式访问二维数组 arr\n");

   for(i=0;i<4;i++)
   {
          for(j=0;j<4;j++)
```

```
            {
                    printf("arr[%d][%d]=%d\t",i,j,*(*(p1+i)+j));
            }
            printf("\n");
    }

    printf("\n 使用指针数组的方式访问二维数组 arr\n");

    for(k=0;k<4;k++)
            p2[k]=arr[k];
    for(i=0;i<4;i++)
    {
            for(j=0;j<4;j++)
            {
                    printf("arr[%d][%d]=%d\t",i,j,*(p2[i]+j));
            }
            printf("\n");
    }

    return ;
}
```

运行结果：

```
使用数组指针的方式访问二维数组 arr
arr[0][0]=0     arr[0][1]=1     arr[0][2]=2     arr[0][3]=3
arr[1][0]=4     arr[1][1]=5     arr[1][2]=6     arr[1][3]=7
arr[2][0]=8     arr[2][1]=9     arr[2][2]=10    arr[2][3]=11
arr[3][0]=12    arr[3][1]=13    arr[3][2]=14    arr[3][3]=15

使用指针数组的方式访问二维数组 arr
arr[0][0]=0     arr[0][1]=1     arr[0][2]=2     arr[0][3]=3
arr[1][0]=4     arr[1][1]=5     arr[1][2]=6     arr[1][3]=7
arr[2][0]=8     arr[2][1]=9     arr[2][2]=10    arr[2][3]=11
arr[3][0]=12    arr[3][1]=13    arr[3][2]=14    arr[3][3]=15
```

我们成功地使用数组指针和指针数组的方式访问了二维数组，在分析它们各自的访问方式之前，先通过图 1-5 了解二维数组中元素的存放方式。

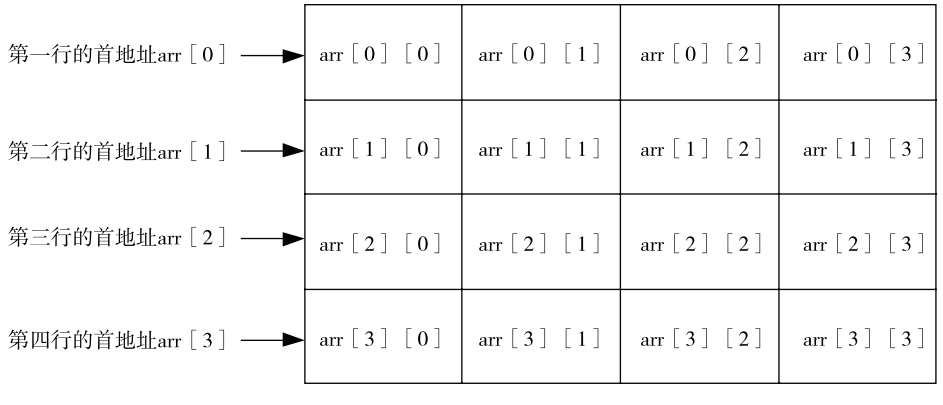


图 1-5　二维数组

在分析指针数组和数组指针如何访问二维数组中的各个元素之前，我们要明白二维数组每行的起始地址并不是只能用图 1-5 中的那种表示方式，还有很多方法可以表示每行的起始地址，如 *(arr+i) 和 arr+i 等。为了帮助读者更好地记忆，我们通过下面一段代码来学习其他表示二维数组每行起始地址的方式。

```
#include <stdio.h>

void main(int argc,char *argv[])
{
   int arr[4][4]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
   int i;

   for(i=0;i<4;i++)
   {
          printf(" 使用 arr+i 求得二维数组 arr 第 %d 行的起始地址为 :%d\n",i+1,arr+i);
          printf(" 使用 arr[i] 求得二维数组 arr 第 %d 行的起始地址为 :%d\n",i+1,arr[i]);
          printf(" 使用 *(arr+i) 求得二维数组 arr 第 %d 行的起始地址为 :%d\n",i+1,*(arr+i));
          printf(" 使用 &arr[i] 求得二维数组 arr 第 %d 行的起始地址为 :%d\n\n",i+1,&arr[i]);
   }

   return ;
}
```

运行结果：

```
使用 arr+i 求得二维数组 arr 第 1 行的起始地址为 :1244996
使用 arr[i] 求得二维数组 arr 第 1 行的起始地址为 :1244996
使用 *(arr+i) 求得二维数组 arr 第 1 行的起始地址为 :1244996
使用 &arr[i] 求得二维数组 arr 第 1 行的起始地址为 :1244996

使用 arr+i 求得二维数组 arr 第 2 行的起始地址为 :1245012
使用 arr[i] 求得二维数组 arr 第 2 行的起始地址为 :1245012
使用 *(arr+i) 求得二维数组 arr 第 2 行的起始地址为 :1245012
使用 &arr[i] 求得二维数组 arr 第 2 行的起始地址为 :1245012

使用 arr+i 求得二维数组 arr 第 3 行的起始地址为 :1245028
使用 arr[i] 求得二维数组 arr 第 3 行的起始地址为 :1245028
使用 *(arr+i) 求得二维数组 arr 第 3 行的起始地址为 :1245028
使用 &arr[i] 求得二维数组 arr 第 3 行的起始地址为 :1245028

使用 arr+i 求得二维数组 arr 第 4 行的起始地址为 :1245044
使用 arr[i] 求得二维数组 arr 第 4 行的起始地址为 :1245044
使用 *(arr+i) 求得二维数组 arr 第 4 行的起始地址为 :1245044
使用 &arr[i] 求得二维数组 arr 第 4 行的起始地址为 :1245044
```

在上面的代码中，我们使用了 4 种方式来获得每行的起始地址，因此行起始地址的表示方式并不唯一，读者在使用的时候可以自行选择。

下面接着讲解数组指针和指针数组是如何访问二维数组的，先看数组指针的访问方式。因为数组指针指向的是一个有 4 个整型元素的数组，所以可以把二维数组 arr 看成由 4 个元素 arr[0]，arr[1]，arr[2]，arr[3] 组成，每个元素都是含有 4 个整型元素的一维数组，所以当在代码中使用 p1=arr 的时候，p1 就指向了二维数组的第一行的首地址。在接下来的访问中，

由于 p1 指向的类型是 int [4]，所以从 p1 到 p1+1 的变化值为 44 个字节，即 p1+1=1245012。从前面的运行结果中可以发现，p1+1 刚好指向第二行的起始地址。至于为什么刚好能指向二维数组 arr 的第二行的首地址，这个问题将在第 4 章进行讲解。通过 p1+i 刚好能够取遍每行的起始地址，有了每行的起始地址之后，就可以通过“*(*(p1+i)+j)”来取出二维数组中每行的每一个元素。

指针数组的访问方式要更容易一些，因为定义的指针数组 p2 由 4 个元素 p2[0]，p2[1]，p2[2]，p2[3] 组成，每个元素都是一个整型指针，所以只需要在程序中取出每行的起始地址并放到 p2 指针数组对应的元素中，就可以访问二维数组 arr 中的元素了。

所以，在程序中使用指针数组和数组指针的时候，必须对它们有清晰的认识，要知道它们的本质是什么，以及如何使用。

1.7 指针函数和函数指针

指针函数其实是一个简称，是指带指针的函数，它本质上是一个函数，只是返回的是某种类型的指针。其定义的格式为：

```
类型标识符 * 函数名（参数表）
```

函数指针，从本质上说是一个指针，只是它指向的不是一般的变量，而是一个函数。因为每个函数都有一个入口地址，函数指针指向的就是函数的入口地址。其定义的格式为：

```
类型标识符（* 指针变量名）（形参列表）
```

接下来，通过分析下面的代码加深读者对指针函数和函数指针的理解。代码的功能为在输入字符串中查找指定的字符，如果查找成功，则打印出所查找字符后面的字符串，如果查找失败，则给出提示信息。

```
#include <stdio.h>

char* (*fun)(char *str,char *substr);

void input(char *str,char *substr)
{
   printf("请输入字符串:");
   gets(str);
   printf("请输入要搜索的字符串:");
   gets(substr);
}

int strlen(char *str)
{
   int i=0;
   while(str[i]!='\0')
           i++;
   return i;
}
```

```
char* serch_str(char *str,char *serch_str)
{
   int i,j,k;

   k = strlen(str) - strlen(serch_str);

   if ( k > 0 && NULL!=str && NULL!=serch_str)
   {
          for ( i = 0; i <= k; i++ )
                 for ( j = i; str[j] == serch_str[j-i]; j++ )
                        if ( serch_str[j-i+1] == '\0' )
                               return str+i+strlen(serch_str);
   }

   return NULL;
}

void print(char* ret_str)
{

   if ( ret_str !=NULL )
          printf(" 所搜索字符串之后的字符为 :%s\n",ret_str);
   else
          printf(" 没有找到所要搜索的字符串 \n");
}

void main()
{
   char str1[50],str2[50];
   char serch_str1[50],serch_str2[50];
   char* ret_str1,* ret_str2;

   input(str1,serch_str1);

   ret_str1 = serch_str(str1,serch_str1);

   printf(" 直接调用函数 serch_str()\n");
   print(ret_str1);

   input(str2,serch_str2);

   fun = serch_str;
   ret_str2 = fun(str2,serch_str2);

   printf(" 使用函数指针 fun 调用函数 serch_str()\n");
   print(ret_str2);

   return ;
}
```

运行结果：

```
请输入字符串 :Never forget to say thanks!
请输入要搜索的字符串 :say
```

```
直接调用函数 serch_str()
所搜索字符串之后的字符为 : thanks!

请输入字符串 :Keep on going never give up!
请输入要搜索的字符串 :going
使用函数指针 fun 调用函数 serch_str()
所搜索字符串之后的字符为 :never give up!
```

分析上面的代码，其中定义函数指针的形式为“char* (*fun)(char *str,char *substr);”，其所指向函数的返回类型为字符指针，所带参数是两个字符指针。在代码的实现中有些需要注意的地方，如在 strlen() 函数中通过一个结束符来判断字符串的长度，这是因为在输入字符串后面会自动添加一个结束符。由运行结果可知，采用了两种方式来实现函数的调用，一种是直接调用，即通过 serch_str() 函数来实现；另外一种是使用函数指针的方式来调用，即通过函数指针 fun 来实现，在调用之前，先使函数指针 fun 指向 serch_str 函数的入口地址，之后才能按照调用 serch_str() 函数的方式来使用。在使用函数指针的时候，需要注意函数指针要与它所指向的函数具有相同的类型，在用函数指针指向函数的时候是用“函数指针名 = 函数名”的方式来引用函数的。函数 serch_str() 是一个指针函数，返回的是一个字符指针。

1.8 传值和传址

传值，函数调用过程中参数传递的是实参的值，就是把实参传递给形参。对形参的修改不会影响到实参，这就相当于一个对实参备份的操作，即对形参的修改只是修改实参的备份，不会影响到实参。

传址，函数调用过程中参数传递的是地址，形参和实参共用一个空间，所以对于形参的修改会影响到实参。

下面通过一段代码来学习传值。

```
#include <stdio.h>

void swap(int p1,int p2)
{
   printf("\np1 和 p2 交换前 \n");
   printf("  p1=%d\tp2=%d\n",p1,p2);

   int temp;
   temp=p1;
   p1=p2;
   p2=temp;

   printf("\np1 和 p2 交换后 \n");
   printf("  p1=%d\tp2=%d\n",p1,p2);

   return ;
}

void main()
```

```
{
   int a,b;

   a=20;
   b=30;
   printf(" 调用 swap() 函数以前 \n");
   printf("  a=%d\tb=%d\n",a,b);

   swap(a,b);

   printf("\n 调用 swap() 函数以后 \n");
   printf("  a=%d\tb=%d\n",a,b);

   return ;
}
```

运行结果：

```
调用 swap() 函数以前
a=20  b=30

p1 和 p2 交换前
p1=20 p2=30

p1 和 p2 交换后
p1=30 p2=20

调用 swap() 函数以后
a=20  b=30
```

分析上面的运行结果发现，main() 函数中调用 swap() 函数前后 a 和 b 的值并没有改变，但是在 swap() 函数中交换前后 p1 和 p2 的值的确交换成功了，而在 main() 函数中为什么没有成功地实现交换呢？为了方便说明，我们用图 1-6 来展示参数是如何进行传值的。

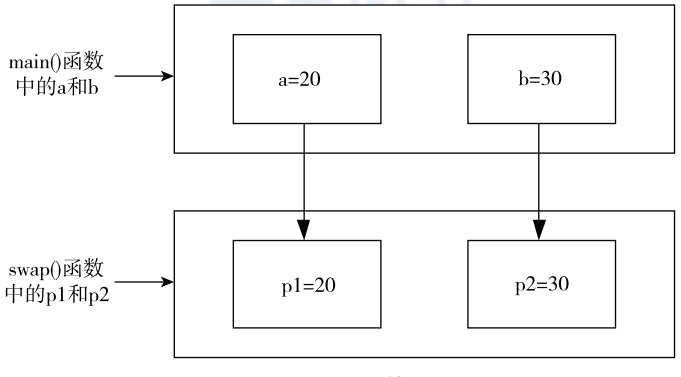


图 1-6　传值

从图 1-6 中清楚地发现，在函数的调用过程中实现的是参数 a 和 b 的传值，即把 a 和 b 的值传递给 p1 和 p2，swap() 函数中的 p1 和 p2 拥有自己的存储空间，所以接下来在 swap() 函数中进行的交换操作仅仅是对 p1 和 p2 进行的，不会影响到 main() 函数中 a 和 b 的值。这

也就是为什么在传值时修改形参不会影响实参。接下来再通过下面一段代码来看看传址。

```
#include <stdio.h>

void swap(int *p1,int *p2)
{
   printf("\n*p1 和 *p2 交换前 \n");
   printf("  *p1=%d\t*p2=%d\n",*p1,*p2);

   int temp;
   temp=*p1;
   *p1=*p2;
   *p2=temp;

   printf("\n*p1 和 *p2 交换后 \n");
   printf("  *p1=%d\t*p2=%d\n",*p1,*p2);

   return ;
}

void main()
{
   int a,b;

   a=20;
   b=30;
   printf(" 调用 swap() 函数以前 \n");
   printf("  a=%d\tb=%d\n",a,b);

   swap(&a,&b);

   printf("\n 调用 swap() 函数以后 \n");
   printf("  a=%d\tb=%d\n",a,b);

   return ;
}
```

运行结果：

```
调用 swap() 函数以前
a=20  b=30

*p1 和 *p2 交换前
*p1=20 *p2=30

*p1 和 *p2 交换后
*p1=30 *p2=20

调用 swap() 函数以后
a=20  b=30
```

分析上面的运行结果发现，此时不仅在 swap() 函数中成功交换了 *p1 和 *p2，而且在 main() 函数中也成功实现了 a 和 b 的交换。为了能够更加直观地说明交换的实现，在此使用图 1-7 来展示参数是如何进行传递的。

在图 1-7 中可以清楚地发现，在函数的调用过程中实现的是参数 a 和 b 的传址，即把 a 和 b 存储单元的地址传递给 p1 和 p2，swap() 函数中的形参不再拥有自己的存储空间，它们分别指向 a 和 b 的存储单元，所以接下来在 swap() 函数中对 p1 和 p2 指向的存储单元进行交换的操作其实是对 a 和 b 进行的。这也是在采用传址的时候修改形参也会影响实参的原因。

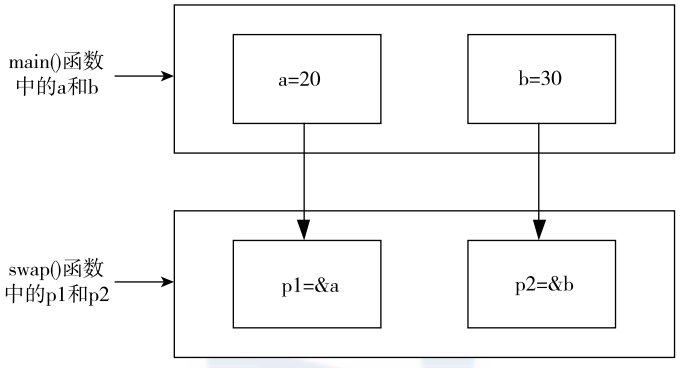


图 1-7　传址

1.9　递归和嵌套

学习过函数的读者，应该对递归和嵌套并不陌生，但是在使用递归和嵌套的时候，我们要知道它们各自的含义、使用方法及注意事项。

函数的嵌套调用就是在一个函数中去调用另外一个函数，但是要注意，可以嵌套调用函数，但不能嵌套定义函数，因为 C 语言的各个函数之间是互相平行的关系，不存在上下级关系，所以不能在一个函数中定义另外一个函数。

函数的递归调用就是函数在调用的过程中自身既是主调函数，又是被调函数。需要注意的是，如果在使用递归调用的过程中没有停止条件，那么递归将会无限制地进行下去，直到程序崩溃为止。所以在使用递归调用的时候要尤其注意给定一个递归调用的停止条件。

下面通过代码来了解函数的嵌套调用和函数的递归调用，先来了解嵌套调用。

```
#include <stdio.h>

void print(int *arr,int n)
{
    int i;

    printf("排序后的数组为 \n");
    for(i=0;i<n;i++)
    {
            printf("arr[%d]=%d\t",i,*(arr+i));
            if((i+1)%4==0)
                    printf("\n");
    }
```

```
    return ;
}
void sort(int *arr, int n)
{
    int i,j,k,temp;

    for(i=0;i<n-1;i++)
    {
            k=i;
            for(j=i+1;j<n;j++)
            {
                    if (arr[j]<arr[k])
                            k=j ;
            }
            if(k!=i)
            {
                    temp=arr[i];
                    arr[i]=arr[k];
                    arr[k]=temp;
            }
    }
    print(arr,n);

    return ;
}

void main()
{
    int arr[8];
    int i;

    for(i=0;i<8;i++)
    {
            printf(" 请输入 arr[%d]:",i);
            scanf("%d",&arr[i]);
    }
    sort(arr,8);

    return ;
}
```

运行结果：

```
请输入 arr[0]:22
请输入 arr[1]:54
请输入 arr[2]:12
请输入 arr[3]:76
请输入 arr[4]:89
请输入 arr[5]:55
请输入 arr[6]:34
请输入 arr[7]:99
排序后的数组为
arr[0]=12       arr[1]=22       arr[2]=34       arr[3]=54
arr[4]=55       arr[5]=76       arr[6]=89       arr[7]=99
```

下面通过图 1-8 说明如何实现函数的嵌套调用。

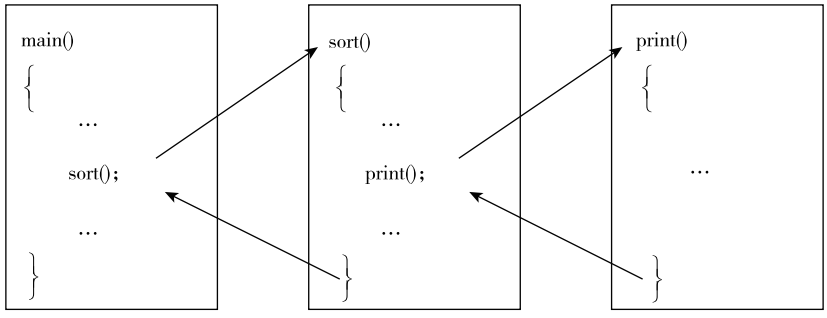


图 1-8　函数的嵌套调用

分析图 1-8 中的嵌套过程，首先执行 main() 函数，在 main() 函数中嵌套了 sort() 函数，当执行到调用 sort() 函数处的时候，中止当前 main() 函数的执行，转到 sort() 函数中去执行。在 sort() 函数中嵌套调用了 print() 函数，当执行到调用 print() 函数处的时候，中止当前的 sort() 函数的执行，转到 print() 函数中去执行。当执行完 print() 函数的时候，再次返回到 sort 函数中的调用 print() 函数处继续往下执行。当执行完 sort() 函数的时候，又返回 main() 函数中调用 sort() 函数处继续往下执行。这就是函数嵌套调用的流程。但是值得注意的是，不能在一个函数中定义另外一个函数。

下面来看一个递归的例子，猴子第一天摘下若干个桃子，当即吃了一半，觉得不过瘾，又多吃了一个。第二天早上将剩下的桃子吃掉一半，又多吃一个。以后每天早上都吃了前一天剩下的一半多一个。到第十天早上想再吃时，只剩下一个桃子了。求第一天共摘了多少桃子。

这个猴子吃桃问题是个典型的递归问题，想要知道猴子第一天总共摘了多少个桃子，就要想办法知道猴子在第二天拥有的桃子数目，而第二天所拥有的桃子数又取决于第三天所拥有的，依此类推，直到第十天。因为知道第十天的桃子数，所以可以推出第九天的桃子数，得出了第九天的桃子数之后又可以推出第八天的桃子数……最终可以推出第一天的桃子数。设猴子第 n 天所拥有的桃子数为 peach_total(n)，那么就可以用下面的公式来表示猴子每天所拥有的桃子数目了。

$$\text{peach\_totaln}(n)=\begin{cases}1 & n=10\\ (\text{peach\_total}(n+1)+1)\times 2 & 1\leqslant n<10\end{cases}$$

有了上面这个公式，写代码就容易多了，下面来看代码的实现。

