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

ISBN：978-7-111-38861-6

本书纸版由机械工业出版社于2012年出版，电子版由华章分社（北京华章图文信息有限公司）全球范围内制作与发行。

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com

官方网址：www.hzmedia.com.cn

新浪微博 @研发书局

腾讯微博 @yanfabook

目　录

附录　如何养成良好的编程习惯

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

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。

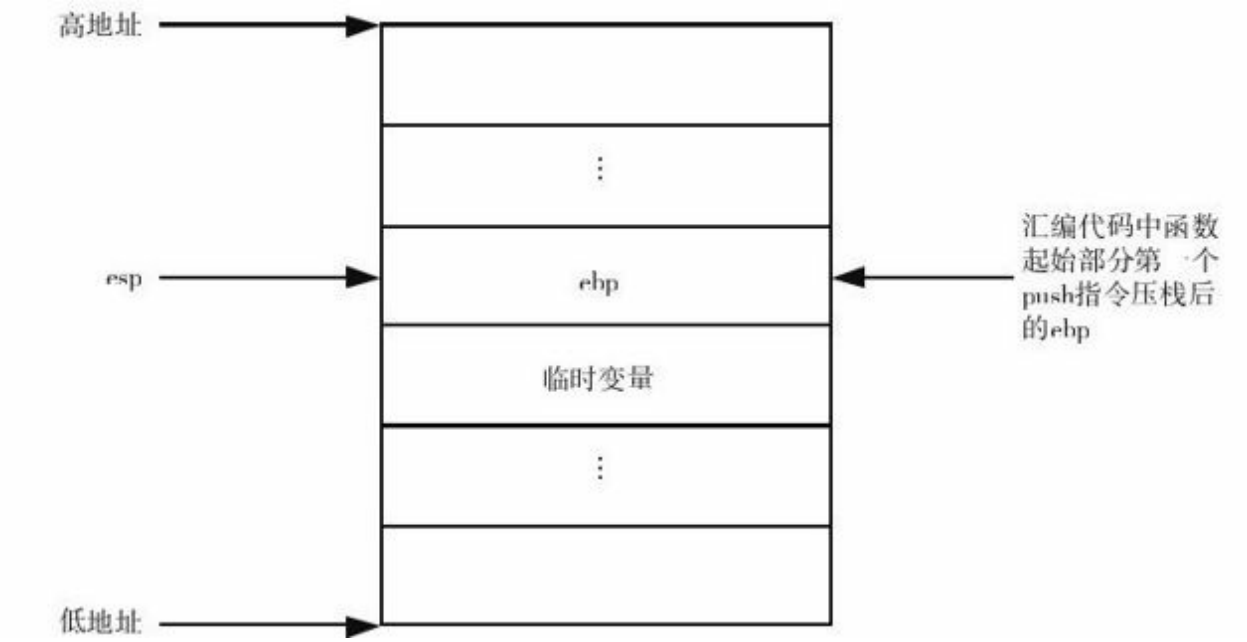

图　1-1　函数调用过程中的压栈流程

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。

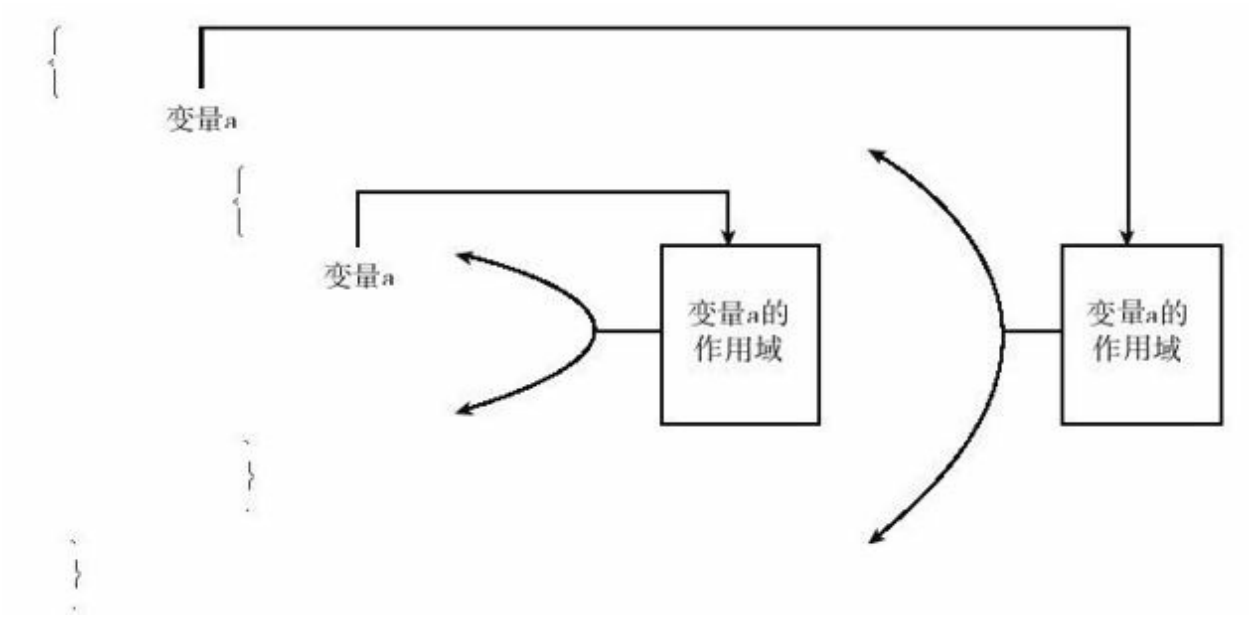

图　1-3　同名变量的不同作用域

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。

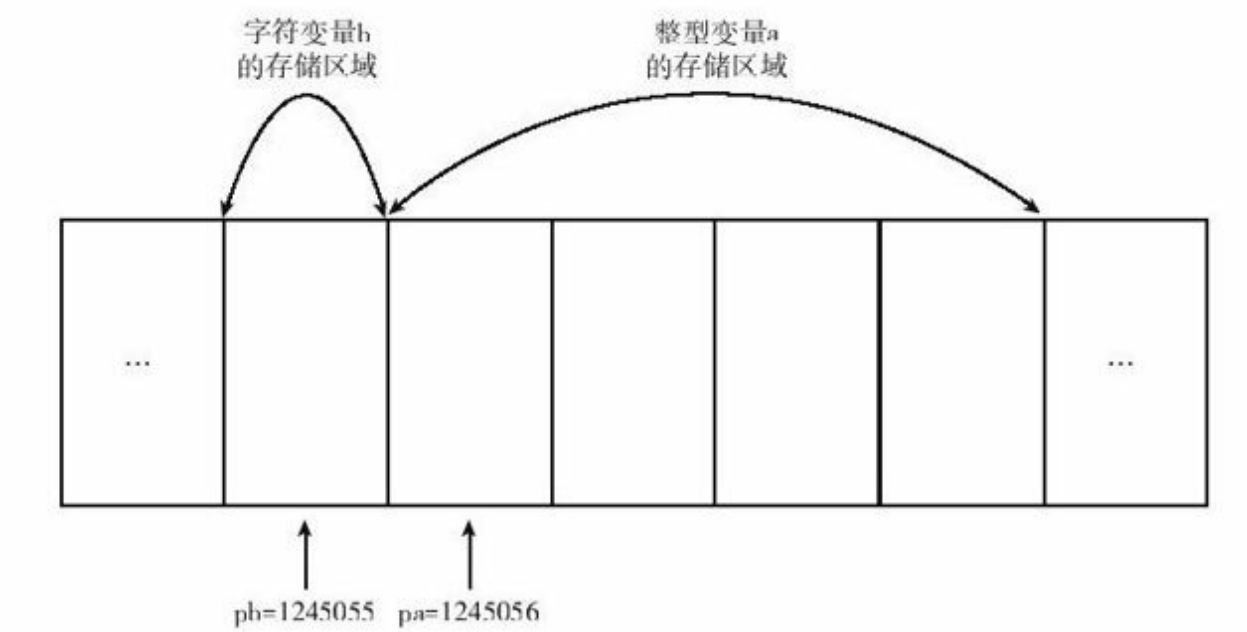

图　1-4　不同类型的指针变量在内存中的存储区域分配

```
#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。

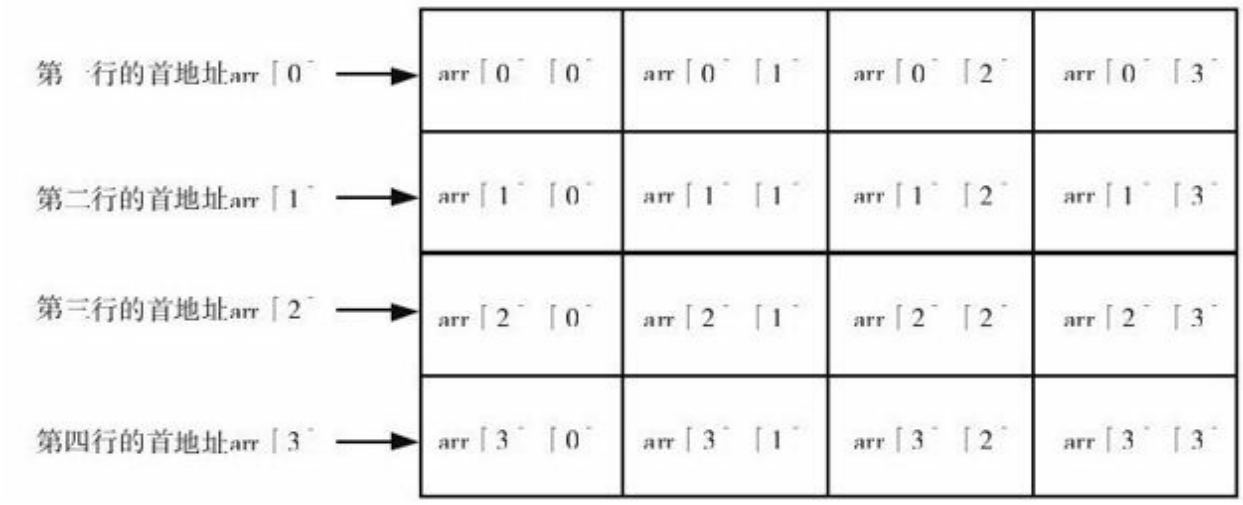

图　1-5　二维数组

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。

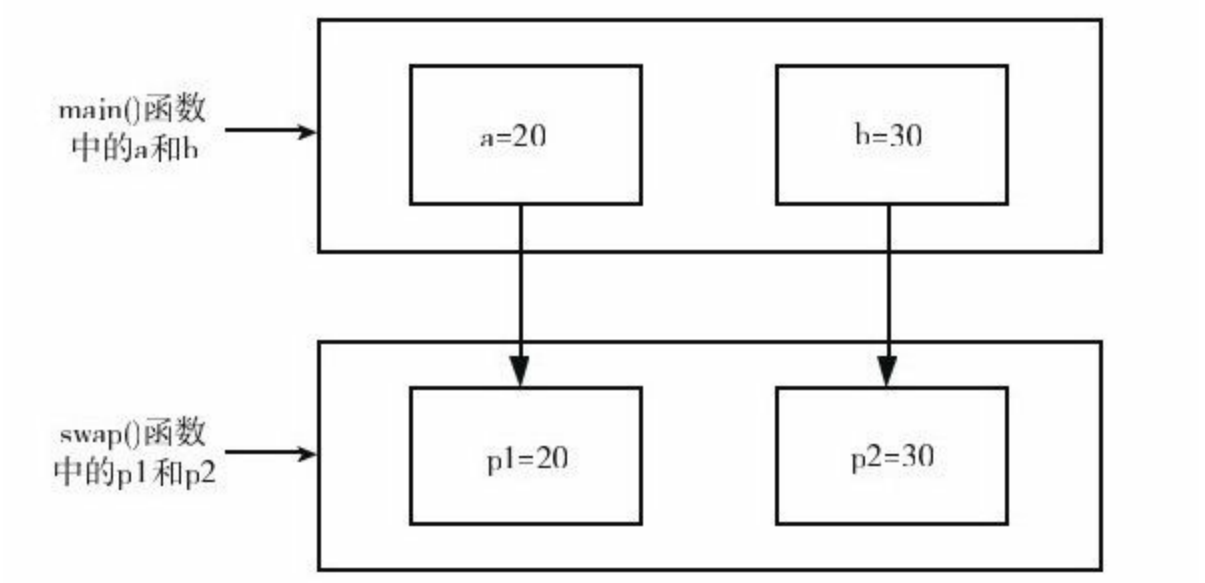

图　1-6　传值

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。