```
#include <stdio.h>

int peach_total(int n)
{
    int total_n;
    if(10==n)
          total_n=1;
    else
          if(n<10)
```

```
                    total_n=(peach_total(n+1)+1)*2;

    return total_n;
}

void main()
{
    int total;

    total=peach_total(1);

    printf(" 猴子一共摘了 %d 个桃子。\n",total);

    return ;
}
```

运行结果：

```
猴子一共摘了 1534 个桃子。
```

下面用图 1-9 来说明递归函数的调用过程。

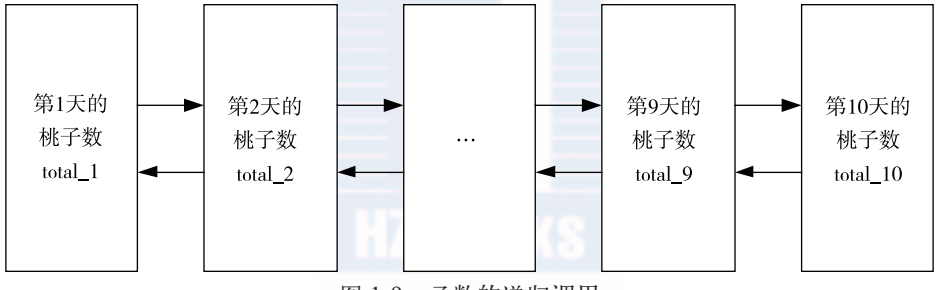


图 1-9　函数的递归调用

从图 1-9 中可以看出，首先在 main() 函数中调用 peach_total(1)，表示求第一天的桃子数目，在 peach_total() 函数中进行递归调用 peach_total(2)，peach_total(2) 又调用 peach_total(3)，依此类推，直到 peach_total(10)，由于 peach_total(10) 已知，因此反过来可以依次推出 peach_total(9)，peach_total(8)，…，peach_total(1)，这样就得到了猴子第一天所摘取的桃子数目，猴子摘桃问题得以解决。

综上所述，大致可以归纳出递归调用有如下特点。

- ❑ 函数直接或者间接调用其本身。
- ❑ 要有递归调用的停止条件，即递归调用的停止条件被满足后，停止调用自身函数。如果没有停止条件，那么递归将永远执行下去，直至将系统资源耗尽。
- ❑ 当不满足递归调用的停止条件时，继续调用涉及递归调用的表达式。在调用函数自身时，有关停止条件的参数会向递归终止的方向变化。

1.10　结构体

在解决实际问题的过程中常常会遇到这样的问题，如存储一个公司员工的基本信息，包括姓名、性别、年龄、月薪等，其中的信息需要使用字符数组、整型、指针类型等，有的读者一开始会想到用数组类存储，但是细想就知道不能使用数组，因为数组只能用来存储相同类型的数据，而这里的数据类型显然不止一种。我们不希望在存储员工信息的时候使用单个变量来分别表示每类信息，因为这样不能够很好地反映出它们之间的内在联系。为了能够将这些不同类型的元素放到一起，可以利用 C 语言中的结构体将这些元素类型“封装”在一起，得到一种新的自定义数据类型。

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。定义结构体的一般形式为：

```
struct 结构体名{
成员类型     成员名;
……
成员类型     成员名;
};
```

结构体名是自定义的标识符，但是要遵循自定义标识符的命名规则。其中的成员类型可以是任何基本数据类型，也可以是指针或数组等复合数据类型，还可以是结构体或共用体。

接下来用结构体按照如下的方式来描述公司员工的基本信息。

```
struct  personnel{
          char          name[20];
          char          sex[10];
          int           age;
          float         salary;
   };
```

定义了这种结构体之后，该如何来定义结构体变量呢？看看下面几种定义结构体变量的实现方法。

```
struct 结构体名{
    成员类型        成员名;
    ……
    成员类型        成员名;
} 变量名 1，变量名 2……;
```

也可以去掉结构体名，直接定义结构体变量。

```
struct  {
    成员类型        成员名;
    ……
    成员类型        成员名;
} 变量名 1，变量名 2……;
```

还可以先定义结构体，再定义结构体变量。

```
struct 结构体名 {
    成员类型        成员名;
    ……
    成员类型        成员名;
};
struct     结构体名 变量名 1，变量名 2……;
```

对于结构体成员的引用，读者可以在编程中根据自己的习惯选择相应的引用方式。值得注意的一点是，结构体为它的每一个成员都分配存储空间，这与接下来所要讲的共用体是不同的。通过前面的介绍，我们对结构体有了一个初步的了解，接下来看一段代码，以加深对结构体的理解。

```
#include <stdio.h>

struct  personnel{
          char    name[20];
          char    sex[10];
          int     age;
          double  salary;
};

void input(struct personnel pers[],int n)
{
   int i;
   for(i=0;i<n;i++)
   {
          printf(" 请依次输入员工的姓名，性别，年龄，月薪 :");
          scanf("%s%s%d%lf",&pers[i].name,&pers[i].sex,&pers[i].age,&pers[i].salary);
   }

   return ;
}

struct personnel find_max(struct personnel pers[],int n)
{
   int i,index;
   double tmp;
   tmp = pers[0].salary;
   for(i=1;i<n;i++)
          if(pers[i].salary>tmp)
          {
                 index=i;
                 tmp=pers[i].salary;
          }

   return  pers[index];
}

struct personnel find_min(struct personnel pers[],int n)
{
   int i,index;
   double tmp;
   index=0;
```

```
    tmp = pers[0].salary;
    for(i=1;i<n;i++)
            if(pers[i].salary<tmp)
            {
                    index=i;
                    tmp=pers[i].salary;
            }
    return  pers[index];
}

void print(struct personnel pers)
{
    printf(" 员工姓名 :%s\t 性别 :%s\t 年龄 :%d\t 月薪 :%6.2f\n",
            pers.name,pers.sex,pers.age,pers.salary);

    return ;
}

void main()
{
    struct  personnel pers[4],pers_max,pers_min;

    input(pers,4);

    pers_max=find_max(pers,4);

    printf("\n 工资最高的员工信息 \n");

    print(pers_max);

    pers_min=find_min(pers,4);

    printf("\n 工资最低的员工信息 \n");

    print(pers_min);

    return ;
}
```

运行结果：

```
请依次输入员工的姓名，性别，年龄，月薪：王小明 男 20 5600
请依次输入员工的姓名，性别，年龄，月薪：王美美 女 22 8666
请依次输入员工的姓名，性别，年龄，月薪：张小明 男 56 12300
请依次输入员工的姓名，性别，年龄，月薪：牟小玲 女 21 5800

工资最高的员工信息
员工姓名：张小明 性别：男 年龄 :56 月薪 :12300.00

工资最低的员工信息
员工姓名：王小明 性别：男 年龄 :20 月薪 :5600.00
```

分析上面的代码，定义的结构体中包含了员工的姓名、性别、年龄和月薪，在 main() 函数中定义了一个含有 4 个元素的结构体数组。定义的函数有输入函数 input()，查找月薪最

高的员工函数 find_max()，查找员工月薪最低的员工函数 find_min()，以及用来打印查找到的员工信息的函数 print()。细心的读者会发现，在上面的代码中，我们将结构体作为函数的返回类型，成功地返回了所需要的信息，因此可以看出，函数的返回类型不仅可以是简单的 char 和 int 等类型，还可以是自定义的结构体等复合类型。

1.11 共用体

共用体是 C 语言的另外一种构造类型，与前面介绍的结构体类似。共用体也由基本数据结构组合而成，但是共用体和结构体却有本质区别，因为结构体中的每个成员都占用存储单元，所以结构体所占用的内存大小为所有成员各自占用的内存大小之和，而共用体占用的内存大小由其成员中占用内存最大的那个决定，所有的成员都占用同一个起始地址和同一段内存空间。对于共用体变量，在某一时刻，只能存储其某一成员的信息。

共用体类型的定义形式为：

```
union       共用体名{
    成员类型 成员名;
    ……
    成员类型 成员名;
};
```

共用体名是定义的共用体类型的标识符，同样要遵循自定义标识符的命名规则。而其中的成员类型可以是任何基本数据类型，也可以是指针、数组等复合数据类型，还可以是结构体或者共用体。

下面来看几种共用体变量的定义方法。

```
union       共用体名{
    成员类型 成员名;
    ……
    成员类型 成员名;
}共用体变量1，共用体变量2……;
```

也可以省略掉共用体名。

```
union       {
    成员类型 成员名;
    ……
    成员类型 成员名;
}共用体变量1，共用体变量2……;
```

还可以先定义共用体类型，再定义共用体变量。

```
union       共用体名{
    成员类型 成员名;
    ……
    成员类型 成员名;
};
union       共用体名 共用体变量1，共用体变量2……;
```

我们发现共用体和结构体不管是在定义方式上还是在变量定义上都非常相似，但是它们之间有本质的区别，为了使读者更好地区别它们，我们通过下面的一段代码来看结构体和共用体之间究竟有什么样的区别。

```
#include <stdio.h>

struct  str{
          int     a;
          int     b;
          int     c;
};

union  uni{
          char    a;
          int     b;
          int     c;
};

void main()
{
   struct  str     x;
   union   uni     y;

   printf(" 结构体所占的内存大小为 %d 字节 \n",sizeof(x));
   printf(" 结构体中成员变量 a 的地址为 %d\n",&x.a);
   printf(" 结构体中成员变量 b 的地址为 %d\n",&x.b);
   printf(" 结构体中成员变量 c 的地址为 %d\n\n",&x.c);

   printf(" 共用体所占的内存大小为 %d 字节 \n",sizeof(y));
   printf(" 共用体中成员变量 a 的地址为 %d\n",&y.a);
   printf(" 共用体中成员变量 b 的地址为 %d\n",&y.b);
   printf(" 共用体中成员变量 c 的地址为 %d\n",&y.c);

   return ;
}
```

运行结果：

```
结构体所占的内存大小为 12 字节
结构体中成员变量 a 的地址为 1245048
结构体中成员变量 b 的地址为 1245052
结构体中成员变量 c 的地址为 1245056

共用体所占的内存大小为 4 字节
共用体中成员变量 a 的地址为 1245044
共用体中成员变量 b 的地址为 1245044
共用体中成员变量 c 的地址为 1245044
```

分析上面的代码，sizeof 操作符的作用就是计算结构体变量 x 和共用体变量 y 所占用的内存空间。通过运行结果我们发现，x 所占用的内存空间大小为 12 字节，刚好等于 sizeof(a)+ sizeof(b)+ sizeof(c)。正如上面所介绍的，结构体的每个成员都有自己的存储空间，每个成员的起始地址都不相同，它所占用的内存大小等于各个成员所占用的内存大小之和。

y 所占用的内存大小为 4 字节，而 sizeof(a)+ sizeof(b)+ sizeof(c)=9 字节，即共用体所占用的内存大小并不等于它的每个成员所占用的内存大小之和，正如前面所讲的，共同体所占用的内存大小就等于其占用内存最大的成员所占用的内存大小。y 中占用内存最大的为 int 型变量 b 和 int 型变量 c，占用 4 字节，所以共用体占用的内存大小为 4 字节，并且共用体中每个成员的起始地址都相同，它们共用一个存储空间。我们可以用图 1-10 和图 1-11 来说明结构体和共用体在内存中的结构。

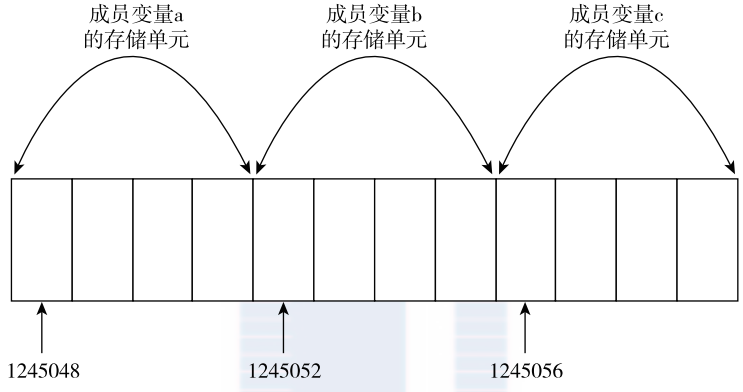


图 1-10　结构体 x 的内存结构

可以通过下面的代码来验证图 1-10 和图 1-11 中 x 和 y 的内存结构。

```
#include <stdio.h>

struct  str{
          int            a;
          int            b;
          int            c;
};

union  uni{
          char           a;
          int            b;
          int            c;
};

void main()
{
   struct  str     x;
   union   uni     y;
   x.a = 0x2a3d;
   x.b = 0xc4df;
   x.c = 0x5bac;
   printf(" 结构体中成员变量 a 的值为 %x\n",x.a);
   printf(" 结构体中成员变量 b 的值为 %x\n",x.b);
   printf(" 结构体中成员变量 c 的值为 %x\n\n",x.c);
```

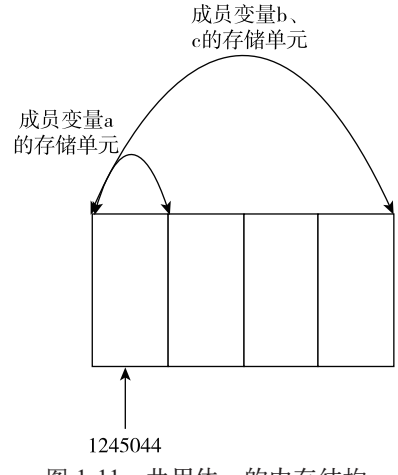


图 1-11　共用体 y 的内存结构

```
    y.a=0x1345;
    y.b=0x1345;
    y.c=0xb548;
    printf("共用体中成员变量 a 的值为 %x\n",y.a);
    printf("共用体中成员变量 b 的值为 %x\n",y.b);
    printf("共用体中成员变量 c 的值为 %x\n",y.c);

    return ;
}
```

运行结果：

```
结构体中成员变量 a 的值为 2a3d
结构体中成员变量 b 的值为 c4df
结构体中成员变量 c 的值为 5bac

共用体中成员变量 a 的值为 48
共用体中成员变量 b 的值为 b548
共用体中成员变量 c 的值为 b548
```

上述代码先对结构体变量 x 的每个成员赋初值，然后输出，结果和初始值完全一致，但是当对共用体赋初值并输出的时候，其结果都是最后一次对共用体变量 y 中成员 c 的赋值。由此也可以看出，共用体是共享存储空间的，对其成员变量赋值会覆盖之前对共用体中变量所赋的值。当打印 a 的值时，因为它在内存中占用的是最低字节的内存，所以打印出来的是最后对共用体变量 y 的成员 c 赋值的低字节部分 48。

接下来我们用结构体和共用体嵌套定义一个自定义类型来登记学校老师和学生的信息。

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

struct  infor{
          char    name[20];
          char    sex[10];
          int     age;
          char    identity;
          union   otherinf{
                  struct {
                          char    profession[10];
                          char    department[20];
                          double  salary;
                  }teacher;
                  struct  {
                          char    num[20];
                          char    department[20];
                          char    major[20];
                  }student;
          }perinf;
};

void print(struct infor per[],int n)
{
```

```
    int i;
    for(i=0;i<n;i++)
    {
            printf("姓名 :%s\t 性别 :%s\t 年龄 :%d\t",per[i].name,per[i].sex,per[i].age);
            if('s'==per[i].identity)
            {
                    printf("学生的学号 :%s\t 所属的院系 :%s\t 专业 :%s\n",per[i].perinf.student.num,
                            per[i].perinf.student.department,per[i].perinf.student.major);
            }
            else
            {
                    printf("教师的职称 :%s\t 所属的院系 :%s\t 月薪 :%6.2f",per[i].perinf.teacher.profession,
                            per[i].perinf.teacher.department,per[i].perinf.teacher.salary);
            }
    }

    return ;
}

void input(struct infor per[],int n)
{
    int i;

    for(i=0;i<n;i++)
    {
            printf("请依次输入姓名，性别，年龄，身份：");
            scanf("%s%s%d%s",&per[i].name,&per[i].sex,&per[i].age,&per[i].identity);
            if('s'==per[i].identity)
            {
                    printf("请依次输入学生的学号，所属的院系，专业 :");
                    scanf("%s%s%s",&per[i].perinf.student.num,&per[i].perinf.student.
department,&per[i].perinf.student.major);
            }
            else if('t'==per[i].identity)
            {
                    printf("请依次输入教师的职称，所属的院系，月薪 :");
                    scanf("%s%s%lf",&per[i].perinf.teacher.profession,&per[i].
perinf.teacher.department,&per[i].perinf.teacher.salary);
            }
            else
            {
                    printf("输入出错 !\n");
                    exit(0);
            }
    }

    return ;
}

void main()
{
    struct infor per[2];

    input(per,2);
```

```
    print(per,2);

    return ;
}
```

运行结果：

```
请依次输入姓名，性别，年龄，身份：张丽玲 女 35 t
请依次输入教师的职称，所属的院系，月薪：教授 电信学院 8900
请依次输入姓名，性别，年龄，身份：张晓明 男 22 s
请依次输入学生的学号，所属的院系，专业：202060639 电信学院 信息与系统

姓名：张丽玲      性别：女 年龄：35 教师的职称：教授 所属的院系：电信学院      月薪：890
0.00
姓名：张晓明      性别：男 年龄：22 学生的学号：202060639     所属的院系：电信学院
专业：信息与系统
```

分析上面的代码，其中定义了一个登记学生和教师信息的结构体，结构体中包含姓名（name）、性别（sex）、年龄（age）和身份（identity），其中身份的取值为 's' 和 't'，分别代表学生和教师。在结构体中嵌套了共用体，共用体中又嵌套了两个结构体，如果身份为学生，那么选择共用体中的学生信息结构体类型成员，包括学号（num）、院系（department）、专业（major）相关信息；如果身份为教师，那么选择共用体中的教师信息结构体类型成员，包括职称（profession）、院系（department）、月薪（salary）相关信息。根据身份的不同，在共用体中选择不同结构体类型的成员。在使用结构体和共用体进行嵌套的时候要尤其注意其中成员的引用方法，从最外层类型变量开始引用它的成员，如果它的成员是共用体或者结构体类型的变量，那么接着以共用体或者结构体类型变量的方式引用它的成员变量。

1.12　枚举

枚举，从字面来理解，就是一一列举。而在 C 语言中有一种枚举类型，其含义就是将具有相同属性的一类数据一一列举出来。

定义枚举类型的一般形式为：

```
enum        枚举类型名{
    标识符 1 [= 整型常数],
    ……
    标识符 n [= 整型常数],
};
```

其中，[] 中的部分可有可无。枚举类型是有序类型，如果没有为枚举常量指定值，那么它的值比前一个值大 1，枚举常量的值默认从 0 开始。枚举元素按照定义时的先后顺序分别编号为 0，1，2，…，n-1。当然，也可以人为指定枚举类型常量的值。枚举类型名的命名同样要遵循标识符的命名规则，而其中的标识符 1，2,…，n 是定义的枚举类型的全部取值。定义枚举类型的几种方法与上面的结构体和共用体的定义方法类似，有以下三种。

```
enum       枚举类型名{
   标识符1 [=整型常数],
   ……
   标识符n [=整型常数],
}枚举变量1，枚举变量2……;
```

也可以省略枚举类型名，如：

```
enum       {
   标识符1 [=整型常数],
   ……
   标识符n [=整型常数],
}枚举变量1，枚举变量2……;
```

还可以采用先定义枚举类型，后定义枚举变量的方法，如：

```
enum       枚举类型名{
   标识符1 [=整型常数],
   ……
   标识符n [=整型常数],
};
enum       枚举类型名        枚举变量1，枚举变量2……;
```

介绍完枚举类型的几种定义方法，下面通过代码进一步了解枚举类型。

```
#include <stdio.h>

enum  nu1{
           a,
           b,
           c,
           d,
};

enum  nu2{
           e=3,
           f=2,
           g=1,
           h,
};

void main()
{
   printf("枚举类型常量a的值为:%d  b的值为:%d  c的值为:%d  d的值为:%d\n",a,b,c,d);
   printf("枚举类型变量e的值为:%d  f的值为:%d  g的值为:%d  h的值为:%d\n",d,f,g,h);

   return ;
}
```

运行结果：

```
枚举类型常量a的值为:0  b的值为:1  c的值为:2  d的值为:3
枚举类型变量e的值为:3  f的值为:2  g的值为:1  h的值为:2
```

分析上面的运行结果，在定义的枚举类型 nu1 中，我们没有指定枚举常量的值，而是采

用默认的方法，打印出来的结果与前面分析的一致，从 0 开始，后面的枚举常量的值比前面的枚举常量的值大 1；在枚举类型 nu2 中，我们指定了枚举常量的值，所以打印出来的结果就是指定的值，但是没有指定最后一个枚举常量的值，所以它比前面的枚举常量的值大 1。

使用枚举类型时需要注意，在同一个作用域内不能出现重名的枚举常量名，如：

```
#include <stdio.h>

void main()
{
   enum  nu1{
          a,
   };

   enum  nu2{
          a,
   };

   return ;
}
```

编译上面这段代码时会出现错误，提示信息为“error C2371: 'a' : redefinition; different basic types”。如果修改一下上面这段代码，把枚举类型 nu2 的作用域用一个 {} 限制起来就不会出错了，如：

```
#include <stdio.h>

void main()
{
   enum  nu1{
          a,
   };
   {
          enum  nu2{
                 a,
          };
   }

   return ;
}
```

这样就不会出错了，因为将枚举类型 nu2 的作用域限制在 {} 范围内，而枚举类型 nu1 的作用域是整个 main() 函数体。而对结构体和共用体，则没有这样的要求。

1.13　位域

在存储信息的时候，我们可能并不需要占用一个完整的字节，而只需占一个或几个二进制位，如要存储一个八进制数据，只需要 3 个二进制位就够了。为了节省存储空间，C 语言提供了位域这种数据结构。所谓位域，就是把存储空间中的二进制位划分为几个不同的区域，并说明每个区域的位数，每个域有一个域名，允许在程序中按域名进行操作。定义位域

的一般形式为：

```
struct     位域结构名 {
    类型说明符        位域名：位域长度；
    ……
    类型说明符        位域名：位域长度；
};
```

位域结构名同样要遵循标识符的命名规则。位域变量的定义与之前讲解的结构体等非常类似，有以下三种方法。

```
struct     位域结构名 {
    类型说明符        位域名：位域长度；
    ……
    类型说明符        位域名：位域长度；
} 位域变量名 1，位域变量名 2…… ；
```

其中，位域结构名可以省略掉，直接定义位域变量，如：

```
struct     {
    类型说明符        位域名：位域长度；
    ……
    类型说明符        位域名：位域长度；
} 位域变量名 1，位域变量名 2…… ；
```

还可以先定义位域类型，再定义位域变量名，如：

```
struct     位域结构名 {
    类型说明符        位域名：位域长度；
    ……
    类型说明符        位域名：位域长度；
};
struct     位域结构名        位域变量名 1，位域变量名 2…… ；
```

读者可以根据自己的实际情况来决定使用哪种方式定义位域变量。下面通过代码对位域加以分析。

```
#include <stdio.h>

struct     _data
{
   char   a:6;
   char   b:2;
   char   c:7;
}data;

void main()
{
   printf(" 位域变量 data 起始地址为 ： %d\n",&data);
   printf(" 位域变量 data 占用内存大小为 :%d 字节 \n",sizeof(data));

   return ;
}
```

运行结果：

```
位域变量 data 起始地址为 : 4233496
位域变量 data 占用内存大小为 :2 字节
```

我们通过图 1-12 说明位域变量 data 的内存结构。

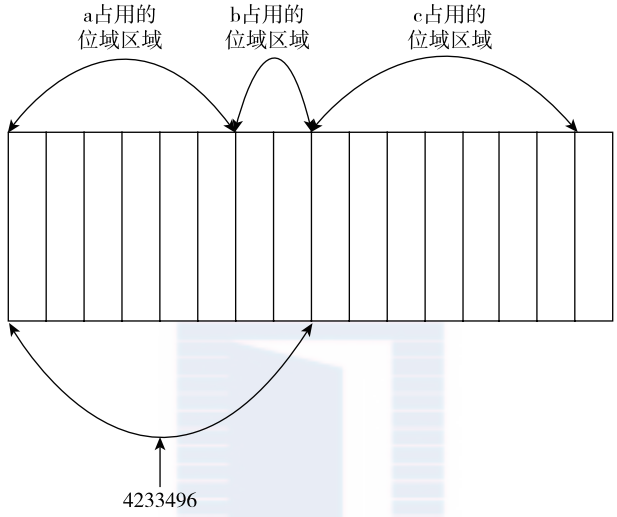


图 1-12　位域变量 data 的内存结构（1）

在图 1-12 的内存结构中，位域变量 data 只占用 2 个字节。当相邻位域的类型相同时，如果其位宽之和小于该类型所占用的位宽大小，那么后面的位域紧邻前面的位域存储，直到不能容纳为止；如果位宽之和大于类型所占用的位宽大小，那么就从下一个存储单元开始存放。我们适当修改上面的代码来看看位宽之和大于类型所占用的位宽大小的情形。

```
#include <stdio.h>

struct     _data
{
   char   a:6;
   char   b:4;
   char   c:7;
}data;

void main()
{
   printf(" 位域变量 data 起始地址为 : %d\n",&data);
   printf(" 位域变量 data 占用内存大小为 :%d 字节 \n",sizeof(data));

   return ;
}
```

运行结果：

```
位域变量 data 起始地址为 : 4233496
```

```
位域变量 data 占用内存大小为 :3 字节
```

再通过图 1-13 来说明此时 data 的内存结构。

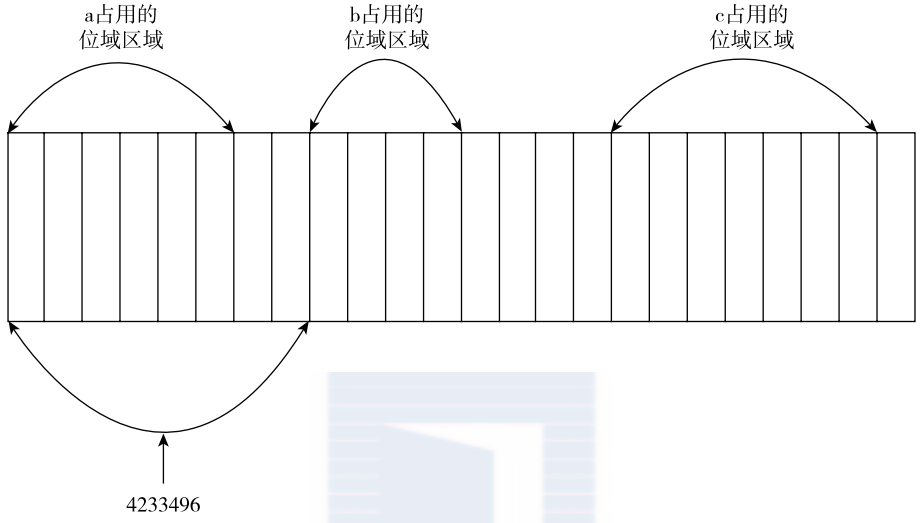


图 1-13　位域变量 data 的内存结构（2）

如果相邻位域的类型不同，不同编译器的处理方式可能有所不同，在此以 VC++ 6.0 为准进行讲解。VC++ 6.0 在进行编译的时候，不同类型的位域存放在不同的位域类型字节中，如：

```
#include <stdio.h>

struct     _data
{
   char           a:6;
   int            b:22;
   char           c:7;
}data;

void main()
{
   printf(" 位域变量 data 起始地址为 : %d\n",&data);
   printf(" 位域变量 data 占用内存大小为 :%d 字节 \n",sizeof(data));

   return ;
}
```

运行结果：

```
位域变量 data 起始地址为 : 4233496
位域变量 data 占用内存大小为 :12 字节
```

我们通过图 1-14 来说明 data 的内存结构。

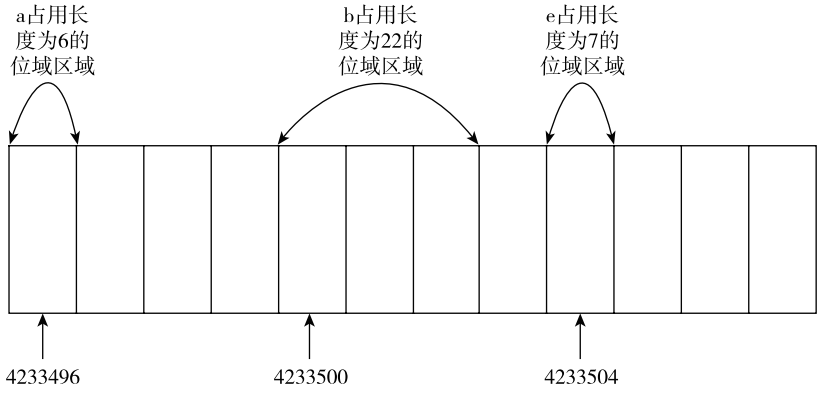


图 1-14　位域变量 data 的内存结构（3）

在图 1-14 中我们发现，默认情况下，位域结构中的字节对齐方式由其中占用字节数最大的类型所决定。在前面定义的位域中，占用内存最大的是 int 型，占用 4 字节，所以使用 4 字节对齐。首先从起始地址 4233496 处开始使用 6 个位域的长度来存储位域 a，由于位域 a 和位域 b 为不同类型，所以不能存储在同一个字节当中，寻找下一个起始地址来存储位域 b，存储位域 b 时要求地址的偏移量（这里的偏移量为成员起始地址相对于位域变量的起始地址，也就是相对于第一个成员的起始地址）必须是所使用的字节对齐方式和自身类型所占用字节数这两者中最小值的整数倍，这里为 4 字节对齐，而 int 变量所占用的内存大小也为 4 字节，即偏移量必须为 4 的整数倍，由此可知域 b 的起始地址为 4233500。由于接下来的位域 c 是 char 型，与位域 b 不同，所以不能在 int 型变量所占用的存储空间中存放位域 c，存储位域 c 从起始地址 4233504 开始，因为是 4 字节对齐，要求最终位域结构所占用的存储空间必须是 4 的整数倍，所以位域最终占用了 12 字节大小的存储空间。

适当修改上面的代码，再来看看运行结果。

```
#include <stdio.h>

struct     _data
{
   int             b:22;
   char            a:6;
   char            c:7;
}data;

void main()
{
   printf(" 位域变量 data 起始地址为 : %d\n",&data);
   printf(" 位域变量 data 占用内存大小为 :%d 字节 \n",sizeof(data));

   return ;
}
```

运行结果：

```
位域变量 data 起始地址为 : 4233496
位域变量 data 占用内存大小为 :8 字节
```

我们发现此时位域结构所占用的内存空间变小了，变为了 8 字节。我们仅仅交换了位域 a 和位域 b 的位置，就导致所占用的内存空间发生了变化，这是为什么呢？首先从起始地址 4233496 处开始使用 22 个位域的长度来存储位域 b，因为接下来的位域是 char 型，所以必须存储在 int 型所占内存单元之外，因为位域 a 是 char 型，占用 1 字节，而采用的是 4 字节对齐，所以只需要偏移量是 1 的整数倍，也就是可以在接下来的地址 4233500 所指向的存储单元存储位域 a。接下来的位域 c 也是 char 型，由于位域 a 和位域 c 两者的位宽之和为 13，大于 char 型所占用的位宽 8，所以要使用接下来的地址 4233501 所指向的存储单元存储位域 c，由于是 4 字节对齐，因此最终所占用的内存大小必须是 4 的整数倍，此时位域结构占用了 8 字节。

上面都是使用默认的字节对齐方式，接下来通过“#pragma pack (2)”来指定采用 2 字节对齐。

```
#include <stdio.h>

#pragma pack (2)

struct     _data
{
   int             a:16;
   char            b:4;
   char            c:6;
}data;

void main()
{
   printf(" 位域变量 data 起始地址为 : %d\n",&data);
   printf(" 位域变量 data 占用内存大小为 :%d 字节 \n",sizeof(data));

   return ;
}
```

运行结果：

```
位域变量 data 起始地址为 : 4233496
位域变量 data 占用内存大小为 :6 字节
```

此时，data 的内存结构如图 1-15 所示。

我们在代码中使用了一句“#pragma pack (2)”来指定采用 2 字节对齐方式，与前面的代码最大的区别是，此时位域结构所占用的内存空间必须是 2 的整数倍，而不是 4 的整数倍，所以此时所占用的内存大小为 6。

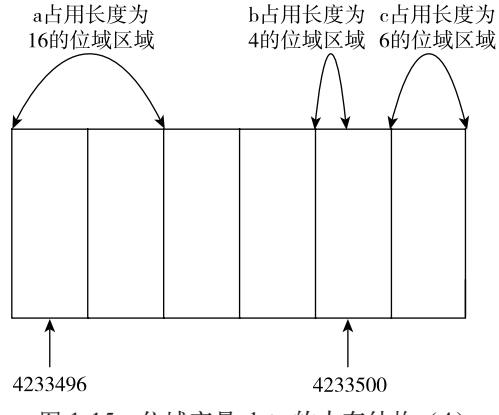


图 1-15　位域变量 data 的内存结构（4）

看完上面的讲解，细心的读者会发现一个问题，对于那些没有使用的位域段，编译器是怎么处理的呢？我们通过一段代码来分析编译器对没有使用的位域段的处理方法。

```
#include <stdio.h>

#pragma pack (2)

struct     _data
{
   int                    a:16;
   unsigned  char         b:5;
   char                   c:5;
}data;

void main()
{
   int *p=(int *)&data;

   printf(" 位域结构的起始地址为 :%d\n\n",p);
   data.a=2;

   printf(" 整型指针 p 所指向的单元存储的值为 :%d\n",*p);
   printf(" 位域 a 的值为 :%d\n",data.a);

   char *p1=(char*)(p+1);
   data.b=18;
   printf("\n 字符指针 p1 所指向的单元存储的值为 :%d\n",*p1);
   printf(" 位域 b 的值为 :%d\n",data.b);

   data.c=255;
   char *p2;
   p2 = p1+1;
   printf("\n 字符指针 p2 所指向的单元存储的值为 :%d\n",*p2);
   printf(" 位域 c 的值为 :%d\n",data.c);

   return ;
}
```

运行结果：

```
位域结构的起始地址为 :4233624

整型指针 p 所指向的单元存储的值为 :2
位域 a 的值为 :2

字符指针 p1 所指向的单元存储的值为 :18
位域 b 的值为 :18

字符指针 p2 所指向的单元存储的值为 :31
位域 c 的值为 :-1
```

在分析代码前，我们先来看看 data 的内存结构，如图 1-16 所示。

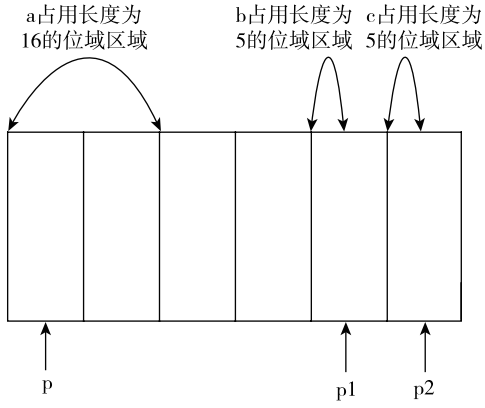


图 1-16　位域变量 data 的内存结构（5）

&data 为 data 位域结构的起始地址，将其强制转换为 int 型指针，并赋值给 p，所以 p 的值就是 data 位域的起始地址，即 4233624，p 指针指向的就是以 4233624 为起始地址的连续 4 个字节的内存单元；接下来执行“char *p1=(char*)(p+1);”使 p1 的值为 4233628，p1 就指向地址为 4233628 的内存单元；执行“p2 = p1+1;”使 p2 的值为 4233629，char 型指针指向地址为 4233629 的内存单元。我们发现，*p 的值和位域 a 的值相同。由此可以看出，VC++ 6.0 在编译的时候，对于那些没有使用的位域段，编译器对其进行填充 0 的处理。看看位域 c 的运行结果，我们发现输出与输入不相符，这是因为在编译的过程中对 char 型位域默认执行有符号处理，所以输出值为 −1，而对位域 b 指定了无符号的处理方式，所以输出与输入完全一致。

第 2 章

预 处 理

如果只是为了应付考试而学习 C 语言，那么可能不用对 C 语言中的预处理知识了解得太深，但是我们不能因此轻视预处理部分的知识点，因为预处理是 C 语言的一个重要知识点，能改善程序设计的环境，有助于编写易移植、易调试的程序。我们有必要掌握好预处理命令，以便在编程时灵活地使用它，使编写的程序结构优良，更易于调试和阅读。接下来，我尽可能地将预处理中的重要知识点和那些易错点向读者讲解清楚，使读者能够在自己以后的编程中熟练使用预处理命令。

2.1 文件的包含方式

在 C 语言代码中，我们可能会经常见到以下两种头文件的引用方式：

❑ #include " 文件名 "

❑ #include < 文件名 >

这两种引用方式之间的区别就在于，在以 < 文件名 > 方式引用的时候，如果采用 VC++ 6.0 进行编译，那么会先在系统头文件目录中查找，若查找失败，再到当前目录中查找，还查找不到则报错；如果在 Linux 环境下采用 gcc 进行编译，那么仅在系统头文件目录中查找，查找不到则报错（这就是后面采用 <print.h> 方式引用自定义的 print.h 头文件编译失败的原因）。以“文件名”方式引用的时候，不管是用 VC++ 6.0 还是用 gcc 编译，编译时都先在当前目录中查找，如果查找失败，再到系统头文件目录中查找，还查找不到则报错。

接下来，我们通过下面两段代码来具体分析。

第一段：

```
#include "stdio.h"

int main()
{
   printf("Hello World!\n");

   return 0;
}
```

VC++ 6.0 编译运行的结果：

```
Hello World!
```

在 Linux 环境下用 gcc 编译运行的结果：

```
Hello World!
```

第二段：

```
#include <stdio.h>

int main()
{
   printf("Hello World!\n");
```

```
    return 0;
}
```

VC++ 6.0 编译运行的结果：

```
Hello World!
```

在 Linux 环境下用 gcc 编译运行的结果：

```
Hello World!
```

我们发现这两种引用方式没有任何区别，都能成功打印出“Hello World！”。适当修改一下代码，定义一个头文件 print.h，在 print.h 头文件中定义一个 print() 函数，实现打印输出“Hello World！”。在 main() 函数源文件中加入 print.h 头文件，然后在 main() 函数中调用 print() 函数实现打印输出。其中，print.h 头文件的代码如下：

```
#include <stdio.h>

void print()
{
    printf("Hello World!\n");

    return ;
}
```

然后用两种方法来引用 print.h 头文件。

方法一：以 #include "print.h" 方法来引用，代码如下。

```
#include "print.h"

int main()
{
    print();

    return 0;
}
```

VC++ 6.0 编译运行的结果：

```
Hello World!
```

在 Linux 环境下用 gcc 编译运行的结果：

```
Hello World!
```

方法二：以 #include <print.h> 方法来引用，代码如下：

```
#include <print.h>

int main()
{
    print();
```

```
    return 0;
}
```

VC++ 6.0 编译运行结果：

```
Hello World!
```

在 Linux 环境下，gcc 编译运行时出错，错误信息：

```
main.c:1:19: fatal error: print.h: No such file or directory
compilation terminated.
```

我们发现，通过 VC++ 6.0 编译运行时，两种头文件的引用方法没有区别。但是在 Linux 环境下采用 gcc 编译运行的时候，对于系统头文件，使用两种方法都可以，但是对于自己定义的头文件，只能使用 #include “文件名”的方式。

2.2 宏定义

宏定义又称为宏替换，简称宏。它是在预处理阶段用预先定义的字符串替代标识符的过程。其定义的一般形式为：

```
#define          标识符          字符串
```

宏定义中的标识符都采用大写，这是编程中一种约定俗成的习惯。在了解如何使用宏定义之前，我们先来了解使用宏的过程中需要注意的几个要点。

- ❑ 宏替换不做语法检查，所以在使用的时候要格外小心。
- ❑ 宏替换通常在文件开头部分，写在函数的花括号外边，作用域为其后的程序，直到用 #undef 命令终止宏定义的作用域。
- ❑ 不要在字符串中使用宏，如果宏名出现在字符串中，那么将按字符串进行处理。