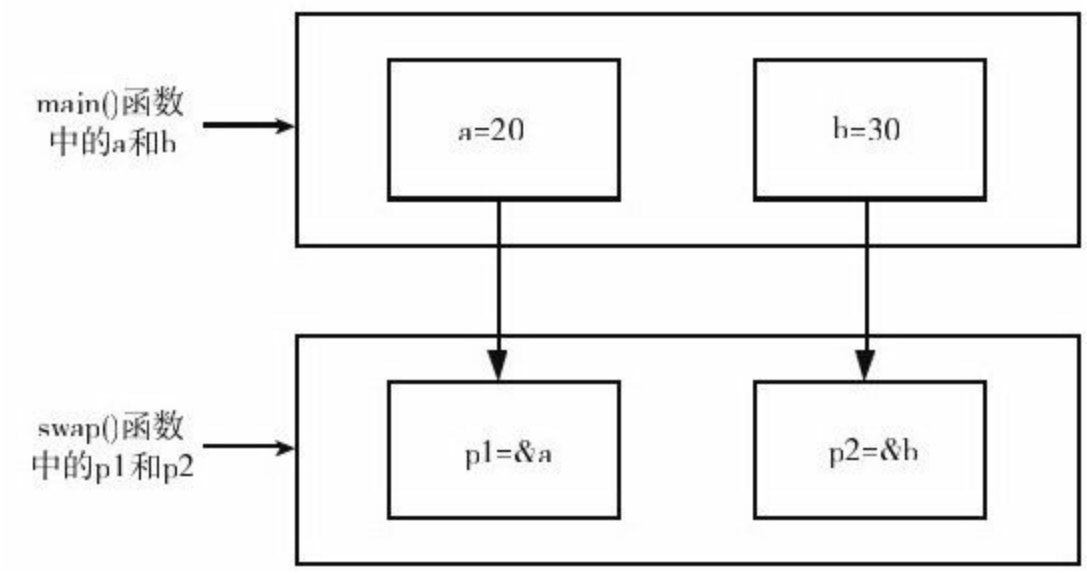

图　1-7　传址

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。

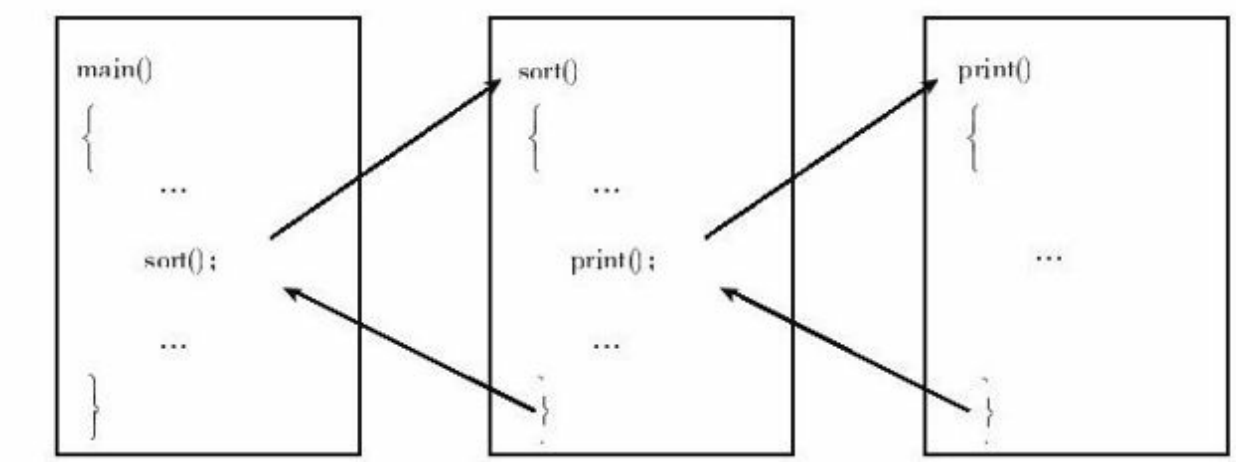

图　1-8　函数的嵌套调用

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。

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

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。

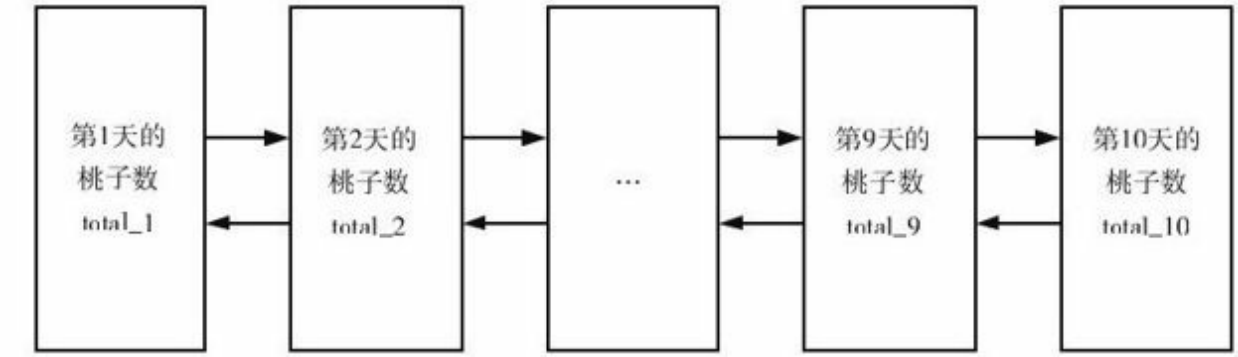

图　1-9　函数的递归调用

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。

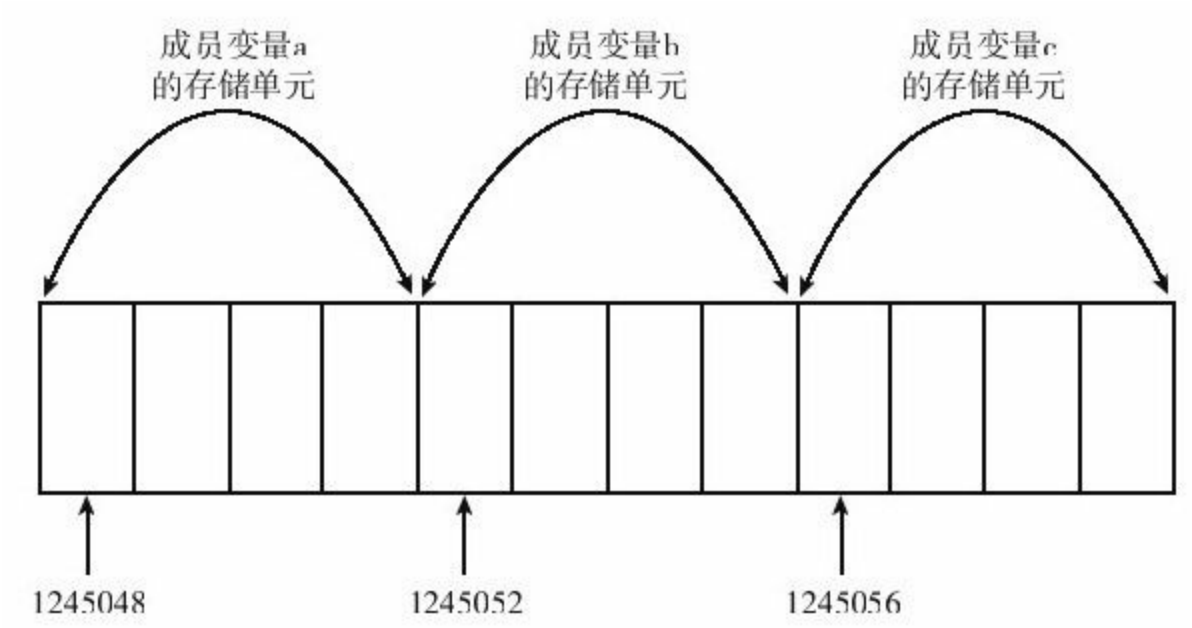