2.2.1 简单宏替换

简单宏替换在编程中通常用来定义常量。如在编程中多次用到同一个常量时，我们可以为该常量定义一个宏名，以便只修改赋值语句中的值就可以实现对程序中所有该宏名出现处的值进行修改。同时，使用宏名还可以使程序的可读性得以提升。

1. 简单宏定义的优点

（1）减少不必要的修改，提升程序的可预读性

如涉及圆周率，我们可以采用如下的宏定义来实现：

```
#define          PI          3.1415
```

如果需要修改圆周率的精度，只需要在宏定义中修改就可以实现了，而不必到程序中逐个修改所有用到的圆周率。同时，采用宏的方式比直接采用数值的方式使程序更容易理解，可读性增强了。

（2）提升代码的可移植性

使用宏定义不仅仅可以增强代码的可读性，还可以提高代码的可移植性，如：

```
#define           INT_SIZE        sizeof(int)
```

在某些编译环境下，sizeof(int) 的值可能为 4，但是这并不代表在所有的环境下它的运行结果都为 4，也可能为 2 或 8，所以针对此类情况，为了提高代码的可移植性，在编程时最好采用宏定义的方式。

2. 使用简单宏定义要注意的问题

在宏定义中，有一点是我们不得不注意的，那就是宏定义仅仅是简单宏替换，它不负责任何计算顺序，这就可能不小心带来难以查找的错误，所以在使用宏定义计算表达式的值时要格外小心，如：

```
#define           A               12+12
#define           B               10+10
```

当定义了以上的宏定义之后，如果在代码中执行 A*B，那么在预处理阶段，A*B 将被扩展为 12+12*10+10，从扩展后的表达式可知得到的结果并不是我们所期望的值，所以要想得到正确的结果，在宏定义的时候可以采用以下方式。

```
#define           A               (12+12)
#define           B               (10+10)
```

在宏定义中也可以通过 #undef 来设定宏名的作用域，我们可以通过以下代码来具体看看 #undef 的使用。

```
#include <stdio.h>

#define N 9

void main ()
{
    int i,a[N];
    for(i=0;i<N;i++)
    {
        a[i]=i;
        printf("a[%d]=%d\t",i,a[i]);
        if((i+1)%3==0)
            printf("\n");
    }

//#undef N
    printf("%d\n",N);

    return ;
}
```

运行结果为：

```
a[0]=0  a[1]=1  a[2]=2
a[3]=3  a[4]=4  a[5]=5
a[6]=6  a[7]=7  a[8]=8
9
```

由于通过 #undef 可以设定宏名的作用域，当在以上代码中注释掉“#undef N”时，接下来的打印语句能够正常打印出 N 的值；而没有注释掉“#undef N”时，由于此时 N 的作用域结束，所以接下来在打印语句部分就会出现“error C2065: 'N' : undeclared identifier”错误，提示 N 没有定义。由此可以看出，在编程时可以用 #undef 来设定定义的宏的作用域。

2.2.2 带参数的宏替换

带参数的宏替换，其定义的一般形式为：

```
#define          宏名（参数表）          字符串
```

在讲解带参数的宏的使用之前，同样先来看看使用带参数的宏时需要注意的几点。

- ❑ 宏名和参数表的括号间不能有空格。
- ❑ 宏替换只做替换，不做计算和表达式求解，这一点要格外注意。
- ❑ 函数调用在编译后程序运行时进行，并且分配内存。宏替换在编译前进行，不分配内存。
- ❑ 宏的哑实结合（哑实结合类似于函数调用过程中实参替代形参的过程）不存在类型，也没有类型转换。
- ❑ 宏展开使源程序变长，而函数调用则不会。

下面通过 Linux 下的两个典型的宏定义来介绍带参数的宏定义。说其典型，是由于其宏定义的“完整性”，至于“完整性”究竟体现在什么地方，我们通过代码来逐一分析。

```
#define min(x,y) ({ typeof(x) _x = (x); typeof(y) _y = (y); (void) (&_x == &_y);
 _x < _y ? _x : _y; })

#define max(x,y) ({ typeof(x) _x = (x); typeof(y) _y = (y); (void) (&_x == &_y);
 _x > _y ? _x : _y; })
```

在上面的两个宏中都有代码“(void) (&_x == &_y);”，可能不少读者对其并不理解，下面进行仔细分析。首先分析“==”，这是一个逻辑表达式的运算符，它要求两边的比较类型必须一致。如果 &x 和 &y 的类型不一致，一个为 char*，另一个为 int*，那么使用 gcc 编译就会出现警告信息，用 VC++ 6.0 编译时则会报错“error C2446: '==' : no conversion from 'char *' to 'int *'”。代码“(void) (&_x == &_y); ”的功能就相当于执行一个简单的判断操作，判断 x 和 y 的类型是否一致。别小看了这句代码，学会使用它会为编码带来不少便捷。下面给出一个小示例。

```
#include<stdio.h>

void print()
{
    printf("hello world!!!\n");
```

```
    return ;
}

void main(int argc,char*argv)
{
    print();

    return ;
}
```

运行结果：

```
hello world!!!
```

现在适当修改一下上面的代码。

```
#include<stdio.h>

void print()
{
    printf("hello world!!!\n");
    return ;
}

void main(int argc,char*argv)
{
    #define print() ((void)(3))
    print();

    return ;
}
```

运行结果没有任何输出。

这次的结果没有了之前的那句“hello world!!!”，可以看出此时函数并没有被调用，这是因为“#define print() ((void)(3))”使之后的调用函数 print() 成为一个空操作，所以这个函数在接下来的代码中都不会被调用了，就像被“冲刷掉”了一样。看了上面给出的宏，细心的读者会有另外一个疑惑：在“#define min(x,y) ({ typeof(x) _x = (x); typeof(y) _y = (y); (void) (&_x == &_y); _x < _y ? _x : _y; })”中，为什么要使用“typeof(y) _y = (y)”这样的替换，而不直接使用“typeof(x)==typeof(y)”或者“x < y ? x : y;”呢？因为使用“typeof(x)==typeof(y)”就像使用“char==int”一样，这是不允许的。如果在宏中没有使用“(void) (&_x == &_y);”这样的语句，那么编译时就相当于失去了类型检测功能。在上面的宏中使用“typeof(y) _y = (y)”这样的转换是为了防止 x 和 y 为一个表达式的情况，如 x=i++，如果不转换，那么 i++ 就会多执行几次操作，得到的就不是想要的结果。如果使用了“typeof(y) _y = (y)”这样的转换，就不会出现这样的问题了。我们可以通过下面一段代码来看看它们之间的区别。

```
#include <stdio.h>

#define min(x,y) ({ typeof(x) _x = (x); typeof(y) _y = (y); (void) (&_x == &_y);
```

```
 _x < _y ? _x : _y; })

#define min_replace(x,y) ({ x < y ? x : y; })

void main()
{
   int x=1;
   int y=2;

   int result = min(x++,y);
   printf(" 没有替换时的运行结果为 :%d\n",result);

   int x1=1;
   int y1=2;
   int result1 = min_replace(x1++,y1);
   printf(" 替换之后的运行结果为：%d\n",result1);

   return ;
}
```

在 Linux 环境下使用 gcc 编译的运行结果：

```
没有替换时的运行结果为 :1
替换之后的运行结果为：2
```

分析上面的运行结果可以发现，使用相同输入的两种宏得到的最终结果并不一样，在 2.3 节中我们还会对其进行详细分析。

下面来看如何使用宏定义实现变参，先看看实现方法。

```
#define    print(...)   printf(__VA_ARGS__)
```

在这个宏中，“...”指可变参数。可变参数的实现方式就是使用“...”所代表的内容替代 __VA_ARGS__，看看下面的代码。

```
#include<stdio.h>

#define print(...)   printf(__VA_ARGS__)

int main(int argc,char*argv)
{
   print("hello world----%d\n",1111);

    return 0;
}
```

在 Linux 环境下采用 gcc 进行编译的运行结果：

```
hello world----1111
```

再看代码：

```
#define printf (tem, ...) fprintf (stdout, tem, ## __VA_ARGS__)
```

可能有些读者对 fprintf() 函数感觉有些陌生，在此对 fprintf() 函数进行简单的讲解，其函数原型为：

```
int printf(FILE *stream,char *format [,argument])
```

这个函数的功能为根据指定的 format 格式发送消息到 stream（流）指定的文件中，在前面的宏中使用 stdout 表示标准输出，fprintf() 的返回值是输出的字符数，发生错误时返回一个负值。

```
#include<stdio.h>

#define print(temp, ...) fprintf(stdout, temp, ##__VA_ARGS__)

int main(int argc,char*argv)
{
   print("hello world----%d\n",1111);

   return 0;
}
```

在 Linux 环境下采用 gcc 进行编译的运行结果：

```
hello world----1111
```

temp 在此处的作用为设定输出字符串的格式，后面的“...”为可变参数。现在问题来了，在宏定义中为什么要使用“##”呢？如果没有使用 ##，会怎么样呢？看看下面的代码：

```
#include<stdio.h>

#define print(temp, ...) fprintf(stdout, temp, __VA_ARGS__)

int main(int argc,char*argv)
{
   print("hello world\n");

   return 0;
}
```

在 Linux 环境下采用 gcc 进行编译时发生了如下错误：

```
arg.c: In function 'main':
arg.c:7:2: error: expected expression before ')' token
```

为什么会出现上述错误呢？现在我们来分析一下。进行宏替换，“print("hello world\n")”变为“fprintf(stdout, "hello world\n",)”后，会发现后面出现了一个逗号导致发生错误。如果有“##”，就不会出现这样的错误，这是因为可变参数被忽略或为空，“##”操作将使预处理器去除它前面的那个逗号。如果存在可变参数，“##”也能正常工作。

介绍了“##”，再来介绍一下“#”。先来看看下面一段代码。

```
#include<stdio.h>

#define return_exam(p) if(!(p)) \
```

```
    {printf("error: "#p" file_name:%s\tfunction_name:%s\tline:%d .\n",\
    __FILE__, __func__, __LINE__); return 0;}

int print()
{
    return_exam(0);
}

int main(int argc,char*argv)
{
    print();
    printf("hello world!!!\n");

    return 0;
}
```

在 Linux 环境下采用 gcc 进行编译的运行结果：

```
error: 0 file_name:arg.c    function_name:print line:9 .
hello world!!!
```

因为这里只是为了体现要讲解的宏，所以对代码做了最大的简化，后续章节还将深入讲解如何使用宏来调试代码。“#”的作用就是对其后面的宏参数进行字符串化操作，即在对宏变量进行替换之后在其左右各加上一个双引号，这就使得“"#p"”变为了“""p""”，我们发现这样两边的“"""”就消失了。

2.2.3 嵌套宏替换

所谓嵌套宏替换，就是指在一个宏的定义中使用另外一个宏。关于嵌套宏替换的具体使用，可以看看下面的宏定义：

```
#define           N    3
#define           N_CUBE  N*N*N
#define           CUBE_ABS              ((N_CUBE>0) ? ( N_CUBE) : -1*( N_CUBE))
```

嵌套宏替换在预处理阶段进行扩展的时候是逐层进行的，以上面的 CUBE_ABS 为例，在预处理阶段将对其中的每个宏名进行扩展，直到宏定义中没有宏名为止。

2.3 宏定义常见错误解析

在前面讲解带参数的宏定义和不带参数的宏定义时，只是简单提示了使用宏的一些注意事项，并没有详细地分析，下面针对带参数的宏定义和不带参数的宏定义的注意事项进行讲解。

2.3.1 不带参数的宏

下面先通过一段代码来看看不带参数的宏定义中容易被忽略的地方。

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

```
#define     INT_P         int *

void main()
{
   int            i,j;
   int            a[9];
   INT_P   p;

   for(i=0;i<9;i++)
   {
          a[i]=i+1;
   }

   for(j=0,p=a;p<a+9;p++)
   {
          printf("a[%d]=%d\t",j++,*p);
          if(0==j%3)
                 printf("\n");
   }

   return ;
}
```

运行结果：

```
a[0]=1  a[1]=2  a[2]=3
a[3]=4  a[4]=5  a[5]=6
a[6]=7  a[7]=8  a[8]=9
```

在上面的代码中，宏定义部分使用了 #define INT_P int *，所以接下来定义整型指针时只需要使用 INT_P 定义一个整型指针即可。我们发现，使用这种方式定义的类型名更具可读性。接下用指针 p 来打印出数组中的每个元素，如果使用它来定义多个变量，就会出现问题。修改下上面的代码，使用 INT_P 来定义多个变量。

```
#include <stdio.h>

#define INT_P     int *

void main()
{
   int            i,j;
   int            a[9];
   INT_P   p,p1;

   for(i=0;i<9;i++)
   {
          a[i]=i+1;
   }

   for(j=0,p1=a;p1<a+9;p1++)
   {
          printf("a[%d]=%d\t",j++,*p1);
```

```
            if(0==j%3)
                    printf("\n");
    }

    return ;
}
```

在用 INT_P 定义整型指针 p 时多定义了一个 p1，且没有使用整型指针 p 来打印数组 a 中的每个元素，而是把 p1 当成整型指针来用，使用 p1 来打印数组 a 中的每个元素，结果在编译的时候出错了。看看其中一条主要的错误提示信息：

```
error C2440: '=' : cannot convert from 'int [9]' to 'int'
```

为什么会出现上面的错误呢？现在来分析下，错误提示 p1 是一个整型变量，并非我们想要的整型指针。我们进行一下宏扩展，将“INT_P p,p1;”扩展为“int* p,p1;”，这样就可以清晰地发现问题的所在了，原来，在 p 之后定义的 p1 并非我们想要的整型指针，而是一个整型变量。因此在使用宏定义来定义想要的类型时要注意，对没有把握的地方最好进行一下宏扩展，分析扩展开的代码。

当然，难免有读者会犯下面的这种错误。

```
#include <stdio.h>

#define N   10;

void main()
{
    printf("N 的值为 :%d\n",N);

    return ;
}
```

编译运行的时候提出如下错误：

```
error C2143: syntax error : missing ')' before ';'
error C2059: syntax error : ')'
```

进行一下宏扩展就会发现，在宏定义部分多了一个分号。将“printf("N 的值为 :%d\n",N);”扩展为“printf("N 的值为 :%d\n",10;);”，清楚地发现后面多了一个分号。

读者还要注意的一点是，不要在字符串中使用宏，如果宏名出现在字符串中，那么将把宏按照字符串来处理，例如：

```
#include <stdio.h>

#define STR        "Hello World!"

void main()
{
    char *STRING="This is a string!!!";
    printf(" 字符串中的宏 %s\n","STR!");
    printf(" 字符串中的宏 :STR 和不在字符串中的宏 : %s\n",STR);
```

```
    printf(" 出现在字符串变量名中的宏 : %s\n",STRING);

    return ;
}
```

运行结果：

```
字符串中的宏 STR!
字符串中的宏 :STR 和不在字符串中的宏 : Hello World!
出现在字符串变量名中的宏 : This is a string!!!
```

从上面的运行结果可以发现，出现在字符串中的宏被编译器按照字符串来处理了，因此在使用宏时不能在字符串中使用宏，否则宏将被当成一般字符串来处理。

2.3.2 带参数的宏

下面介绍带参数的宏在定义时的一些注意事项。有不少读者在编写程序求两数之和时通常会使用下面的方法。

```
#include <stdio.h>

#define SUM(x,y) x+y

void main()
{
    int x=6;
    int y=9;
    int s=SUM(x,y);

    printf("x+y 的值为 :%d\n",s);

    return ;
}
```

运行结果：

```
x 和 y 中较大的数为 :15
```

上述代码此时没有任何问题，但是适当修改一下代码。将“int s=SUM(x,y)”修改为“int s=SUM(x,y)*10”。此时的运行结果为：

```
x+y 的值再乘以 10 为 :96
```

我们发现，结果跟想要的不相符，本意是先求 x+y 的值，再乘以 10，结果应该为 150，而这里得到的是 96。还是通过宏扩展来查看出错的原因，将“int s=SUM(x,y)*10;”扩展为“int s=x+y*10;”，我们发现，扩展之后的表达式跟我们的初衷相差甚远。可以进一步修改上面的宏定义的实现方法，将其中的宏定义“#define SUM(x,y) x+y”修改为“#define SUM(x,y) (x+y)”，此时的运行结果为：

```
x+y 的值再乘以 10 为 :150
```

这时得到的就是我们想要的结果。只是在宏定义部分加了一个括号，以保证在进行宏扩展时 x+y 是一个整体，不会被拆开。

再来看括号在宏定义中的另一种使用方法。在编写求两个数之差的绝对值的时候，不少人会采用以下宏定义的实现方法。

```
#include <stdio.h>

#define SUB_ABS(x,y) x>y?x-y:y-x

void main()
{
   int x=-6;
   int y=-9;
   int abs=SUB_ABS(x,y);

   printf("x 和 y 之差的绝对值为 :%d\n",abs);

   return ;
}
```

运行结果：

```
x 和 y 之差的绝对值为 :3
```

运行结果是我们想要的 3，乍一看，上面的宏定义没有什么问题，现在一步步地找出它存在的问题。修改上面的代码，将其中的“int abs=SUB_ABS(x, y);”修改为“int abs=SUB_ABS(x+y, x-y);”。此时的运行结果为：

```
x+y 和 x-y 之差的绝对值为 :0
```

这时候就出现问题了，0 不是我们想要的结果，动手算算就知道得到的结果应该为 18，还是用宏扩展的老方法，将“int abs=SUB_ABS(x+y,x-y);”扩展为“int abs=x+y>x-y?x+y-x-y:x-y-x+y;”，宏扩展后的结果显然是 0。所以应该将其宏定义“#define SUB_ABS(x,y) x>y?x-y:y-x”修改为“#define SUB_ABS(x,y) (x)>(y)?(x)-(y):(y)-(x)”。此时的运行结果为：

```
x+y 和 x-y 之差的绝对值为 :18
```

这时得到的才是正确的结果，是不是这样的宏定义就完全正确呢？当然不是的，如果将其中的“int abs=SUB_ABS(x+y, x-y);”修改为“int abs=SUB_ABS(x+y,x-y)*0;”，此时的运行结果为：

```
x+y 和 x-y 之差的绝对值乘以 0 的值为 :3
```

通过上述结果我们就发现上面的宏定义仍然存在问题，相信这时读者应该知道问题的所在了，因为没有使用 () 将条件表达式表示成为一个整体，所以出现了错误的结果 3。进一步修改上面的代码，将其中的宏定义“#define SUB_ABS(x,y) x>y?x-y:y-x”修改为“#define SUB_ABS(x,y) ((x)>(y)?(x)-(y):(y)-(x))”，此时的运行结果为：

```
x+y 和 x-y 之差的绝对值乘以 0 的值为 :0
```

这时得到的就是想要的结果。

以上从不同方面分析了带参数的宏定义的注意事项。在讲解带参数的宏定义时候，特别提到了关于参数替换的问题。在此，同样通过修改上面的代码来分析。

```
#include <stdio.h>

#define SUB_ABS(x,y) ((x)>(y)?(x)-(y):(y)-(x))

void main()
{
   int x=-6;
   int y=-9;
   int abs=SUB_ABS(++x,y);

   printf("++x 和 y 之差的绝对为 :%d\n",abs);

   return ;
}
```

运行结果：

```
++x 和 y 之差的绝对值为 :5
```

上述代码的意思是求 −5 和 −9 之差的绝对值，正确的结果应该为 4。下面进行宏扩展来查看出错的原因，将“int abs=SUB_ABS(++x,y);”扩展为“int abs=((++x)>(y)?(++x)-(y):(y)-(++x));”后可以发现，不管输入的 x 和 y 之间是什么样的大小关系，x 自加运算都执行两次，比预期多执行了一次，所以最终得到的是错误的结果。如果对参数进行替换，例如：

```
#include <stdio.h>

#define SUB_ABS(x,y) ({typeof(x) _x=x;typeof(y) _y=y;(_x)>(_y)?(_x)-(_y):(_y)-(_y);})

void main()
{
   int x=-6;
   int y=-9;
   int abs=SUB_ABS(++x,y);

   printf("++x 和 y 之差的绝对为 :%d\n",abs);

   return ;
}
```

在 Linux 环境下采用 gcc 进行编译的运行结果：

```
++x 和 y 之差的绝对为 :4
```

由于 VC++ 6.0 不支持 typeof 操作符，所以在 Linux 环境下使用 gcc 编译运行时，typeof 操作符的功能是得到变量的数据类型，这时得到的结果才与我们的意图相符。