图　1-10　结构体x的内存结构

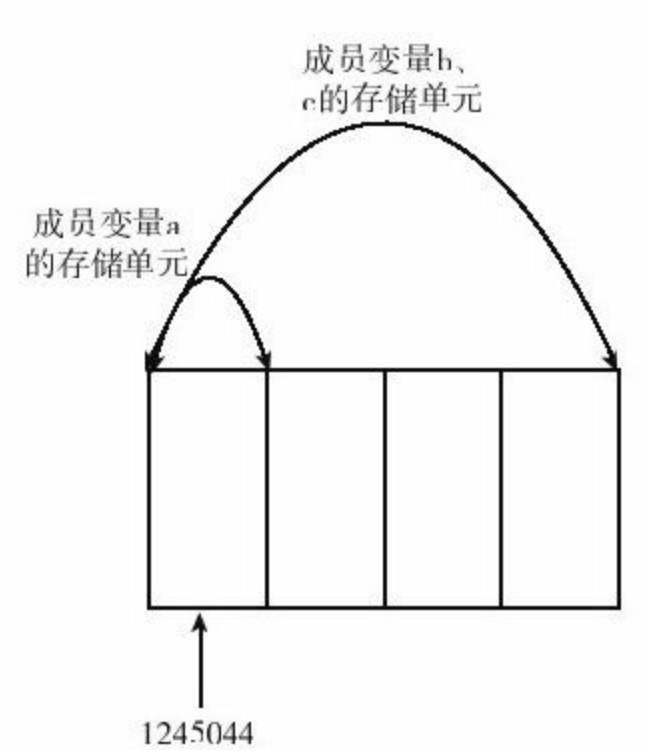

图　1-11　共用体y的内存结构

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）；
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。

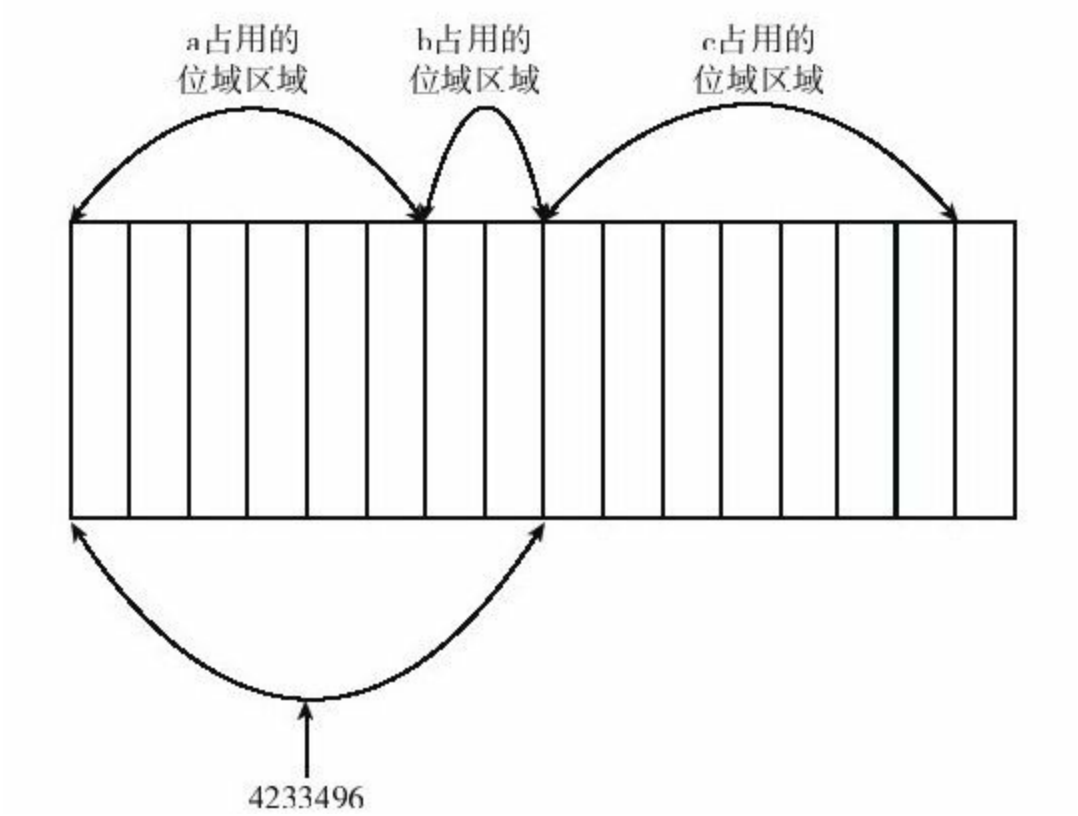

图　1-12　位域变量data的内存结构（1）

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。

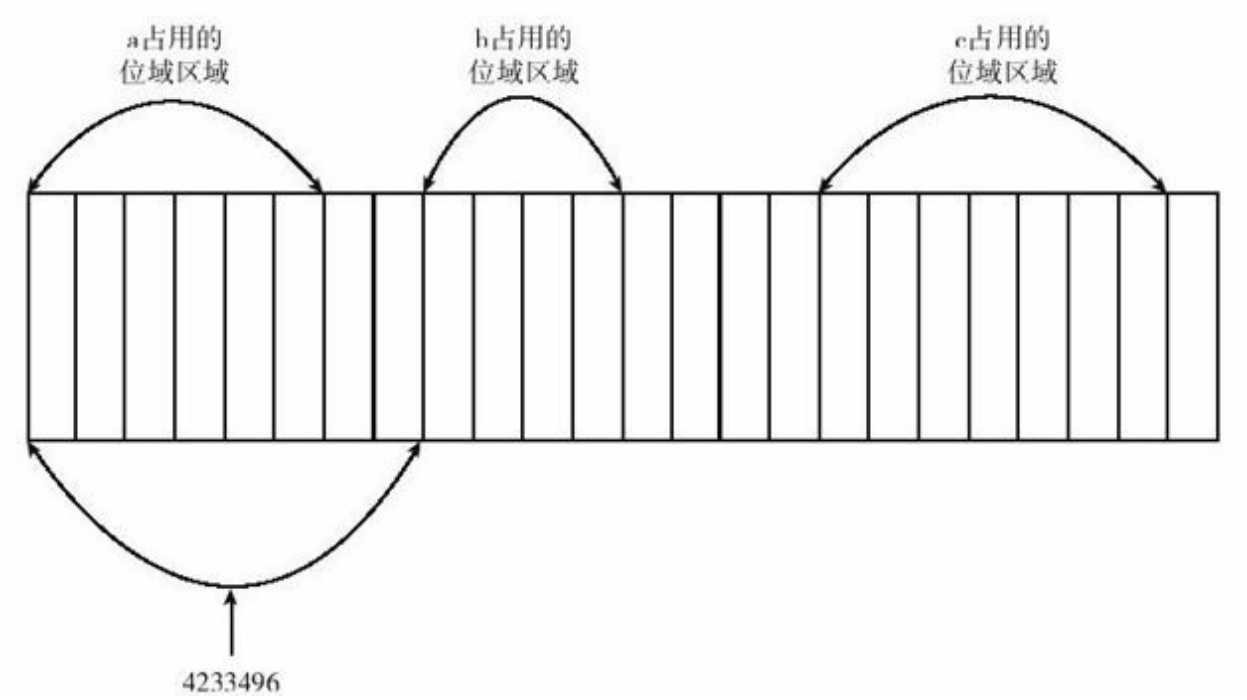

图 1-13 位域变量data的内存结构（2）

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。

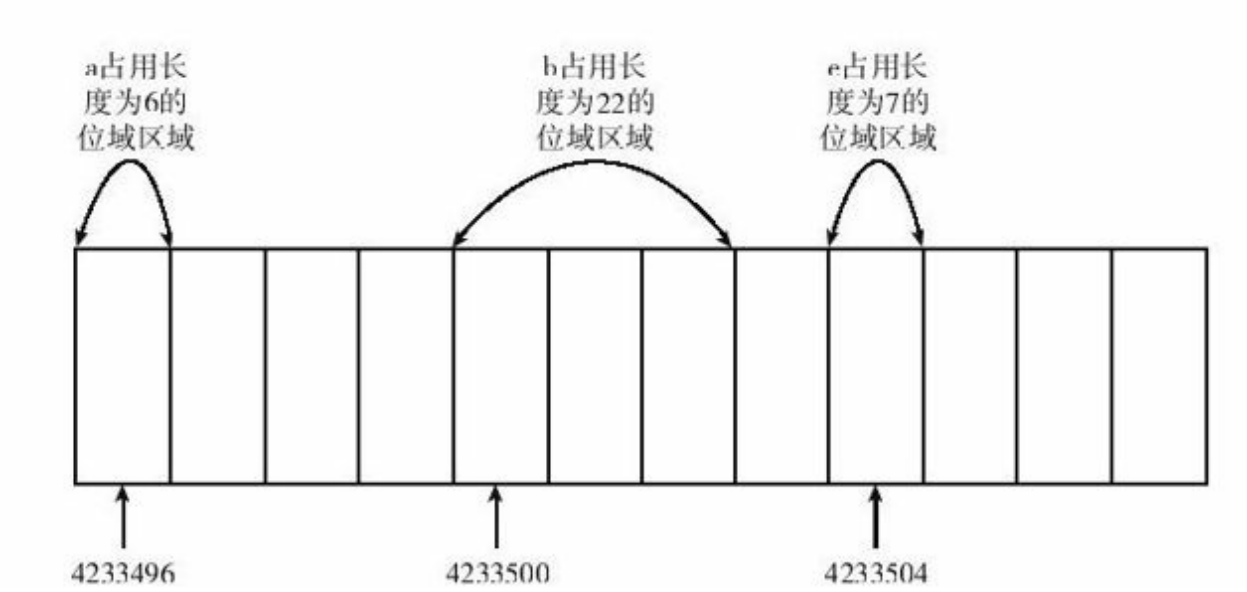

图　1-14　位域变量data的内存结构（3）

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。

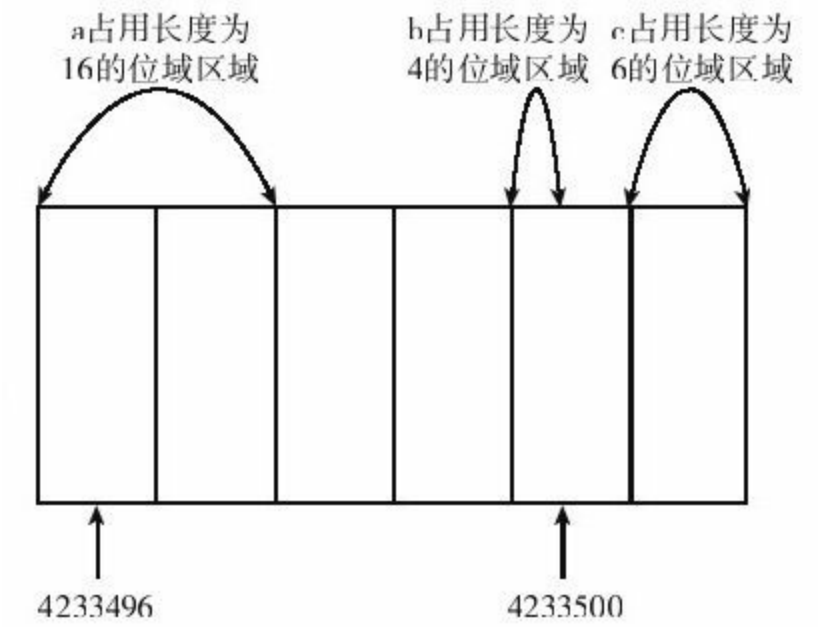

图　1-15　位域变量data的内存结构（4）

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。

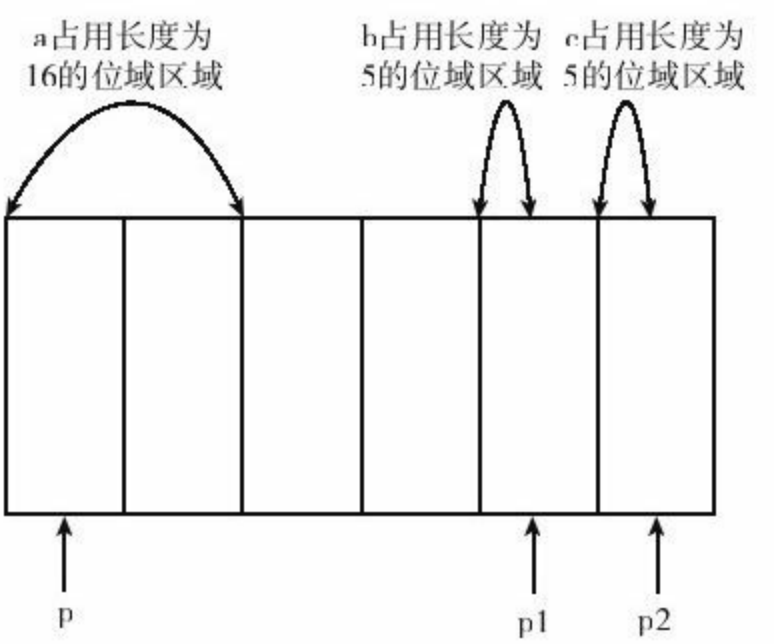

图　1-16　位域变量data的内存结构（5）

＆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_Pp,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。

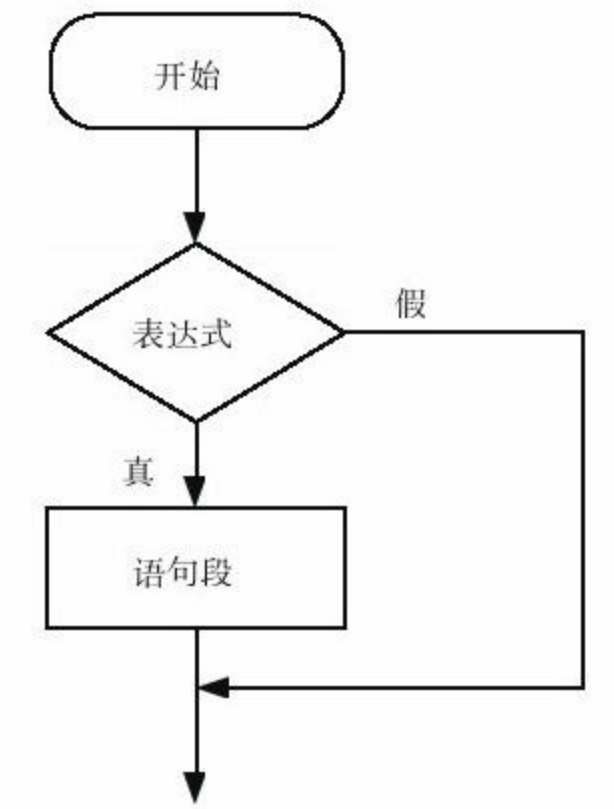

图　3-1　if语句流程图

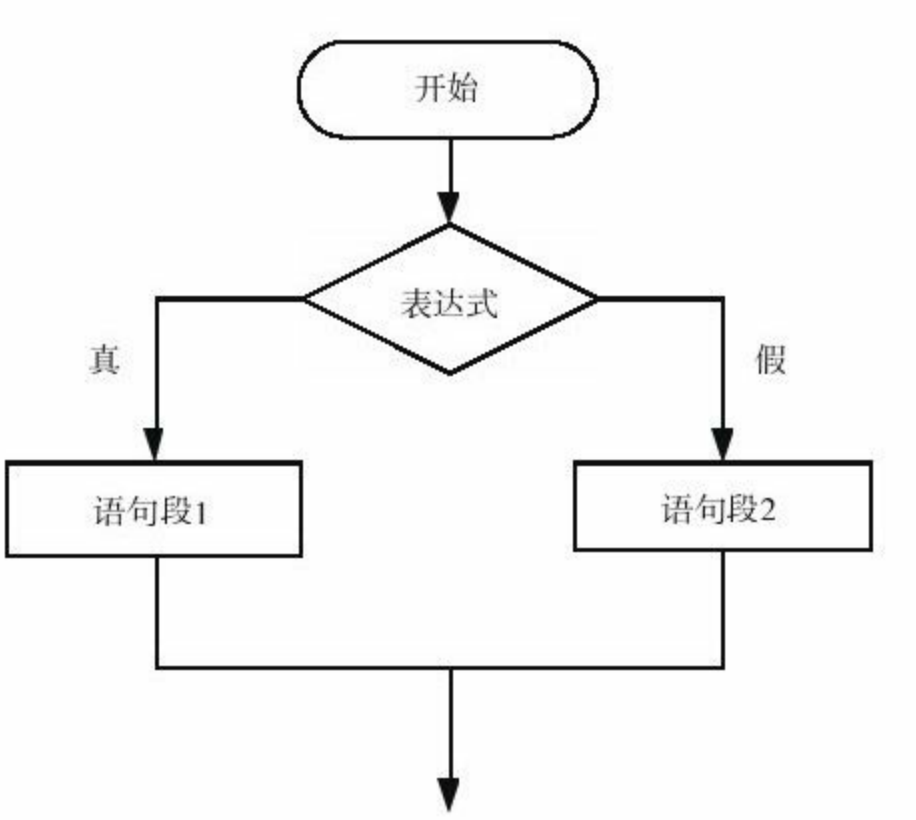