所以在代码中要特别注意带参数宏的使用，否则可能带来一些意想不到的错误，为代

码调试带来很多的麻烦。当然，最好的方法就是采用宏扩展的方式来看看是否存在宏定义的错误。

从上面对宏定义的常见错误分析可以看出，在使用宏定义的时候尤其要注意括号的灵活使用，如果不小心使用，可能给我们的程序带来意想不到的结果。同时，由于宏定义不进行语法检测，所以相对来说进行查错的难度就大大地增加了。在定义带参数的宏定义时，需要注意参数是否涉及自加自减运算，如果代码中的参数可能涉及自加自减运算，那么最好进行参数的替换，以免自加自减运算对运行结果带来影响。

2.4 条件编译指令的使用

预处理程序提供了条件编译的功能，用户可以选择性地编译程序，进而产生不同的目标代码文件，这对程序的移植和调试来说是非常有用的。下面先来看看条件编译命令的几种使用方式。

第一种方式：

```
#if   常量表达式
                程序段 1;
[#else
                程序段 2; ]
#endif
```

功能：当常量表达式为非 0（“逻辑真”）时，编译程序段 1，否则编译程序段 2。

第二种方式：

```
#ifdef  标识符
                 程序段 1;
[#else
                 程序段 2; ]
#endif
```

功能：如果标识符已经被 #define 命令定义过，则编译程序段 1，否则编译程序段 2。

第三种方式：

```
#ifndef  标识符
                 程序段 1;
[#else
                 程序段 2; ]
#endif
```

功能：如果标识符未被 #define 命令定义过，则编译程序段 1，否则编译程序段 2。

了解了条件编译指令的使用方式之后，我们在调试代码的时候，就不能再随心所欲地删减代码了。如果不希望某段代码被编译，那么可以使用条件编译指令来将其注释掉，例如：

```
#if (0)
                 注释代码段;
#endif
```

这样就可以将代码注释掉。需要的时候还可以将注释掉的代码重新启用，不必为需要重新编辑代码时发现代码已被删除而头疼了。下面通过具体的代码来了解条件编译命令的使用。

```
#include<stdio.h>

#define    NUM            0
#define ON_OFF    0

int main(int argc,char*argv)
{
   #if NUM>0
        printf("NUM 的值大于 0\n");
   #elif NUM<0
          printf("NUM 的值小于 0\n");
   #else
        printf("NUM 的值等于 0\n");
    #endif

   #if     ON_OFF
          printf(" 使用条件编译命令注释掉的语句部分 \n");
   #endif

    return 0;
}
```

运行结果：

```
NUM 的值等于 0
```

通过上面的代码，我们学会如何使用条件编译命令。值得注意的是，常量表达式在编译时求值，所以表达式只能是常量或者已经定义过的标识符，不能是变量，也不能是那些在编译时候求值的操作符，如 sizeof。

看看下面的代码。

```
#include<stdio.h>

#define N 1

int main(int argc,char*argv)
{
    int a=3;

    #if(a)
        printf("#if 后面的表达式为变量 \n");
    #endif

    #if(N)
        printf("#if 后面的表达式已定义，且不为 0---success\n");
    #else
        printf("#if 后面的表达式已定义，且不为 0---fail\n");
    #endif
```

```
    return 0;
}
```

运行结果：

```
#if 后面的表达式已定义，且不为 0---success
```

从上面的代码我们发现，当表达式为变量 a 时，并没有打印出其后面的语句来，所以不能在其后的常量表达式中使用变量。如果使用 sizeof 操作符会怎么样呢？为了加深印象，我们来看下面的代码。

```
#include<stdio.h>

int main(int argc,char*argv)
{
    int a=9;
    #if(sizeof(a))
        printf("#if 后面的表达式含有 sizeof 操作符 \n");
    #endif

    return 0;
}
```

编译时产生了如下错误：

```
fatal error C1017: invalid integer constant expression
```

所以，在使用条件编译时要牢记：常量表达式不能是变量和含有 sizeof 等在编译时求值的操作符，在使用条件编译命令时尤其要注意。接下来看看另外两种条件编译命令的使用。

```
#include<stdio.h>

#define    NUM

int main(int argc,char*argv)
{

   #ifdef NUM
        printf("NUM 已经定义过了 \n");
   #else
        printf("NUM 没有定义过 \n");
   #endif

   return 0;
}
```

运行结果：

```
NUM 已经定义过了
```

在编写程序时，对于那些不确定是否已经定义过的宏采用这种方法来测试输出。适当地修改上面的代码，再看看另外一种实现方法。

```
#include<stdio.h>

#define    NUM

int main(int argc,char*argv)
{
   #undef  NUM
   #ifndef         NUM
        printf("NUM 没有定义过 \n");
   #else
        printf("NUM 已经定义过了 \n");
   #endif

   return 0;
}
```

运行结果：

```
NUM 没有定义过
```

在条件编译命令前面用“#undef NUM”取消了接下来的作用域中 NUM 宏的作用，所以接下来使用条件编译命令时打印出来的就是“NUM 没有定义过”。

2.5　#pragma 指令的使用

如果读者认真阅读了本书前面的代码，那么应该对 #pragma 指令有印象，在之前的代码中曾使用过 #pragma 指令来设置编译器的字节对齐方式。接下来看看预处理中的 #pragma 指令，其作用是设置编译器的状态或指示编译器完成一些特定的动作。使用 #pragma 指令的一般形式为：

```
#pragma  para
```

其中，para 为参数。下面对一些常见的参数进行讲解。

（1）#pragma message(" 消息 ")

至于“#pragma message(" 消息 ")”究竟有什么作用，可以通过下面的一段代码来了解其具体的使用方式。

```
#include<stdio.h>

#define STR

void main(int argc,char*argv)
{
   printf(" 学习 #pragma 命令中 message 参数的使用 !\n");

   #ifdef STR
          #pragma message("STR 已经定义过了 ")
   #endif
```

```
            return ;
}
```

在 Linux 环境下使用 gcc 编译运行的结果：

```
root@ubuntu:/home# gcc message.c -o msg
message.c: In function 'main':
message.c:10:11: note: #pragma message: STR 已经定义过了
root@ubuntu:/home# ./msg
学习 #pragma 命令中 message 参数的使用！
```

我们发现，在编译的时候会打印出 message 参数中的信息。通过这种方式，可以在代码中输出想要的信息，也可以看某个宏是否已经被定义过。与之前使用 printf() 函数实现打印的不同之处在于：message 打印消息出现在编译的时候，不会出现在程序最终的运行结果中；而 printf() 函数的打印消息却会出现在最终的运行结果中。有时候，我们并不希望运行结果中出现与结果无关的信息，这时可以使用 #pragma 命令，选择 message 参数来实现信息的打印输出。

（2）#pragma once

如果在头文件的开头部分加入这条指令，那么就能保证头文件只被编译一次。

（3）#pragma hdrstop

该指令表示编译头文件到此为止，后面的无需再编译了。

（4）#pragma pack()

我们在此前的代码中已经接触过这个指令了，但是没有进行详细的讲解。接下来了解使用这个参数的几个典型应用，看看下面的代码。

```
#include<stdio.h>

void main(int argc,char*argv)
{
   #pragma pack(2)

   struct  _stu1{
           char    name[20];
           char    num[10];
           int     score;
           char    sex;
   }stu1;

   printf("str1 占用内存的大小为 :%d 个字节 \n",sizeof(stu1));

   #pragma pack()

   struct  _stu2{
           char    name[20];
           char    num[10];
           int     score;
           char    sex;
   }stu2;
```

```
    printf("str2 占用内存的大小为 :%d 个字节 \n",sizeof(stu2));

     return ;
}
```

运行结果：

```
str1 占用内存的大小为 :36 个字节
str2 占用内存的大小为 :40 个字节
```

在上面的代码中，在结构体 st1 的前面使用了 #pragma pack(2)，其作用是设置 2 字节对齐，接下来使用了 #pragma pack()，其作用是取消之前设置的字节对齐方式，采用默认的 4 字节对齐。在输出结果中，由于 stu1 在内存中采用 2 字节对齐，而 stu2 在内存中采用 4 字节对齐，所以它们输出的结果不一致。将上面的代码修改为以下的形式。

```
#include<stdio.h>

void main(int argc,char*argv)
{
   #pragma pack(push)
   #pragma pack(2)

   struct  _stu1{
           char    name[20];
           char    num[10];
           int     score;
           char    sex;
   }stu1;

   printf("str1 占用内存的大小为 :%d 个字节 \n",sizeof(stu1));

   #pragma pack(pop)

   struct  _stu2{
           char    name[20];
           char    num[10];
           int     score;
           char    sex;
   }stu2;

   printf("str2 占用内存的大小为 :%d 个字节 \n",sizeof(stu2));

    return ;
}
```

运行结果与前面代码的运行结果完全一致。看看修改的地方，在设置 2 字节对齐方式之前添加了一句代码“#pragma pack(push)”，其作用是保存当前默认的字节对齐方式，而把下面原本的“#pragma pack()”修改为“#pragma pack(pop)”，其作用是恢复默认的字节对齐方式，可以看出这里代码的功能与之前代码的功能完全一致。

（5）#pragma warning()

“#pragma warning(disable：M N；once：H；error：K)”表示不显示 M 号和 N 号的警

告信息，H 号警告信息只报告一次，把 K 号警告信息作为一个错误来处理。也可以将其分开来实现，代码如下。

```
#pragma  warning(disable：M  N)
#pragma  warning(once：H)
#pragma  warning(error：K)
```

这样的实现方式与前面的“#pragma warning(disable：M N；once：H；error：K)”是等价的。也可以使用 #pragma warning(enable：N) 启用 N 号警告信息。

第 3 章

选择结构和循环结构的程序设计

C 语言有三种基本结构，分别是顺序结构、分支结构、循环结构，而本章的重点是介绍编程中较易出错的分支结构和循环结构。深入了解分支结构和循环结构，对查错和编写高质量的代码很有帮助。因此本章主要针对分支结构和循环结构在编程中的一些误区进行分析讲解，在编程时如何避开这些误区，是本章的重要知识点。

3.1 if 语句及其易错点解析

在前面的代码中，读者已多次接触 if 语句，在讲解 if 语句的使用要点之前，先简单回顾一下 if 语句的定义和使用的一般形式。

if 语句用来判断给定条件是否满足，根据判断结果决定是否执行某个操作。if 语句使用的一般形式为：

```
if （表达式）
语句段；
```

当表达式的值为真时，执行接下来的语句段，否则跳过该语句段部分继续执行。if 语句流程图如图 3-1 所示。

通常，if 语句还会包含 else 语句部分，如：

```
if   （表达式）
   语句段 1;
else
   语句段 2;
```

表达式的值为真时执行语句段 1，否则执行语句段 2。if-else 语句流程图如图 3-2 所示。

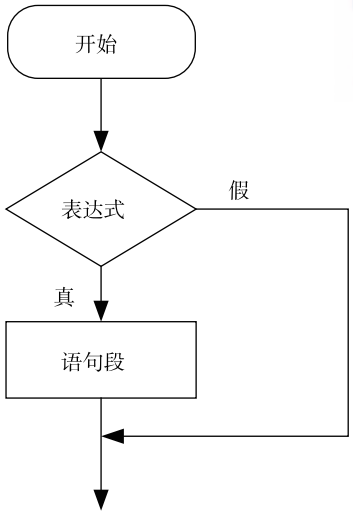


图 3-1　if 语句流程图

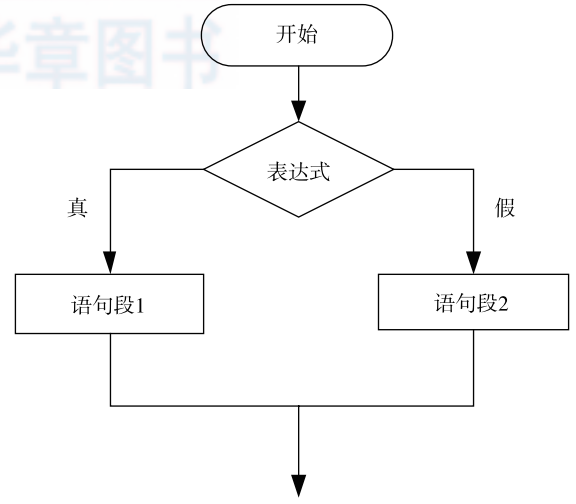


图 3-2　if-else 语句流程图

多重 if 语句嵌套的一般形式为：

```
if   (表达式1)
   语句段1;
else if(表达式2)
   语句段2;
else if(表达式3)
   语句段3;
   ……
else
   语句段n+1;
```

当表达式 N（N=1，2，…，n）的值为真时，执行其后的语句段 N，否则执行语句段 n+1。多重 if 语句流程图如图 3-3 所示。

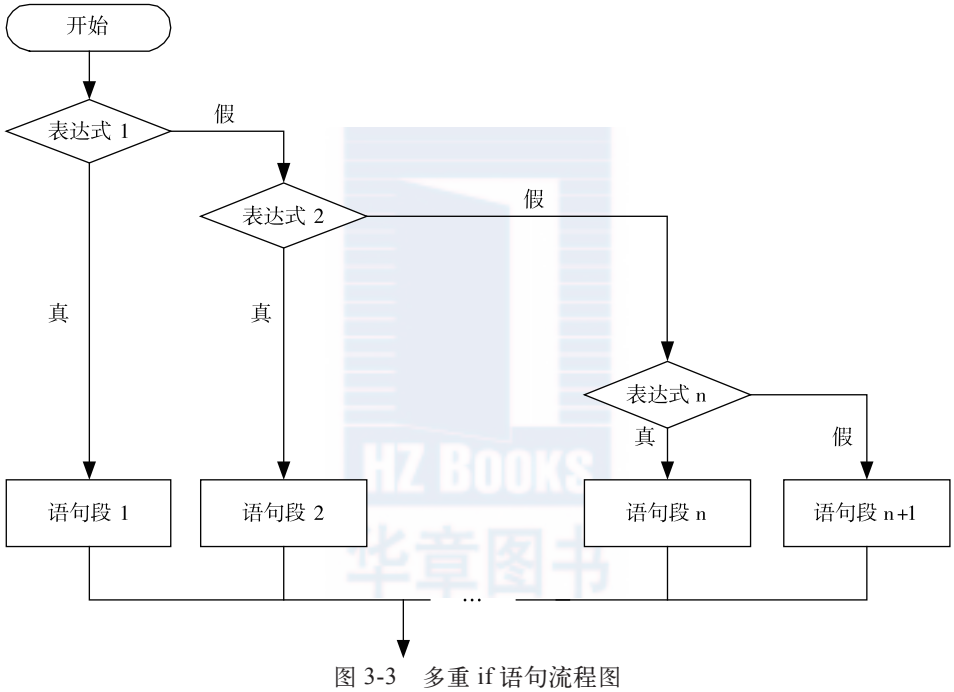


图 3-3　多重 if 语句流程图

了解了以上几种 if 语句结构，接下来看看在编程的过程中关于 if 语句的一些注意事项。

1. 条件表达式

细心的读者在阅读之前代码中的 if 语句时会发现，在表达式中通常把常量放在“==”的左边。这样写的好处是如果在编写代码的过程中不小心少写了一个“=”，那么编译时就会提示出现错误。因为在 C 语言中，赋值运算符的左值表示一个存储在计算机内存中的对象，不能是常量。看看下面的代码。

```
#include <stdio.h>

void main()
{
```

```
    int i,j;
    i=1;
    j=0;
    if(1==i)
            printf("i 的值为 1\n");
    if(j=1)
            printf("j 的值为 1\n");

    return ;
}
```

运行结果：

```
i 的值为 1
j 的值为 1
```

在上面的代码中，j 的初始值为 0，由于在 if 语句表达式中少写了一个“=”，导致结果显示“j 的值为 1”的错误信息，并且在编译代码时没有给出任何提示信息。如果写成如下的代码：

```
#include <stdio.h>

void main()
{
    int i;
    i=1;
    if(1=i)
            printf("i 的值为 1\n");

    return ;
}
```

编译代码时就会出现“error C2106: '=' : left operand must be l-value”错误，提示常量不能作为左值。由此可知，在使用常量作为左值时，如果因为疏忽少写了一个“=”，编译时会给出错误提示，但是如果按照一般的方法，把常量放在右边，且少写了一个“=”，编译器则不会给出任何提示，而是默认为一个赋值操作，将赋值操作的最终结果作为表达式的值。所以读者在平时编程时要牢记条件表达式中常量写在左边的语法规则，以防因为疏忽造成难以查找的错误。

2. 嵌套 if 语句

嵌套 if 语句的使用中最易出错的莫过于多个表达式之间的关系，在生活中经常会遇到类似下面的问题：将一个学生的数学成绩归类为优（90<=score<=100）、良（80<=score<90）、中（70<=score<80）、及格（60<=score<70）、差（score<60）。不少人会按照下面这两种方法来解决。

方法一：

```
if(score<60)
            printf(" 该学生的数学成绩类别为：差 \n");
    else if(score<70)
            printf(" 该学生的数学成绩类别为：及格 \n");
    else if(score<80)
```

```
            printf("该学生的数学成绩类别为：中\n");
    else if(score<90)
            printf("该学生的数学成绩类别为：良\n");
    else if(score<=100)
            printf("该学生的数学成绩类别为：优\n");
    else
            printf("输入出错!!!\n");
```

方法二：

```
if(score<=100)
            printf("该学生的数学成绩类别为：优\n");
    else if(score<90)
            printf("该学生的数学成绩类别为：良\n");
    else if(score<80)
            printf("该学生的数学成绩类别为：中\n");
    else if(score<70)
            printf("该学生的数学成绩类别为：及格\n");
    else if(score<60)
            printf("该学生的数学成绩类别为：差\n");
    else
            printf("输入出错!!!\n");
```

分析上面的两种实现方法，首先看方法一，如果输入的学生成绩在正常的范围内，那么能得到正确的结果。但是由于方法一的条件表达式范围并不严格，因此当输入一个负数时，将会视其为不及格。对于方法二，只会出现两种信息，一种就是输入的学生成绩为优，另一种就是输入出错，因此这种方法是错误的。这里建议在采用嵌套 if 语句的时候，对于条件表达式中的变量采用完整的范围限制，即解决上面的问题可以采用下面的方法。

```
#include <stdio.h>

void main()
{
    int score;
    printf("请输入学生的数学成绩:");
    scanf("%d",&score);

    if(score<60 && score>=0)
            printf("该学生的数学成绩类别为：差\n");
    else if(score<70 && score>=60)
            printf("该学生的数学成绩类别为：及格\n");
    else if(score<80 && score>=70)
            printf("该学生的数学成绩类别为：中\n");
    else if(score<90 && score>=80)
            printf("该学生的数学成绩类别为：良\n");
    else if(score<=100 && score>=90)
            printf("该学生的数学成绩类别为：优\n");
    else
            printf("输入出错!!!\n");

    return ;
}
```

运行结果：

```
请输入学生的数学成绩 :98
该学生的数学成绩类别为 : 优
```

分析上面的代码，在使用多重 if 语句的时候，如果表达式 N 的值为真，那么执行表达式 N 后面的语句段 N，在没有一个表达式的值为真的情况下，如果有 else 语句，那么就执行 else 后面的语句段，如果没有 else 语句，那么就执行 if 语句下面的语句段。

3. else 子句的配对

在讲解 else 子句的配对之前，先来看下面一段代码，其代码功能为按照先后顺序输入 A、B、C 三个数，如果 A<B<C，那么打印输出“输入数据呈现递增规律”的信息，否则打印输出“输入数据呈现非递增规律”的信息，代码如下：

```
#include <stdio.h>

int main(void)
{
   int A,B,C;

   printf(" 请依次输入 A、B、C 的值 :");
   scanf("%d%d%d",&A,&B,&C);

   if(A<B)
           if(B<C)
                   printf(" 输入数据呈现递增规律 \n");
   else
           printf(" 输入数据呈现非递增规律 \n");

   return 0;
}
```

先来看一种输入：

```
请依次输入 A、B、C 的值 :3 2 1
```

可以发现运行结果中并没有输出“输入数据呈现非递增规律”，这是为什么呢？对 if 语句很了解的读者应该很快就发现问题的所在，代码中 else 配对出现了问题，程序的本意是将 else 与第一个 if 语句配对，但是按照 if 语句的标准，else 应该与它前面最近的 if 语句配对。因此配对的 if 和 else 必须在同一个作用域内，在不同作用域内的 if 和 else 是不可以配对的。知道了错误的原因，修改上面的代码就很简单了。修改后的代码如下：

```
#include <stdio.h>

int main(void)
{
   int A,B,C;

   printf(" 请依次输入 A、B、C 的值 :");
   scanf("%d%d%d",&A,&B,&C);
```

```
    if(A<B)
            if(B<C)
                    printf(" 输入数据呈现递增规律 \n");
            else
                    printf(" 输入数据呈现非递增规律 \n");
    else
            printf(" 输入数据呈现非递增规律 \n");

    return 0;
}
```