图　3-2　if-else语句流程图

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。

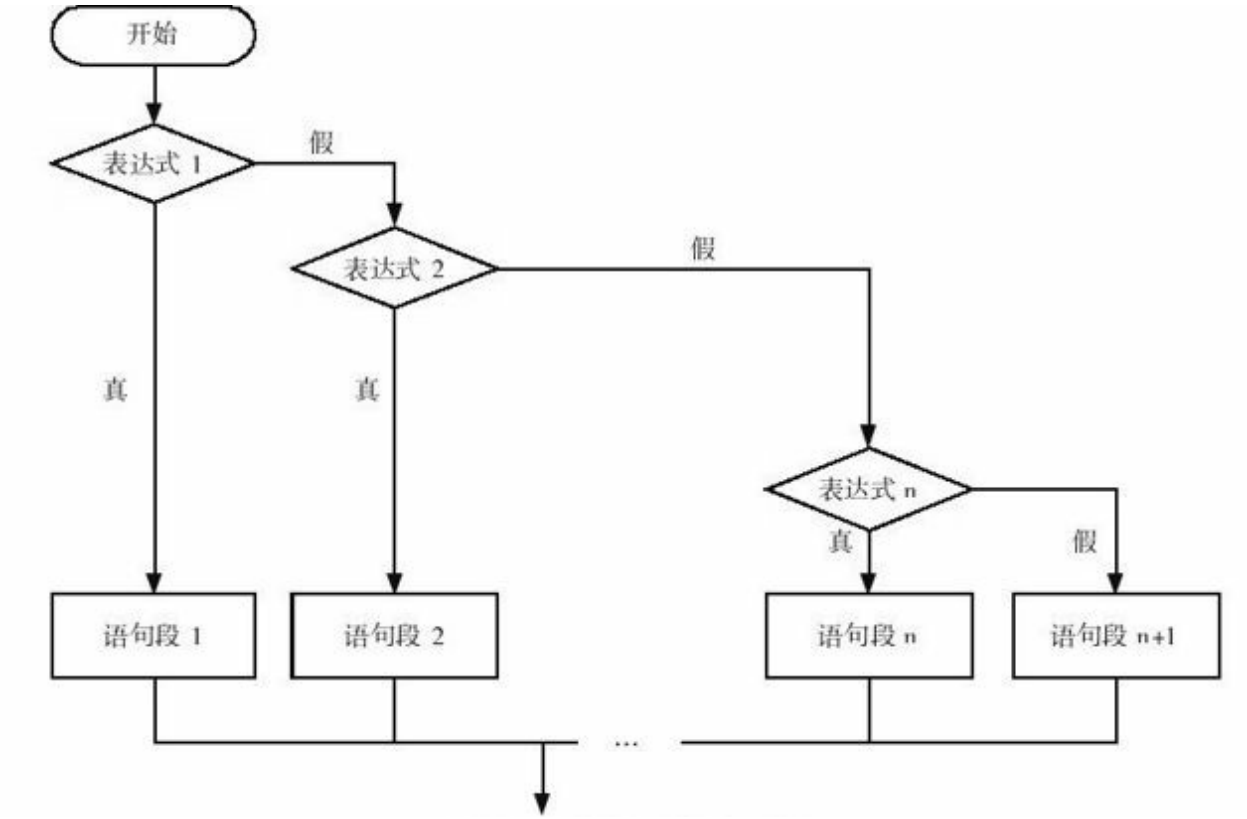

图　3-3　多重if语句流程图

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。

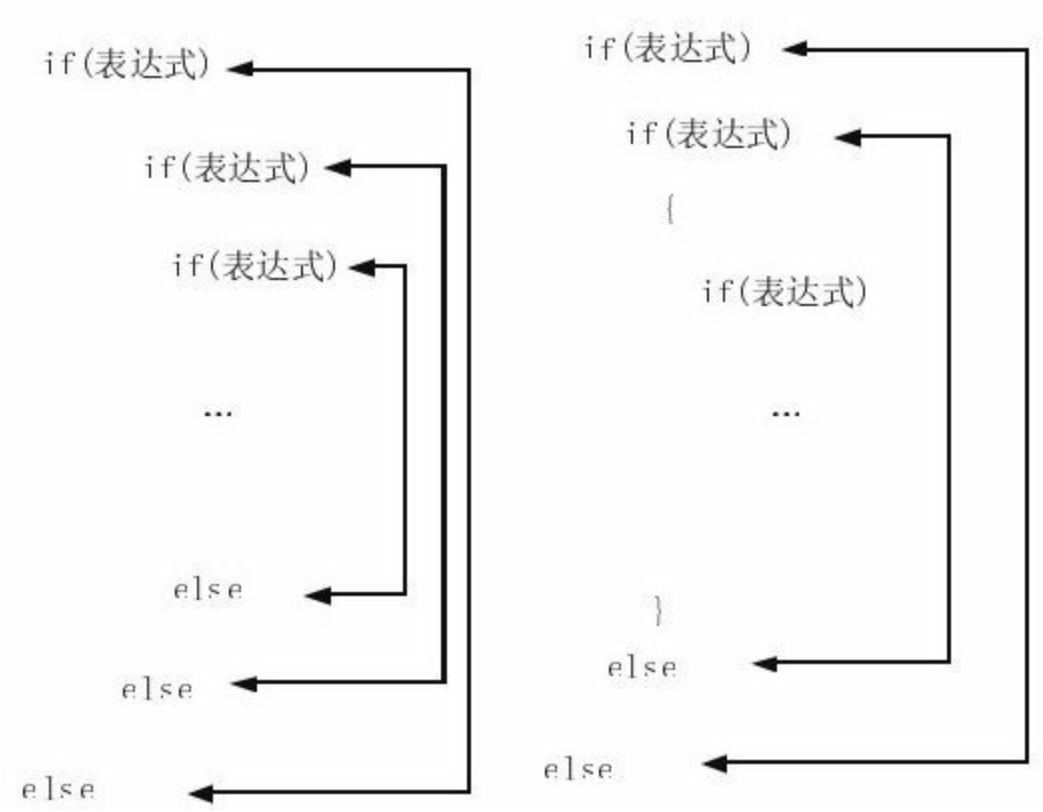

图　3-4　if-else语句的配对

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。

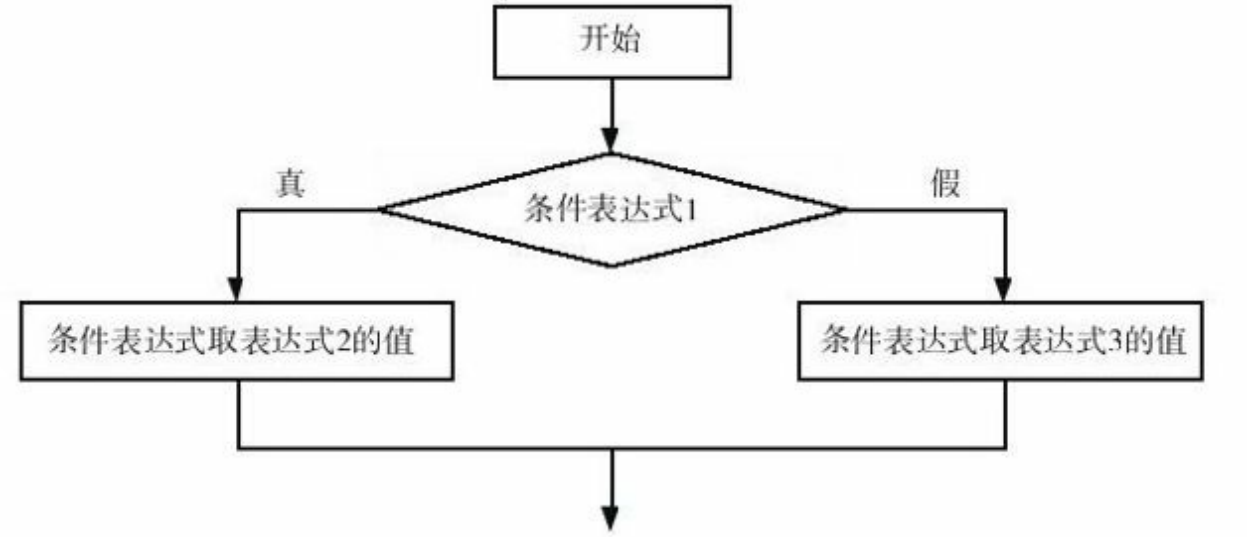

图　3-5　条件表达式流程图

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。

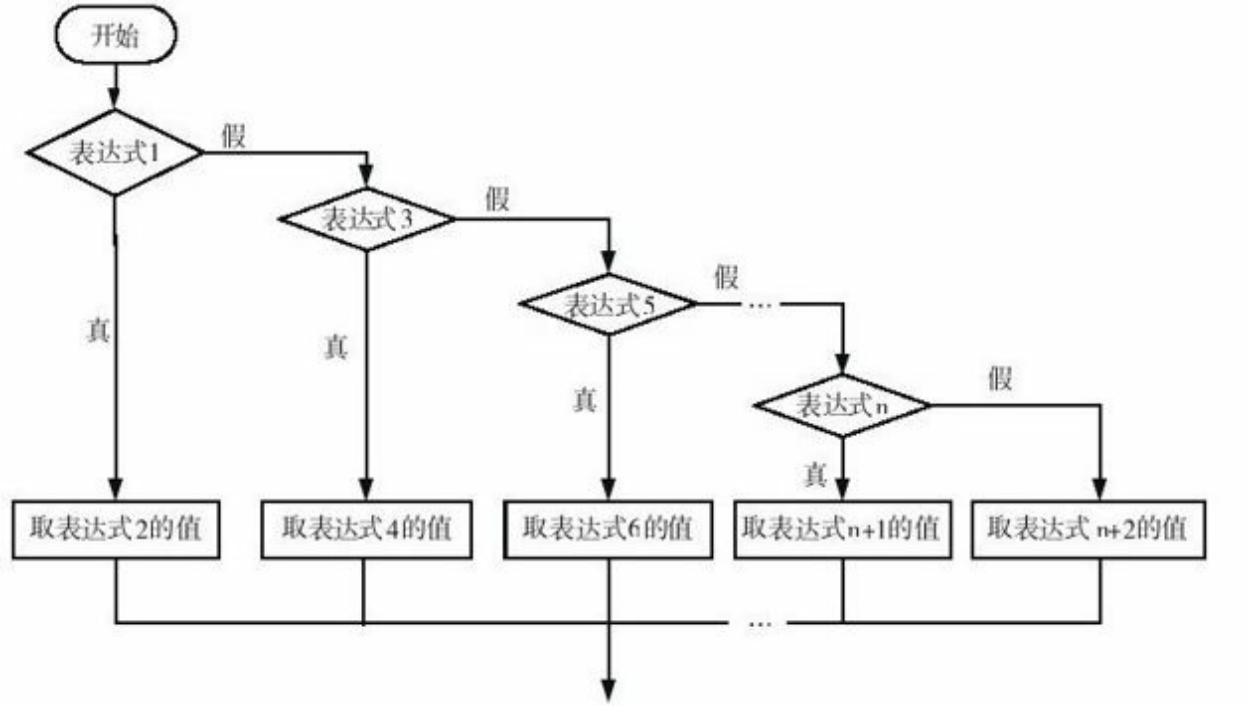

图　3-6　嵌套条件表达式流程图