运行结果：

```
请依次输入 A、B、C 的值 :3 2 1
输入数据呈现非递增规律
```

此时就能够得到正确的结果。其实，完全可以将 A、B、C 的大小比较放到一个表达式中，为了讲解 else 的配对问题，这里特地写成上面的形式。代码中第一个 else 与第二个 if 语句配对，第二个 else 与第一个 if 语句配对。在使用 if 语句的过程中要清楚 else 和 if 语句的配对关系。可以通过图 3-4 来了解 if-else 语句的配对关系。

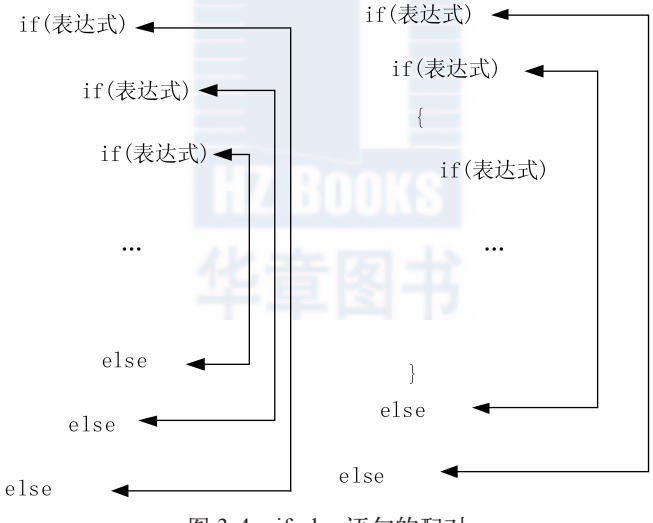


图 3-4　if-else 语句的配对

if 和 else 的配对准则是：else 与距离它最近的在同一个作用域内的没有被配对的 if 进行配对。对于图 3-4 中右边的多重 if 嵌套语句，由于最后一个 if 和第一个 else 不在同一个作用域内，因此不能进行配对。而图 3-4 中左边的第二个 else 没有与离它最近的 if 配对，因为这个 if 已经进行了配对，所以第二个 else 只能与第二个 if 进行配对。在写代码的过程中也要养成将那些配对的 if 和 else 起始位置放在同一列的习惯。

3.2 条件表达式的使用

在讲解条件表达式之前，先简要讲解一下条件运算符。条件运算符有两种：“?”和“：”。条件表达式的格式为：

```
表达式 1 ? 表达式 2：表达式 3
```

条件表达式的含义为，如果表达式 1 为真，那么条件表达式的值取表达式 2 的值，否则条件表达式的值取表达式 3 的值。图 3-5 中的流程图说明了条件表达式的功能。

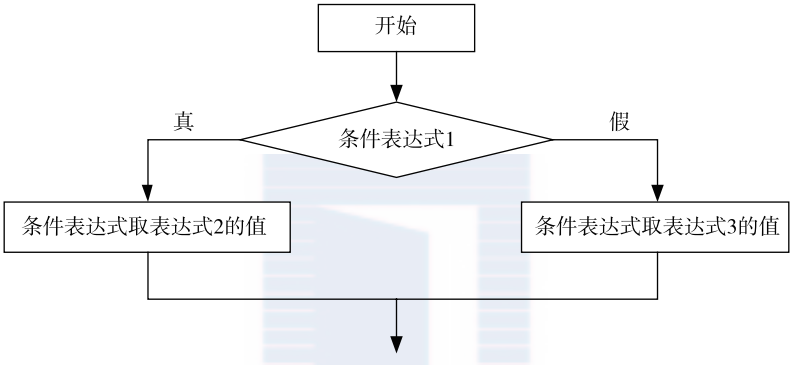


图 3-5 条件表达式流程图

在第 2 章讲解宏定义的时候就使用过条件表达式，还对条件表达式在使用过程中的注意事项做了讲解。下面通过具体的代码来了解条件表达式的使用。

```
#include <stdio.h>

int main(void)
{
   int a,b;
   int max1,max2;
   a=2;
   b=8;
   if(a>b)
           max1=a;
   else
           max1=b;

   max2=a>b?a:b;

   printf(" 使用 if 语句求出的 a、b 中的最大值为 :%d\n",max1);
   printf(" 使用条件表达式求出的 a、b 中的最大值为 :%d\n",max2);

   return 0;
}
```

运行结果：

```
使用 if 语句求出的 a、b 中的最大值为 :8
使用条件表达式求出的 a、b 中的最大值为 :8
```

看看上面的运行结果，使用 if 语句求出的 a、b 中的最大值和使用条件表达式求出的 a、b 中的最大值完全一样，这就说明可以用条件表达式来替换 if 语句。这点从条件表达式的流程图中也可以看出，但是使用 if 语句时可以嵌套，条件表达式是否可以嵌套呢？答案是肯定的。下面来看看使用条件表达式语句嵌套的一般格式。

```
表达式 1 ? 表达式 2:( 表达式 3 ? 表达式 4:( 表达式 5 ? 表达式 6:(……)))
```

嵌套条件表达式的流程图如图 3-6 所示。

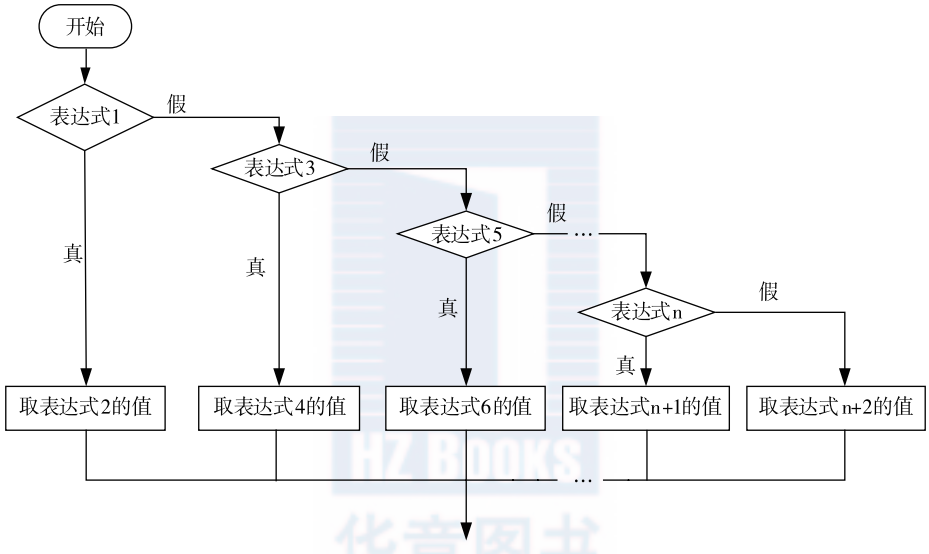


图 3-6　嵌套条件表达式流程图

看下面的代码，其功能为取 a、b、c 三个数的最大值。

```
#include <stdio.h>

int main(void)
{
   int a,b,c;
   int max1,max2;
   a=2;
   b=8;
   c=12;
   if(a>b)
           if(a>c)
                   max1=a;
           else
                   max1=c;
   else
           if(b>c)
```

```
                max1=b;
        else
                max1=c;

    max2=a>b?(a>c?a:c):(b>c?b:c);

    printf("使用 if 语句求出的 a、b、c 中的最大值为 :%d\n",max1);
    printf("使用条件表达式求出的 a、b、c 中的最大值为 :%d\n",max2);

    return 0;
}
```

运行结果：

```
使用 if 语句求出的 a、b、c 中的最大值为 :12
使用条件表达式求出的 a、b、c 中的最大值为 :12
```

从上面的代码中可以发现，实现同样的功能所使用的条件表达式要比 if 语句要短小很多，但是使用条件表达式也存在另外一个问题，那就是代码的可读性变差，所以在编程中要根据实际情况选择是否使用条件表达式，不要一味追求简短，使得代码的可读性很差。

在使用条件表达式的时候还要注意，不要对其中的变量随便使用自加和自减运算符，如：

```
#include <stdio.h>

int main(void)
{
    int a,b,max;
    a=7;
    b=5;
    max=a++>b?a:b;

    printf("使用 if 语句求出的 a、b 中的最大值为 :%d\n",max);

    return 0;
}
```

运行结果：

```
使用 if 语句求出的 a、b 中的最大值为 :8
```

发现运行结果较初始时 a 和 b 的比值发生了变化，所以在使用条件表达式的时候要尤其注意不要对变量使用自加和自减运算符，本书在讲解宏定义的时候也特地对其进行了深入的分析，如果读者对使用条件表达式的注意事项还是不够清楚，可以返回第 2 章关于宏定义注意事项的知识点，在此就不再过多讲解了。

3.3 switch 语句的使用及注意事项

虽然多重 if 语句可以替代 switch 语句，但是在某些时候使用 switch 语句使代码具有更好的可读性，避免了使用过多的 if-else 语句让人眼花缭乱。在讲解使用 switch 语句的注意事

项之前，先来看看它的使用格式。

```
switch    （表达式）{
   case           常量表达式 1:
        语句段 1;
        [break]
   case           常量表达式 2:
        语句段 2;
        [break]
   ……
   case           常量表达式 n:
        语句段 n;
        [break]
   default:
        语句段 n+1;
        [break]
}
```

下面来了解一下 switch 语句的执行过程。首先计算出表达式的值，如果某个 case 后面的常量表达式 N(N=1，2，…，n) 的值等于 switch 语句中表达式的值，那么就执行该 case 后面的语句段 N。值得注意的是，如果语句段 N 的后面有 break，那么执行完语句段 N 后就退出 switch 语句，否则将继续往下执行，直到遇到 break，如果没有 break，那么将执行到最后一条 switch 语句。

1.break 语句的使用

为了加深读者对 switch 语句执行过程的印象，接下来看一下如图 3-7 所示的 switch 语句流程图。

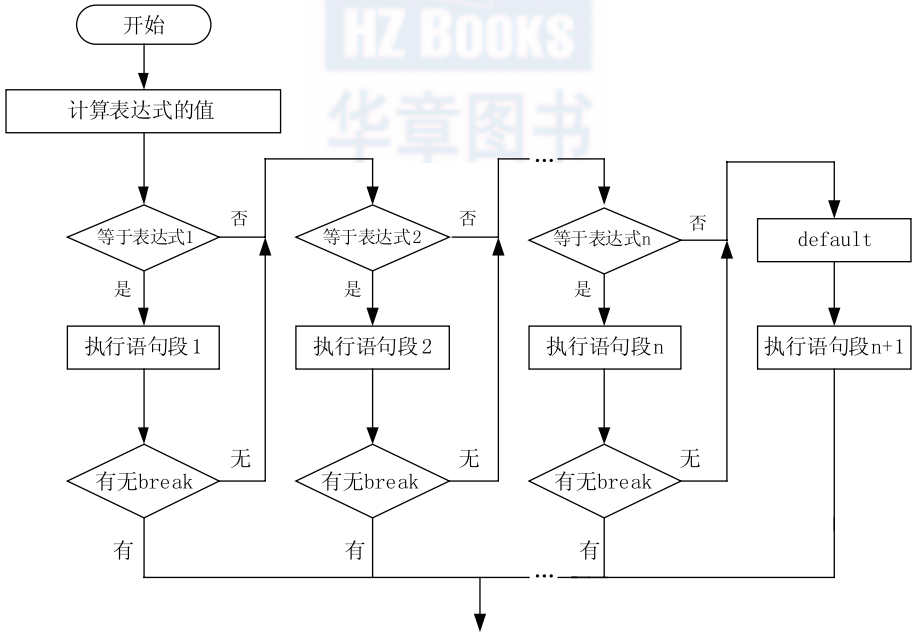


图 3-7　switch 语句流程图

通过上面的流程图，我们能够直观地看出 switch 语句的执行过程。接下来看看如何使用 switch 语句。用 switch 语句编写前面用多重 if 语句实现的输入学生的数学成绩并进行分类的程序，代码如下：

```
#include <stdio.h>

void main(void)
{
   int score;
   printf(" 请输入学生的数学成绩 :");
   scanf("%d",&score);
   if(score>100 || score<0)
   {
          printf(" 输入出错 !\n");
          return ;
   }
   switch(score/10){
          case   0:
          case   1:
          case   2:
          case   3:
          case   4:
          case   5:
                 printf(" 学生成绩类别为 : 差 \n");
                 break;
          case   6:
                 printf(" 学生成绩类别为 : 及格 \n");
                 break;
          case   7:
                 printf(" 学生成绩类别为 : 中 \n");
                 break;
          case   8:
                 printf(" 学生成绩类别为 : 良 \n");
                 break;
          default:
                 printf(" 学生成绩类别为 : 优 \n");
   }

   return ;
}
```

运行结果：

```
请输入学生的数学成绩 :29
学生成绩类别为 : 差
```

通过上面的运行结果可以发现，输入 29 时输出的类别为“差”。现在来分析一下这段代码中的 switch 语句的执行流程，先计算 29 整除 10 等于 2，表达式的值为 2，常量表达式为 2 的 case 后面没有任何语句段，那为什么能够输出正确的类别呢？这是因为在该语句段后面没有使用 break 来终止 switch 语句的执行，所以程序继续向下遍历常量表达式 3 和常量表达式 4，直到常量表达式 5，此时语句段后有 break，所以程序执行完输出后退出了 switch 语句

的执行。

2. 常量表达式的使用

在使用 switch 语句的时候还要注意的一点是，常量表达式必须由常量所构成，不能含有变量。同时，常量表达式的值必须互不相同，也就是说，同一个常量在 switch 语句中只能对应一种处理方案。在 switch 语句中，除了使用整型常量作为常量表达式之外，还可以使用字符。下面的代码实现的功能为：从键盘输入一个英文字符，不区分大小写，通过 switch 语句判断该字符是否为元音字符。

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

void main(void)
{
   char ch;
   printf(" 按下 esc 键即可结束运行 !\n");
   while(1)
   {
          printf(" 请输入字符 :");
          if((ch=getch())==27)
                 break;
          putchar(ch);

          if(ch>='a' && ch<='z')
          {
                 ch = ch-'a'+'A';
          }

          if(ch<'A' || ch>'Z')
          {
                 printf("\n 输入出错 ! 请重新输入 .\n");
                 continue;
          }

          switch(ch){
                 case    'A':
                 case    'E':
                 case    'I':
                 case    'O':
                 case    'U':
                        printf("\n 该字符为元音字符 !\n");
                        break;
                 default:
                        printf("\n 该字符非元音字符 !\n");
          }
   }

   return ;
}
```

运行结果：

```
按下 esc 键即可结束运行！
请输入字符 :4
输入出错！请重新输入 .
请输入字符 :d
该字符非元音字符！
请输入字符 :O
该字符为元音字符！
请输入字符 :u
该字符为元音字符！
请输入字符 :
```

为了便于处理，我们将输入的英文字符转换为大写。看上面的运行结果，因为一开始输入的数据不是所要的测试类型，所以提示出错信息，继续输入数据。continue 语句在此处的作用就是结束本次循环，直接进入下一次循环。

在混合使用字符和整数作为常量表达式的时候，更要注意常量表达式的值不能重复出现，如：

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

void main(void)
{
   char ch;
   printf(" 请输入需要测试的大写字符 :");
   ch=getch();
   putchar(ch);
   switch(ch){
           case            65:
           case            69:
           case            'I':
           case            'O':
           case            'U':
                           printf("\n 该字符为元音字符 !\n");
                           break;
           default:
                           printf("\n 该字符非元音字符 !\n");
   }

   return ;
}
```

运行结果：

```
请输入需要测试的大写字符 :A
该字符为元音字符！
```

在这里把前面代码中的常量表达式 'A' 和 'B' 分别改写成了它们的 ASCII 码 65 和 66，其功能与前面使用 'A' 和 'B' 是等价的，因此在使用字符型常量的时候要留意其 ASCII 码值不能重复出现在常量表达式中。如果在 "case 65:" 前面加上一句 "case A:"，那么在编译的时候会出现“error C2196: case value '65' already used”错误提示信息。

3. switch 语句的嵌套

通过前面对 switch 语句的介绍，我们发现 switch 语句与 if 语句非常相似。if 语句可以实现嵌套，switch 语句同样可以实现嵌套功能，接下来我们就来看看 switch 语句的嵌套。switch 语句嵌套的一般形式为：

```
switch    （表达式 1）{
   case           常量表达式 1：
       switch     （表达式 2）{
                  case           常量表达式 21：
                      语句段 1；
                      [break]
                  case           常量表达式 22：
                      语句段 2；
                      [break]
                  ……
                  case           常量表达式 2n：
                      语句段 n；
                      [break]
                  default：
                      语句段 2n+1；
                      [break]
}
       [break]
   case           常量表达式 2：
       语句段 2
       [break]
   ……
   case           常量表达式 n：
       语句段 n
       [break]
   default：
       语句段 n+1
       [break]
}
```

下面通过一段代码来介绍 switch 语句嵌套的使用，这段代码的功能是显示班级总分前三名同学的各科成绩，科目包括数学（m）、英语（e）、语文（c）。

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

void main(void)
{
   char    ch;
   int     num;
   printf(" 请输入学生的名次 num（取值为 1、2、3）:");
   num = getch();
   putchar(num);
   printf("\n 请输入所要查询的科目前面 ch（取值为 m（数学）、e（英语）、c（语文））:");
   ch = getch();
   putchar(ch);
   printf("\n");
```

```
    switch(num){
            case    '1':
                    switch(ch){
                            case    'm':
                                    printf("数学成绩为：98\n");
                                    break;
                            case    'e':
                                    printf("英语成绩为：97\n");
                                    break;
                            case    'c':
                                    printf("语文成绩为：95\n");
                                    break;
                            default:
                                    printf("输入出错！\n");
                                    return ;
                    }
                    break;
            case    '2':
                    switch(ch){
                            case    'm':
                                    printf("数学成绩为：98\n");
                                    break;
                            case    'e'     :
                                    printf("英语成绩为：89\n");
                                    break;
                            case    'c':
                                    printf("语文成绩为：87\n");
                                    break;
                            default:
                                    printf("输入出错！\n");
                                    return ;
                    }
                    break;
            case    '3':
                    switch(ch){
                            case    'm':
                                    printf("数学成绩为：98\n");
                                    break;
                            case    'e':
                                    printf("英语成绩为：78\n");
                                    break;
                            case    'c':
                                    printf("语文成绩为：75\n");
                                    break;
                            default:
                                    printf("输入出错！\n");
                                    return ;
                    }
                    break;
            default:
                    printf("输入出错！\n");
    }

    return ;
}
```

运行结果：

```
请输入学生的名次 num(取值为 1、2、3):2
请输入所要查询的科目前面 ch(取值为 m(数学)、e(英语)、c(语文)):e
英语成绩为：89
```

分析上面的代码可知，在使用 switch 语句嵌套的过程中须注意 break 语句的使用。在 switch 语句的嵌套使用中，break 语句仅仅是终止当前的 switch 语句，而不是完全退出整个多重 switch 语句。

3.4　goto 语句的使用及注意事项

goto 语句也称为无条件转移语句。值得注意的是，goto 语句只能在函数内部进行转移，不能够跨越函数。goto 语句使用的一般格式为：

```
goto        语句标号；
……
语句标号：
```

或者

```
语句标号：
……
goto        语句标号；
```

其中，语句标号是 goto 语句转向的目标。注意，目标处的语句标号后有“：”；语句标号的命名要遵循 C 语言的命名规则。下面先来看看如何使用 goto 语句建立循环，下面的代码实现了 1 到 100 之间所有整数的累加和。

```
#include <stdio.h>

void main(void)
{
   int num,i;
   int sum;
   num=0;
   sum=0;

loop:
   sum+=num;
   num++;
   if(num<101)
           goto loop;
   printf(" 使用 goto 语句建立循环求得的 sum=%d\n",sum);

   sum=0;
   for(i=0;i<101;i++)
   {
           sum+=i;
   }
```

```
    printf(" 使用 for 循环求得的 sum=%d\n",sum);

    return ;
}
```

运行结果：

```
使用 goto 语句建立循环求得的 sum=5050
使用 for 循环求得的 sum=5050
```

从运行结果来看，goto 语句建立的循环和 for 循环计算出来的结果完全相同。goto 语句不仅可以实现循环体的设计，还可以用于多层循环体的退出。接下来查找一个四位数中的最小的水仙花数，其代码为：