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。

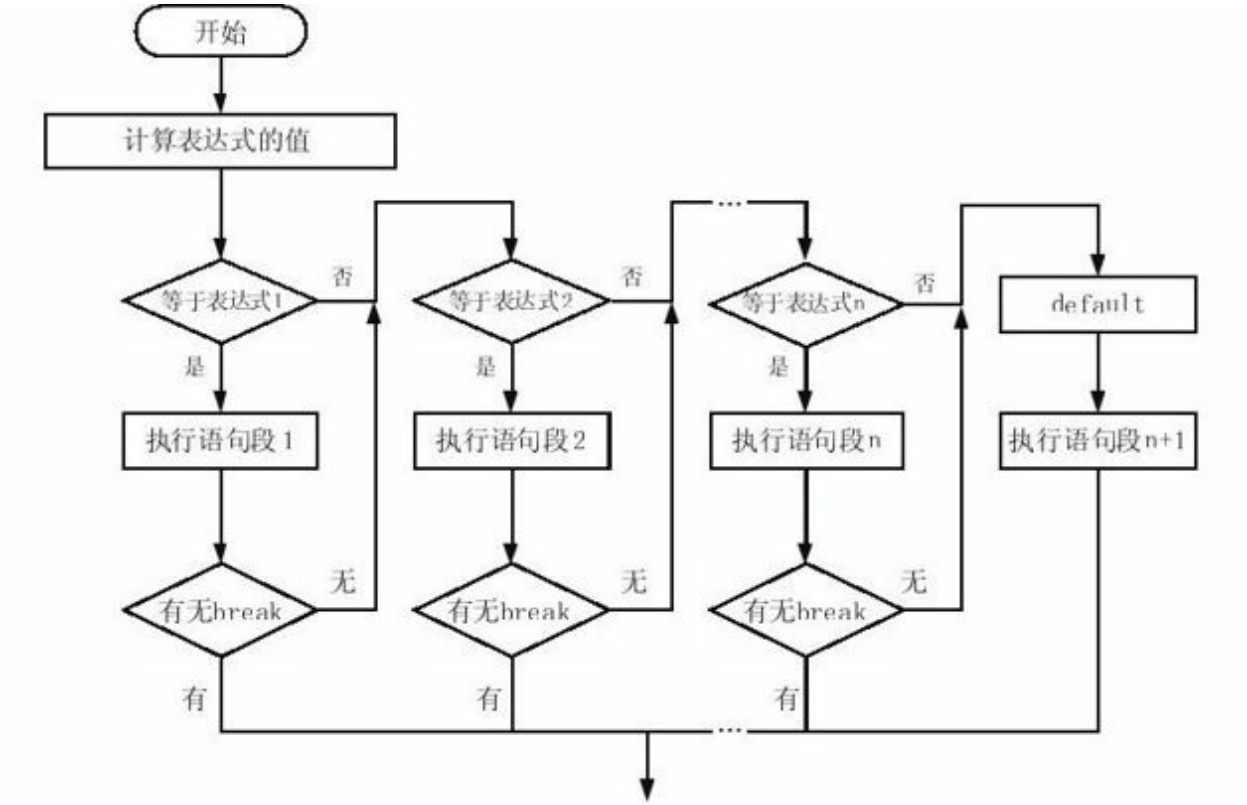

图　3-7　switch语句流程图

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。

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。

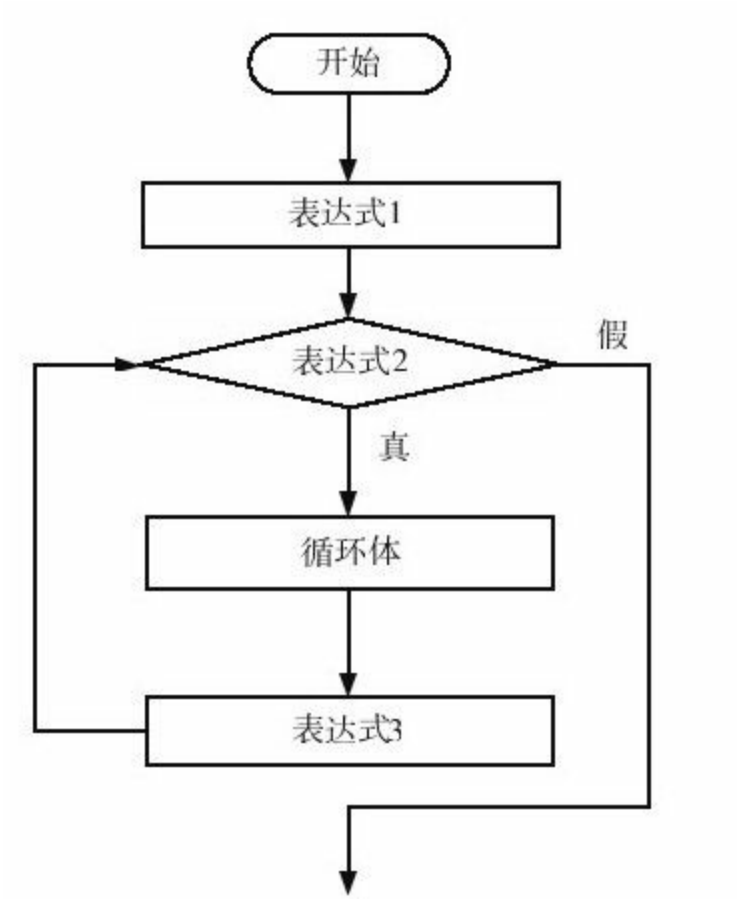

图　3-8　for循环语句流程图

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。

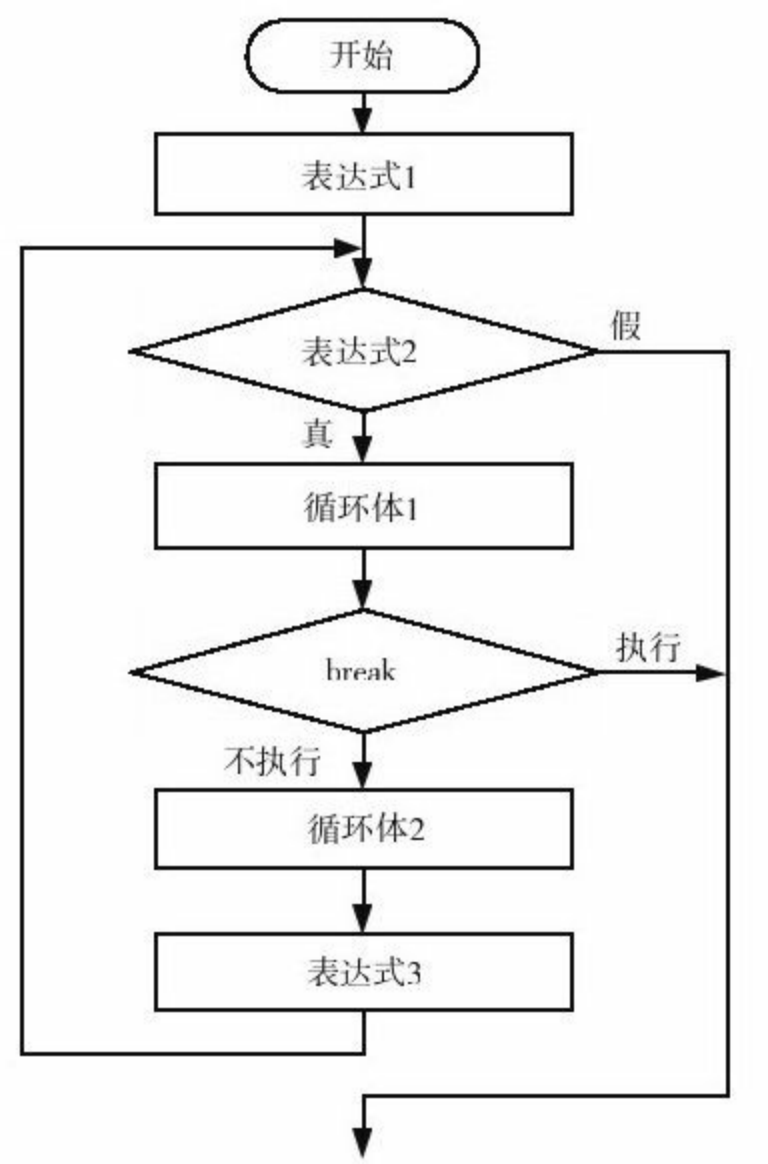

图　3-9　含有break语句的for循环流程图

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
:
4
0
7
```