```
#include <stdio.h>

void main(void)
{
    int i,j,k,f;
    int num;

    for(i=1;i<=9;i++)
            for(j=0;j<=9;j++)
                    for(k=0;k<=9;k++)
                            for(f=0;f<=9;f++)
                            {
                                num=i*1000+j*100+k*10+f;
                                if(num==(i*i*i*i+j*j*j*j+k*k*k*k+f*f*f*f))
                                {
                                    printf("%d\n",num);
                                    goto exit;
                                }
                            }
exit:
    return ;
}
```

运行结果：

```
1634
```

在上面的四重循环中，使用 goto 语句跳出循环体使代码显得很简洁，如果不使用 goto 语句来跳出循环体，那么就要在每个 for 循环中使用满足条件时的退出语句。但是，千万不能在编程中随便使用 goto 语句，因为 goto 语句会使代码的可读性大大降低和难以控制，而且稍有不慎还可能导致程序崩溃。下面一段代码的功能为创建一个动态数组，数组元素的个数由 n 来确定，对数组中的每个元素赋初值，然后输入一个数 n1，打印出数组中前 n1 个元素。

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

void main(void)
{
```

```
    int     n,n1,i;

    printf(" 请输入所要分配的大小 :");
    scanf("%d",&n);
    int* arr=(int *)malloc(sizeof(int)*n);
    int* p;
    for(i=0;i<n;i++)
    {
            arr[i]=i+1;
    }

    printf(" 请输入所要打印的数字元素个数 :");
    scanf("%d",&n1);

    if(n1>n)
            goto    exit;
    p=arr;
    for(i=0;i<n1;i++)
            printf("p[%d]=%d\t",i,p[i]);
    int a;

exit:
    free(p);

    return ;
}
```

如果输入的所要打印的数组元素的个数大于所分配的大小，那么就使用 goto 语句跳过打印部分。下面分析一下上述代码的运行是否有问题。

当输入的打印数组元素个数不大于分配的个数时，运行结果如下：

```
请输入所要分配的大小 :5
请输入所要打印的数字元素个数 :5
p[0]=1  p[1]=2  p[2]=3  p[3]=4  p[4]=5
```

当输入的所要打印的数组元素个数大于分配的个数时，我们发现程序直接崩溃了。为什么会出现这样的情况呢？这是因为使用 goto 语句跳过了对整型指针 p 赋值的语句“p=arr;”。由于 p 是一个空指针，不允许使用 free() 函数，否则会导致程序崩溃。

因此，读者在编程时要慎用 goto 语句，不到迫不得已不要使用 goto 语句，如果用了 goto 语句，要在其旁边写出详细的注释。

3.5　for 语句的使用及注意事项

C 语言中的 for 语句使用极为灵活，不仅可以用于循环次数已经确定的情况，还可以用于循环次数不确定而只给出循环结束条件的情况，它完全可以代替 while 语句。

1. 简单 for 语句

在前面的代码中已经多次用到 for 循环，所以读者对 for 循环语句应该并不陌生。在讲

解 for 循环语句之前，还是先来看看 for 循环语句的构成。for 循环由 4 个部分所组成，即 3 个表达式和 1 个循环体，其一般形式为：

```
for( 表达式 1; 表达式 2; 表达式 3)
     循环体 ;
```

值得注意的是，循环体中的表达式之间用“；”分隔开。表达式 1 通常用来为循环变量赋初始值；表达式 2 作为循环控制表达式，也就是循环条件；表达式 3 通常用来改变循环变量的值。for 循环语句流程图如图 3-8 所示。

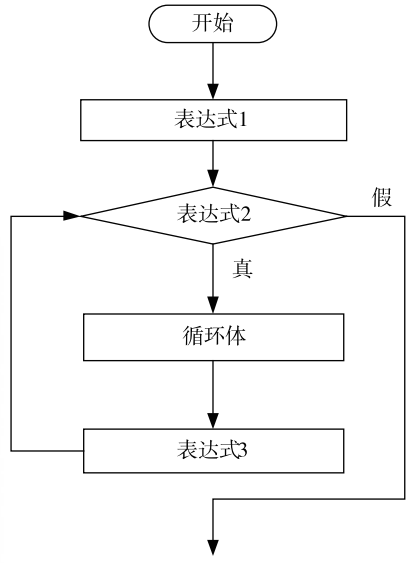


图 3-8　for 循环语句流程图

for 循环的执行大致可分为 4 个步骤：

（1）执行表达式 1，只执行一次。

（2）计算表达式 2 的值，看其否为真（非零），如果为真，那么就执行循环体部分，否则直接退出，执行 for 循环下面的语句。

（3）执行循环体。

（4）计算表达式 3 的值，然后返回步骤 2。

for 循环语句的表达式 1 通常用来对循环变量做初始化操作，在 for 循环的执行流程中仅被执行一次。如果在 for 循环语句之前已经做了初始化处理，那么表达式 1 可以为空。表达式 2 提供一个循环体能够执行的条件，如果表达式 2 为空，那么该 for 循环就是死循环，这时可以在循环体中采用 break 语句来退出循环。表达式 3 通常用来改变循环变量，表达式 3 也可以为空并放在循环体中，但通常不建议这样做，因为这样会使 for 循环的可读性变差。值得注意的是，不管 for 循环中的 3 个表达式是否为空，其中的两个“;”一个都不能少。看看下面一段代码。

```
#include <stdio.h>

void main(void)
{
   int     i;
   char a[20]="Hello World!";
   char b[20];
   i=0;
   for(;b[i];)
   {
           b[i]=a[i];
           i++;
   }
   printf("%s\n",b);

   return ;
}
```

运行结果：

```
Hello World!
```

在上面的代码中，在 for 循环中没有使用表达式 1 和表达式 3。在循环体外对 i 进行了初始化，用表达式 b[i] 作为循环的控制表达式，当“b[i]='\0'”的时候循环结束；将表达式 3 放到了循环体中来进行，所以为空。虽然表达式 1 和表达式 3 为空，但是表达式之间的“；”依然存在。

2. for 语句的嵌套

for 语句的嵌套，就是在一个 for 循环中包含另外一个 for 循环结构。值得注意的是，内层 for 循环被当成外层 for 循环的循环体的一部分在执行。for 循环嵌套的一般形式为：

```
for( 表达式 11；表达式 12；表达式 13)
{
    for( 表达式 21；表达式 22；表达式 23)
    {
        for( 表达式 31；表达式 32；表达式 33)
        {
            循环体 ;
        }
    }
}
```

接下来看一个 for 循环嵌套的示例。以下代码以下三角的方式打印出九九乘法表。

```
#include <stdio.h>

void main(void)
{
   int     i,j;

   for(i=1;i<10;i++)
   {
          for(j=1;j<=i;j++)
                 printf("%d×%d=%d\t",j,i,i*j);
          printf("\n");
   }

   return ;
}
```

运行结果：

```
1×1=1
1×2=2   2×2=4
1×3=3   2×3=6   3×3=9
1×4=4   2×4=8   3×4=12  4×4=16
1×5=5   2×5=10  3×5=15  4×5=20  5×5=25
1×6=6   2×6=12  3×6=18  4×6=24  5×6=30  6×6=36
1×7=7   2×7=14  3×7=21  4×7=28  5×7=35  6×7=42  7×7=49
1×8=8   2×8=16  3×8=24  4×8=32  5×8=40  6×8=48  7×8=56  8×8=64
1×9=9   2×9=18  3×9=27  4×9=36  5×9=45  6×9=54  7×9=63  8×9=72  9×9=81
```

下面分析一下 for 嵌套循环的特点。当外层循环不影响内层循环的执行次数时，内层循环体执行的次数等于一个完整的内层循环执行次数乘以外层执行次数；如果外层循环对内层循环执行的次数有影响，如上面的代码，那么内层循环执行的次数等于内层循环执行次数的叠加。

3. break 语句在 for 循环中的使用

在 for 循环中使用 break 语句的一般形式为：

```
for( 表达式 1; 表达式 2; 表达式 3)
{
    循环体 1;
    break;
    循环体 2;
}
```

通常都是在 for 循环的循环体中通过 if 语句与 break 搭配使用来实现在满足一定条件时退出整个 for 循环，其相应的流程图如图 3-9 所示。

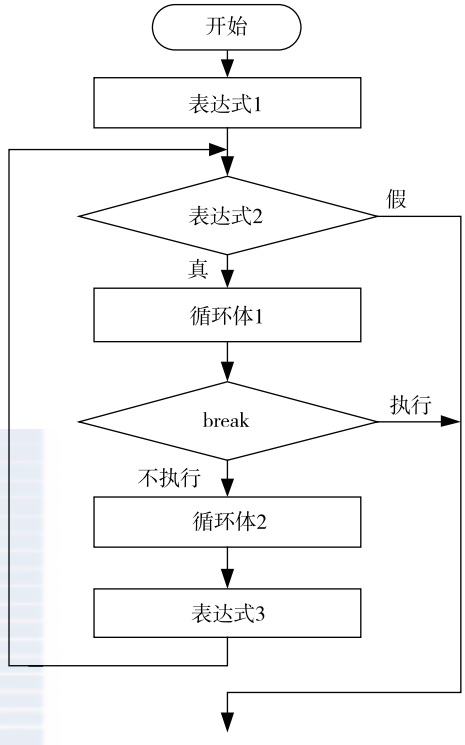


图 3-9　含有 break 语句的 for 循环流程图

下面这一段代码的功能为查找一个三位数的最大水仙花数。

```
#include <stdio.h>

void main(void)
{
    int     i,j,k;
    int num;
    int flag = 0;

    for(i=9;i>=1;i--)
    {
            for(j=9;j>=0;j--)
            {
                    for(k=9;k>=0;k--)
                    {       num = i*100+j*10+k;
                            if(i*i*i+j*j*j+k*k*k == num )
                            {
                                    flag = 1 ;
                                    printf(" 三位数中的最大水仙花数为 :%d\n",num);
                                    break;
                            }
                    }
                    if(1==flag)
                            break;
            }
            if(1==flag)
                    break;
    }
```

```
    return ;
}
```

运行结果：

```
三位数中的最大水仙花数为 :407
```

与前面使用 goto 语句退出多重循环相比，这里使用了一个信号标记，当满足条件的时候，采用逐层退出的方式从内到外依次退出循环体，虽然每次都要判断信号标记是否满足条件，但是相比 goto 语句，break 语句有更好的可读性和易控制性，所以建议使用 break 语句来退出循环。

4. continue 语句在 for 循环中的使用

接下来看看 continue 语句在 for 循环中的使用。continue 语句的作用是结束本次循环，其后的循环体将不会被执行，跳转至下一次循环。可以看出，continue 语句与 break 语句的区别在于：执行 break 语句退出其所在的 for 循环，而执行 continue 语句只结束本次循环，跳转至下一次循环。含有 continue 语句的 for 循环流程图如图 3-10 所示。

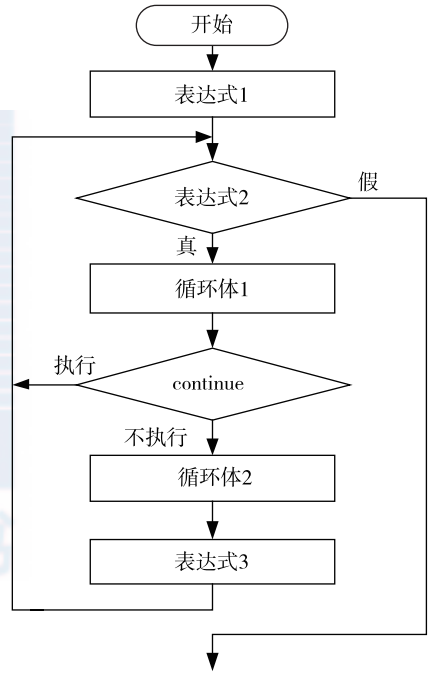


图 3-10　含有 continue 语句的 for 循环流程图

下面这段代码的功能为求 1 到 100 之间不能被 5 整除的整数之和。

```
#include <stdio.h>

void main(void)
{
    int     i;
    int sum;
    sum=0;

    for(i=1;i<101;i++)
    {
            if(0==i%5)
                    continue;
            sum+=i;
    }
    printf("1 到 100 之间不能被 5 所整除的整数之和为 :%d\n",sum);

    return ;
}
```

运行结果：

```
1 到 100 之间不能被 5 所整除的整数之和为 :4000
```

分析上面的代码，如果数 i 能被 5 所整除，那么就执行 continue 语句，跳过下面的“sum+=i；”语句部分，舍弃对不满足条件数据的求和。注意，continue 不能用于 goto 语句构

成的循环语句。

3.6 while 循环与 do while 循环的使用及区别

1. while 循环

while 循环的一般形式为：

```
while( 表达式 )
循环体 ;
```

while 循环的执行流程是先计算表达式的值，如果为真，那么就执行循环体，否则退出循环。while 循环流程图如图 3-11 所示。

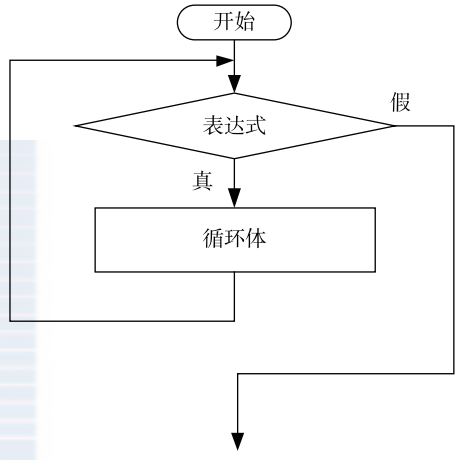


图 3-11 while 循环流程图

下面的代码通过 while 循环来实现 1 到 n 之间的整数之和，n 通过输入来确定。

```
#include <stdio.h>

void main(void)
{
   int     n;
   printf(" 请输入 n 值 :");
   scanf("%d",&n);
   int sum;
   sum=n;

   printf("1 到 %d 之间的整数之和为 :",n);
   while(n--)
   {
           sum+=n;
   }
   printf("%d\n",sum);

   return ;
}
```

运行结果：

```
请输入 n 值 :5
1 到 5 之间的整数之和为 :15
```

在上面的程序中，用“n--”作为 while 循环的表达式，当“n--”的值为 0 时结束循环。当 while 循环的表达式始终为非零时，表达式的值始终为真，这时 while 循环成为了死循环，可以使用前面讲过的 break 语句来结束循环。在 while 循环中使用 break 语句的一般形式为：

```
while（表达式）
{
```

```
循环体 1;
break;
循环体 2;
}
```

含有 break 语句的 while 循环的流程图如图 3-12 所示。

下面通过一段代码来看看 break 语句在 while 循环中的使用。以下代码的功能为求一个数 n 的阶乘，其中，n 由键盘输入，要求 n 的值不大于 20。

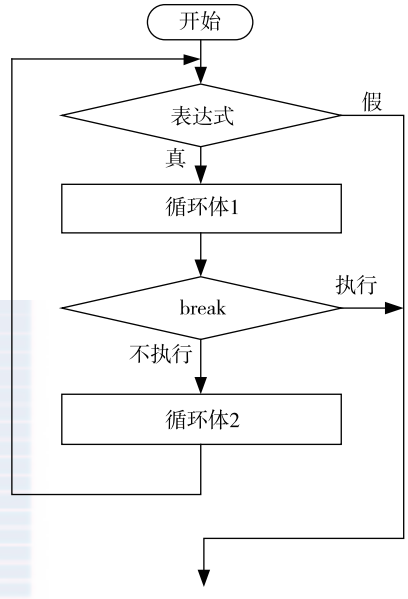


图 3-12　含有 break 语句的 while 循环流程图

```
#include <stdio.h>

void main(void)
{
   int     n;
   printf(" 请输入 n 值 :");
   scanf("%d",&n);
   if(n>20 || n<0)
   {
          printf(" 输入出错 !\n");
          return ;
   }
   int num;
   num=1;
   printf("%d 的阶乘为 :",n);
   while(1)
   {
          if(n<0)
                 break;
          if(0==n)
                 num*=1;
          else
                 num*=n;
          n--;
   }
   printf("%d\n",num);

   return ;
}
```

运行结果：

```
请输入 n 值 :6
6 的阶乘为 :720
```

在上面的代码中，在 while 循环的表达式中使用了一个非零常量 1，所以这个 while 循环是一个死循环，但是在 while 循环体内通过一个 if 语句来判断当前的 n 值，进而决定是否执行 break 语句来退出循环体。当 n 的值为负时，执行 break 语句，退出 while 循环。

在 while 循环中也可以使用 continue 语句来结束循环，其相应的流程图如图 3-13 所示。

continue 语句在 while 循环中的作用是结束本次循环体的执行，不再执行其后的循环体，跳转到表达式处开始新一轮的循环判断。下面通过一段代码来了解 continue 语句在 while 循环体中的使用。此段代码的功能为打印出 1 到 30 之间所有 3 的倍数的整数。

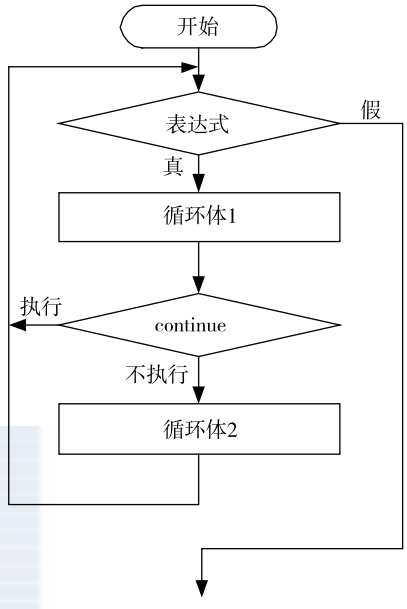


图 3-13　含有 continue 语句的 while 循环流程图

```
#include <stdio.h>

void main(void)
{
   int     n;
   int num;
   n=1;
   num=0;

   while(n<31)
   {

           if(0 != n%3)
           {
                   n++;
                   continue;
           }
           printf("%d\t",n);
           num++;
           if(0 == num%5)
                   printf("\n");
           n++;
   }

   return ;
}
```

运行结果：

```
3       6       9       12      15
18      21      24      27      30
```

在上面的代码中，通过 if 语句来判断当前的 n 是否是 3 的倍数来决定是否执行 continue 语句，如果不是 3 的倍数，那么就执行 continue 语句结束本次循环的执行，其后的循环体不会被执行，转而执行表达式看是否满足本循环体的执行条件，否则就执行接下来的循环体，打印出当前的数据 n。

2. do-while 循环

do-while 循环的一般形式为：

```
do{
   循环体 ;
}while( 表达式 );
```

do-while 循环流程图如图 3-14 所示。

从图 3-14 可以看出，do-while 循环的执行流程与 while 循环的最大区别是：do-while 循环先执行循环体，再判断表达式的值是否为真，如果为真，那么继续执行循环体，否则退出循环，无论在什么情况下，do while 循环体都至少执行一次；而对于 while 循环，如果起始条件不满足，那么循环体一次都不执行。接下来通过下面的代码来看 do-while 循环的使用。代码的功能为求 1 到 n 之间所有正整数的平方和，n 由输入确定。

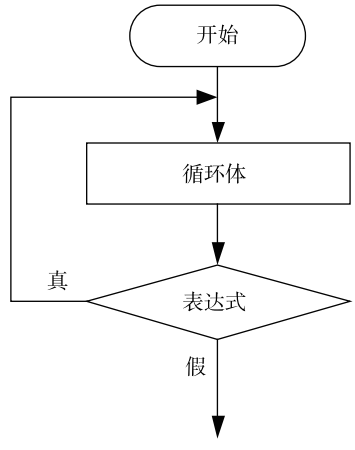


图 3-14　do-while 循环流程图

```
#include <stdio.h>

void main(void)
{
    int     n;
    int sum;
    printf(" 请输入 n:");
    scanf("%d",&n);
    printf("1 到 %d 之间所有正整数的平方和为 :",n);

    sum=0;
    do{
            sum+=n*n;
    }while(--n);
    printf("%d\n",sum);

    return ;
}
```

运行结果：

```
请输入 n:8
1 到 8 之间所有正整数的平方和为 :204
```

像 while 循环一样，也可以使用 break 语句来退出 do-while 循环，其使用的一般形式为：

```
do{
    循环体 1;
    break;
    循环体 2;
}while( 表达式 );
```

其相应的流程图如图 3-15 所示。

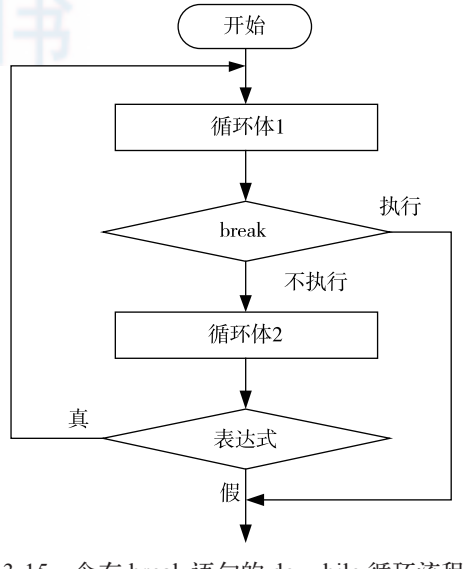


图 3-15　含有 break 语句的 do-while 循环流程图

如果在循环体中执行了 break 语句，那么就直接退出 do-while 循环。接下来看看 break 在 do-while 循环中的使用，以下代码的功能为查找 100 以内能同时被 2、5、9 整除的最大正

整数。

```
#include <stdio.h>

void main(void)
{
   int     n;
   n=100;

   do{
          if(0 == n%2 && 0 == n%5 && 0 == n%9)
                 break;
   }while(n--);
   printf("100 以内能同时被 2、5、9 整数的最大正整数为 :%d\n",n);

   return ;
}
```

运行结果：

```
100 以内能同时被 2、5、9 整数的最大正整数为 :90
```

看完 break 语句在 do-while 循环中的使用，接下来看 continue 语句在 do-while 循环中的使用，其一般形式为：

```
do{
   循环体 1;
   continue;
   循环体 2;
}while( 表达式 );
```

相应的流程图如图 3-16 所示。

从流程图中可以看出，如果在 do-while 循环体中执行了 continue 语句，那么接下来就跳转到表达式执行，不少人不能够正确地画出 do-while 循环使用 continue 语句的流程图，认为执行 continue 语句之后跟前面讲解的 while 循环一样跳转到了循环上面的开始处，这种理解是错误的。下面的代码说明执行了 continue 语句之后跳转到表达式处，而不是直接重新开始执行循环体。

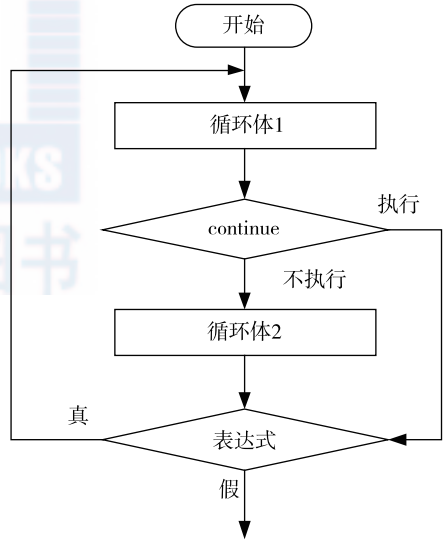


图 3-16　含有 continue 语句的 do-while 循环流程图

```
#include <stdio.h>

void main(void)
{
   int     n;
   n=0;
```

```
    do{
            printf(" 执行 continue 语句之前的打印语句 !\n");
            if(!n)
                    continue;
            printf(" 如果执行 continue 那么该语句不被打印 \n");
    }while(n);
    printf(" 测试成功，跳转到表达式处执行 !\n");

    return ;
}
```

运行结果：

```
执行 continue 语句之前的打印语句 !
测试成功，跳转到表达式处执行 !
```

分析这里的代码，这里给定 n 的初始值为 0，如果执行了 continue 语句之后跳转到循环体的开始处，那么每次都不会执行表达式，这个 do-while 循环会变成死循环。但是测试结果表明这个假设不成立，因为打印语句成功地说明了执行完 continue 语句之后，跳转到表达式处执行。接下来看看 continue 语句在 do-while 循环中的使用。下面这段代码的功能为查找 50 以内能同时被 2、5 整除的正整数。

```
#include <stdio.h>

void main(void)
{
    int     n;
    n=50;

    do{
            if(0!=n%2)
                    continue;
            if(0!=n%5)
                    continue;
            printf(" 能同时被 2 和 5 整除的正整数 :%d\n",n);
    }while(--n);

    return ;
}
```

运行结果：

```
能同时被 2 和 5 整除的正整数 :50
能同时被 2 和 5 整除的正整数 :40
能同时被 2 和 5 整除的正整数 :30
能同时被 2 和 5 整除的正整数 :20
能同时被 2 和 5 整除的正整数 :10
```

通过 if 语句来判断数 n 是否能同时被 2 和 5 整除，如果不能被其中一个整除，那么就结束本次循环，跳转到表达式，判断是否满足进入下一次循环的条件。如果能同时被 2 和 5 整除，那么就打印输出该数。

3.7 循环结构中 break、continue、goto、return 和 exit 的区别

在此之前讲解循环结构时不止一次提到了 break 语句和 continue 语句的使用，接下来看看 break、continue、goto、return 和 exit 在循环结构中的区别和注意事项。

1. break

break 语句的使用场合主要是 switch 语句和循环结构。在循环结构中使用 break 语句，如果执行了 break 语句，那么就退出循环，接着执行循环结构下面的第一条语句。如果在多重嵌套循环中使用 break 语句，当执行 break 语句的时候，退出的是它所在的循环结构，对外层循环没有任何影响。如果循环结构里有 switch 语句，并且在 switch 语句中使用了 break 语句，当执行 switch 语句中的 break 语句时，仅退出 switch 语句，不会退出外面的循环结构。通过图 3-17，读者可以很直观地了解 break 语句的使用。

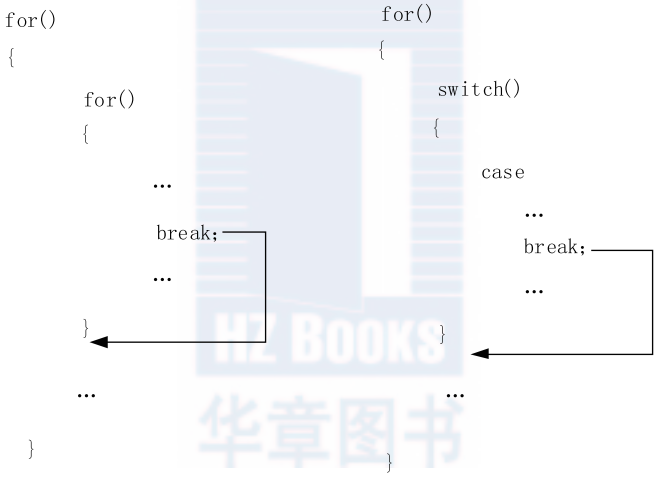


图 3-17 break 语句

2. continue

continue 语句是这 5 种结束循环的方式中最特殊的，因为它并没有真的退出循环，而是只结束本次循环体的执行，所以在使用 continue 的时候要注意这一点。图 3-18 为各种循环结构中 continue 语句的使用。

在 for 循环中，首先执行表达式 1（注意表达式 1 在整个循环中仅执行一次），接着执行表达式 2，如果满足条件，那么执行循环体，如果在循环体中执行了 continue 语句，那么就跳转到表达式 3 处执行，接下进行下一次循环，执行表达式 2，看是否满足条件；在 while 循环中，如果执行了 continue 语句，那么就直接跳转到表达式处，开始下一次的循环判断；在 do while 循环体中如果执行了 continue 语句，那么就跳转到表达式处进行下一次的循环判断，这一点前面已经验证过了。

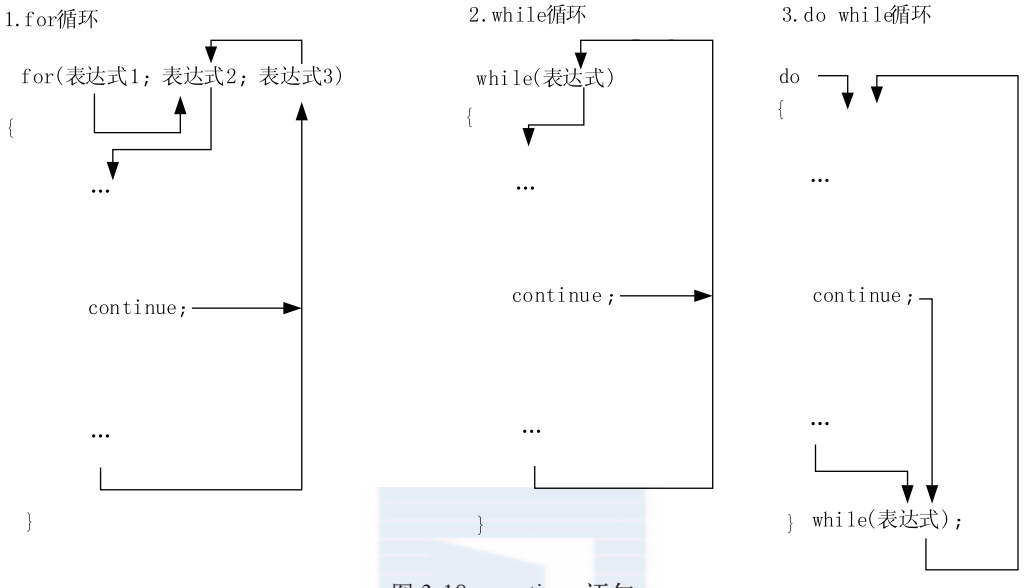


图 3-18　continue 语句

3. goto 语句

在此之前已经讲解了如何使用 goto 语句来退出多重循环，以及使用 goto 语句时的注意事项。图 3-19 中为 goto 语句在各种循环结构中的执行。

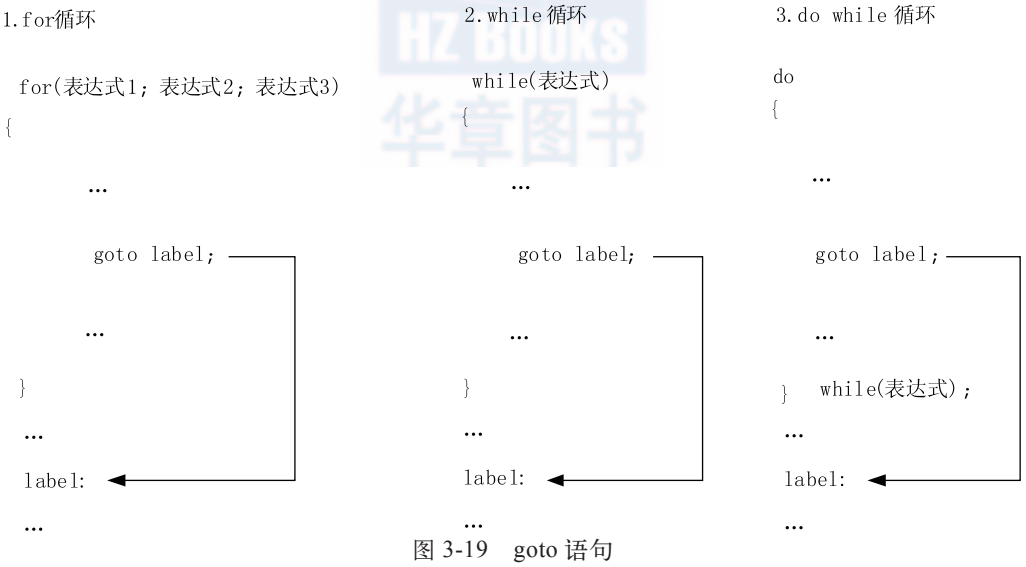


图 3-19　goto 语句

goto 语句可以跳转到标号所在的任何地方继续往下执行，值得注意的是，标号必须与 goto 语句在同一个函数体内，不能跨越函数体。

4. return 语句

如果在程序中遇到 return 语句，那么代码就退出该函数的执行，返回到函数的调用处，如果是 main() 函数，那么结束整个程序的运行。图 3-20 为 return 语句的使用。

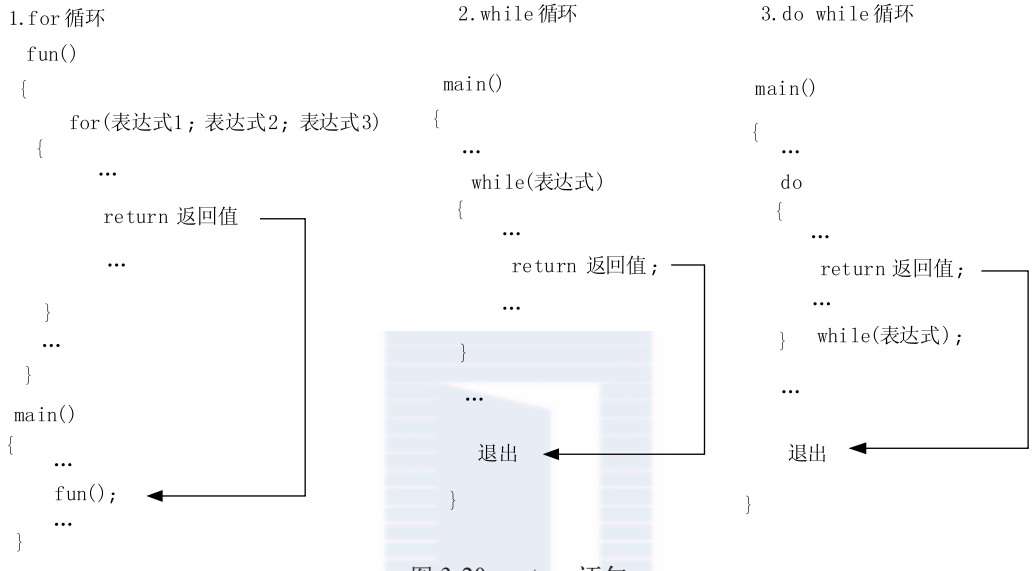


图 3-20　return 语句

如果是在自定义的函数中执行，那么执行 return 之后就返回到函数的调用处继续往下执行。

5. exit() 函数

exit() 函数与 return 语句的最大区别在于，调用 exit() 函数将会结束当前进程，同时删除子进程所占用的内存空间，把返回信息传给父进程。当 exit() 中的参数为 0 时，表示正常退出，其他返回值表示非正常退出，执行 exit() 函数意味着进程结束；而 return 仅表示调用堆栈的返回，其作用是返回函数值，并且退出当前执行的函数体，返回到函数的调用处，在 main() 函数中，return n 和 exit(n) 是等价的。图 3-21 为 exit() 函数的使用。

接下来通过两段代码对 return 语句和 exit() 函数进行简单的对比，先来看 return 语句的使用。

```
#include <stdio.h>
#include <stdlib.h>
int print()
{
    int n;
    n = 0;
    printf(" 使用 return 来结束循环 \n");
    while(1)
    {
```

```
            if(9==n)
                    return n;
            n++;
    }
    return 0;
}

void main(void)
{
    int ret;
    printf(" 调用 print() 函数之前 \n");
    ret = print();
    printf("print() 函数的返回值 ret=%d\n",ret);
    printf(" 调用 print() 函数之后 \n");

    return ;
}
```

运行结果：

```
调用 print() 函数之前
使用 return 来结束循环
print() 函数的返回值 ret=9
调用 print() 函数之后
```

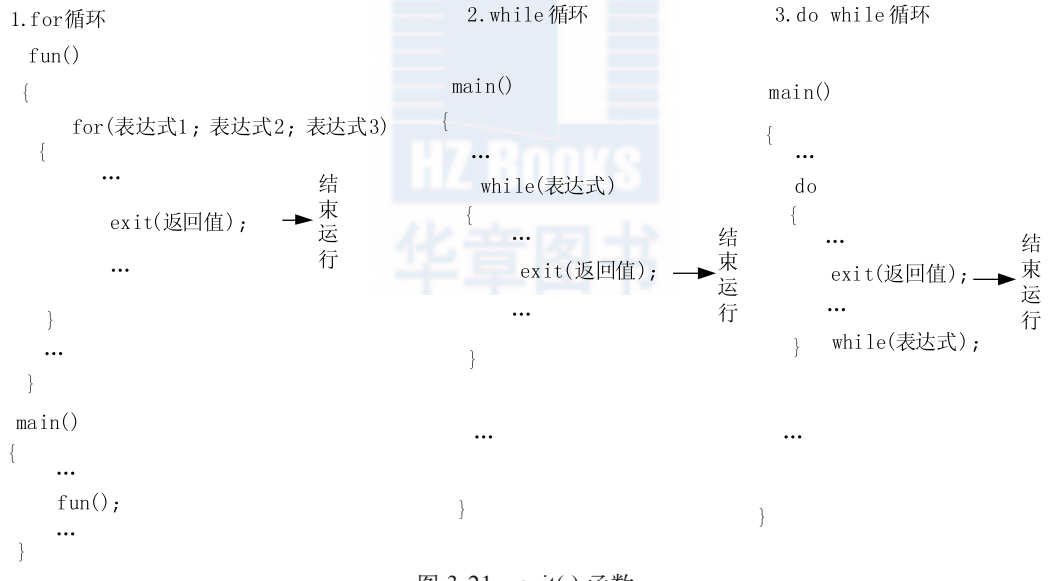


图 3-21　exit( ) 函数

在上面的代码中，用 return 语句来退出 while 死循环，在 main() 函数中 print() 函数的调用处将返回值赋给 ret，打印输出后可以看到使用 return 语句成功地返回了 9。

下面来看 exit() 函数的使用。

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

```
void print()
{
   int n;
   n = 0;
   printf(" 使用 exit 来结束循环 \n");
   while(1)
   {
           if(9==n)
                   exit(1);
           n++;
   }
   return ;
}

void main(void)
{
   int ret;
   printf(" 调用 print() 函数之前 \n");
   print();
   printf(" 调用 print() 函数之后 \n");

   return ;
}
```

运行结果：

```
调用 print() 函数之前
使用 return 来结束循环
```

从以上代码可以看出，如果执行 exit() 函数后能够返回到 main() 函数的调用处，那么可以打印出接下来的信息“调用 print() 函数之后”，但是运行结果表明在调用 exit() 函数之后没有任何输出，所以执行 exit() 函数之后将直接结束整个程序的运行。

51CTO.com 技术成就梦想 联合策划

在编程语言领域，各种新的语言层出不穷，C语言虽已有数十年的历史，但却依然位于编程语言排行榜的榜首，这在编程领域算是独一无二的了。C语言入门相对简单，但是要透彻理解和掌握却不容易，本书对初学者不易理解的难点、疑点和重要知识点进行了解读和剖析，不仅涉及几乎所有的语法知识点，而且包括算法和编码规范方面的话题。对于有一定经验且想进一步提高的C语言开发者而言，本书值得仔细品读，强烈推荐！

—— 51CTO（www.51cto.com）中国领先的IT技术网站

本书主要内容：

- 堆和栈、全局变量和局部变量、生存期和作用域、内部函数和外部函数、指针变量、指针数组和数组指针、指针函数和函数指针、传址和传值、递归和嵌套、结构体和共用体、枚举、位域等较难理解的核心概念的阐述和对比；
- 预处理中的疑难知识点，包括文件的包含方式、宏定义及其常见错误解析、条件编译指令和#pragma指令的使用等；
- if、switch等选择结构语句的使用注意事项和易错点解析；
- for、while、do while等循环结构语句的使用注意事项和易错点解析；
- 循环结构中break、continue、goto、return、exit的区别；
- 一维数组、二维数组、多维数组、字符数组、动态数组的定义和引用，以及操作数组时的各种常见错误解析；
- 不同类型的指针之间的区别，以及指针的一般用法和注意事项；
- 指针与地址、数组、字符串、函数之间的关系，以及指针与指针之间的关系；
- 枚举类型的使用及注意事项，结构体变量和共用体变量的初始化方法及引用；
- 传统链表的实现方法和注意事项，以及对传统链表实现方法的颠覆；
- 与函数参数、变参函数、函数调用、函数指针相关的一些难理解和容易被理解错的知识点解析；
- 文件和指针的使用原则、技巧和注意事项；
- 函数调用和异常处理的注意事项和最佳实践；
- 与strlen、sizeof、const、volatile、void、void*、#define、typedef、realloc、malloc、calloc等相关的一些陷阱知识点的解析；
- 时间复杂度、冒泡排序法、选择排序法、快速排序法、归并排序法、顺序排序法、二分查找等常用算法的详细讲解；
- 良好的编码习惯和编程风格。

上架指导：计算机/程序设计/C语言
ISBN 978-7-111-38861-6
9 787111 388616
定价：59.00元

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