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。

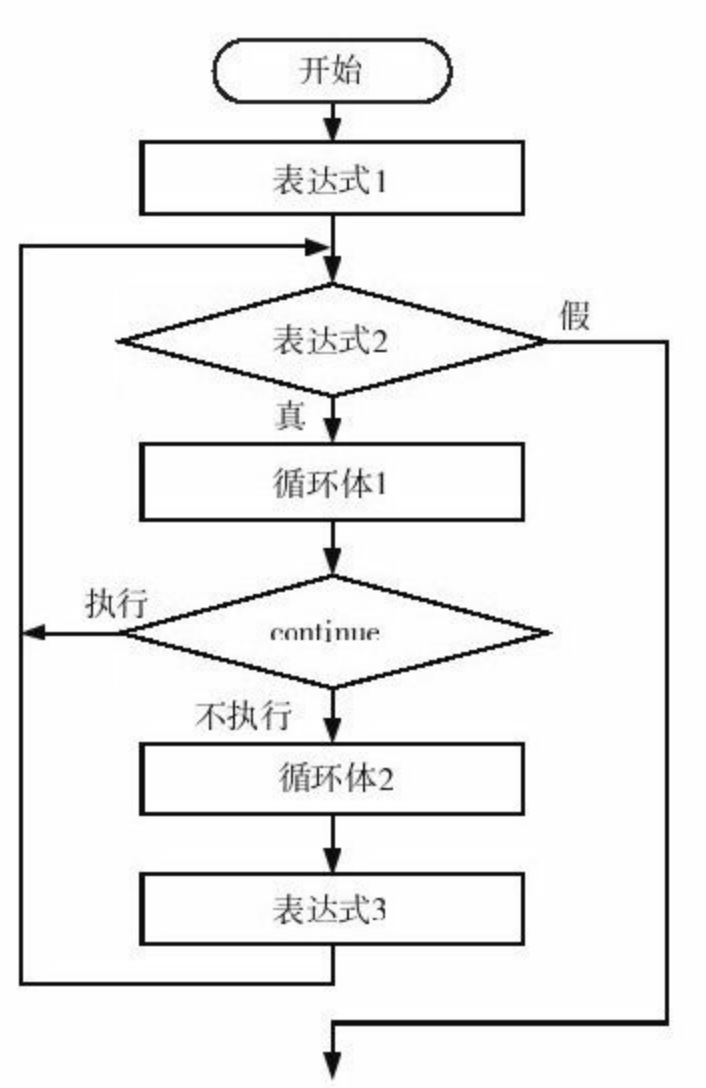

图　3-10　含有continue语句的for循环流程图

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。

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。

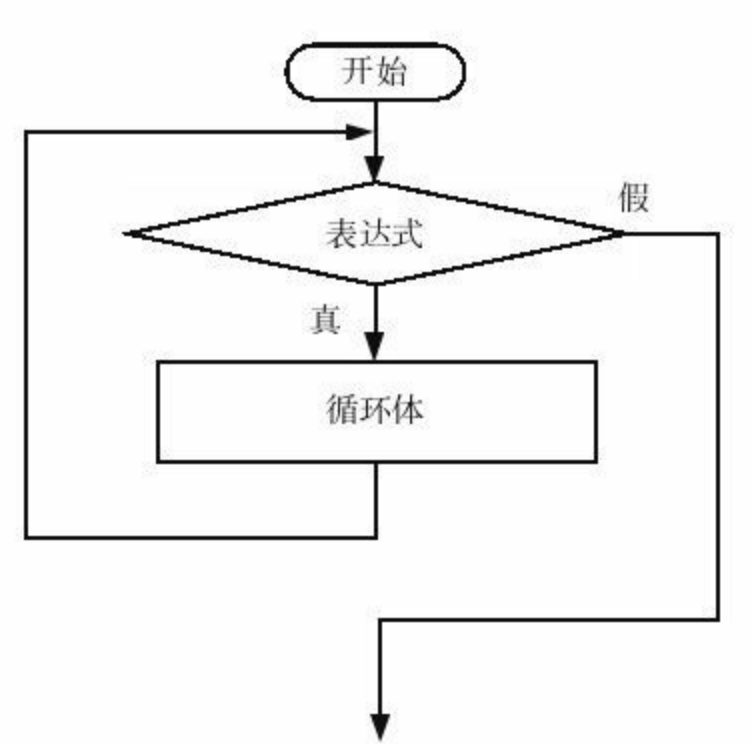

图　3-11　while循环流程图

```
#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。

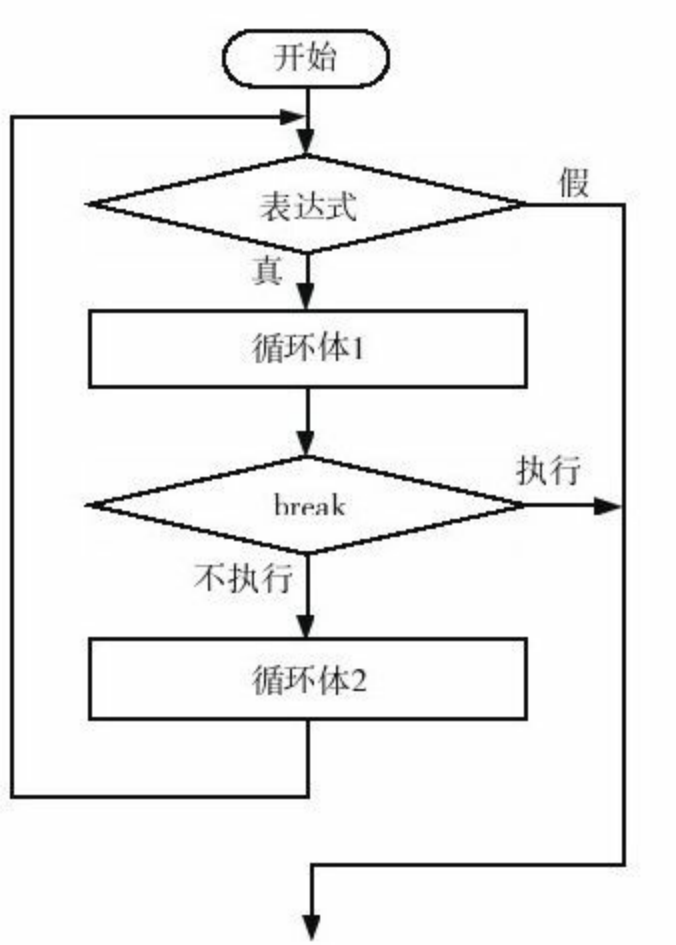

图　3-12　含有break语句的while循环流程图

```
#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。

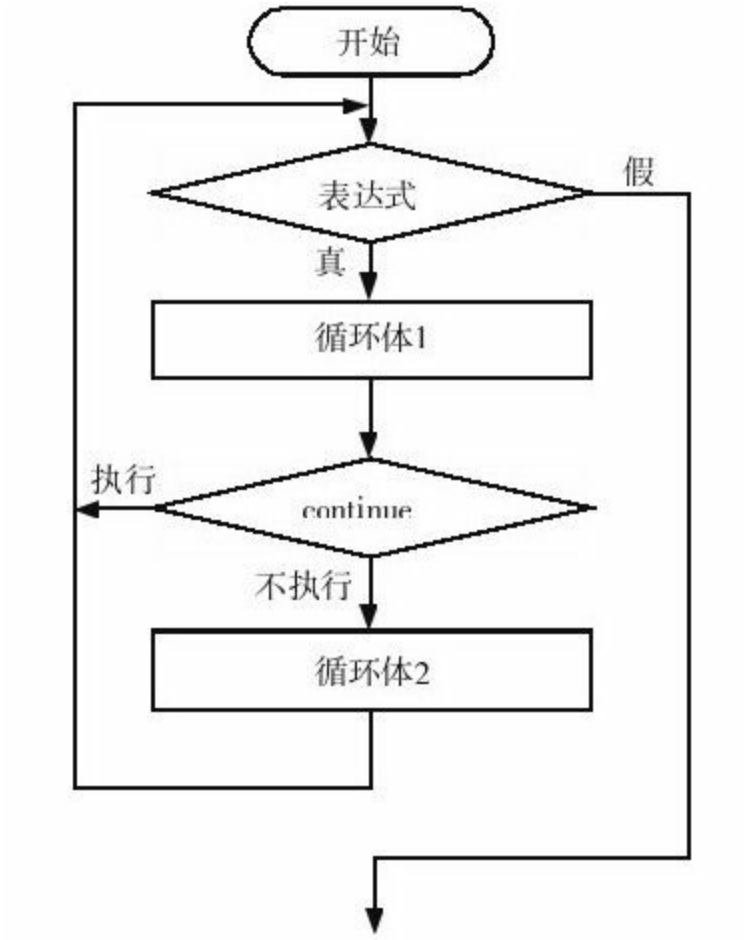

图　3-13　含有continue语句的while循环流程图

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。

```
#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。

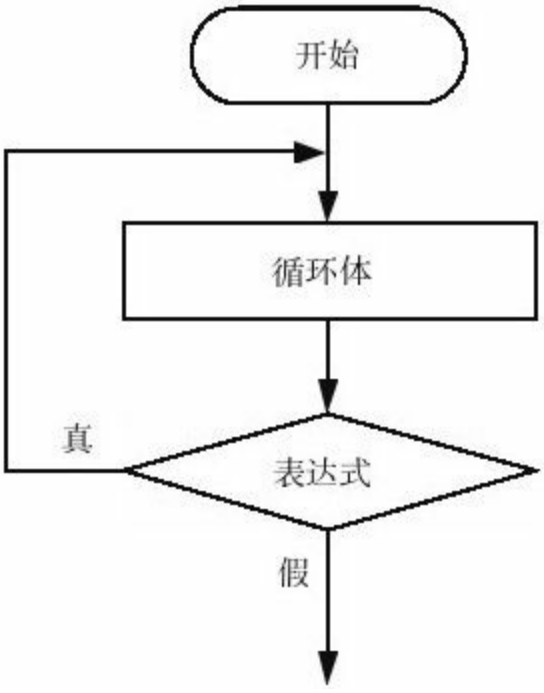

图　3-14　do-while循环流程图

```
#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。

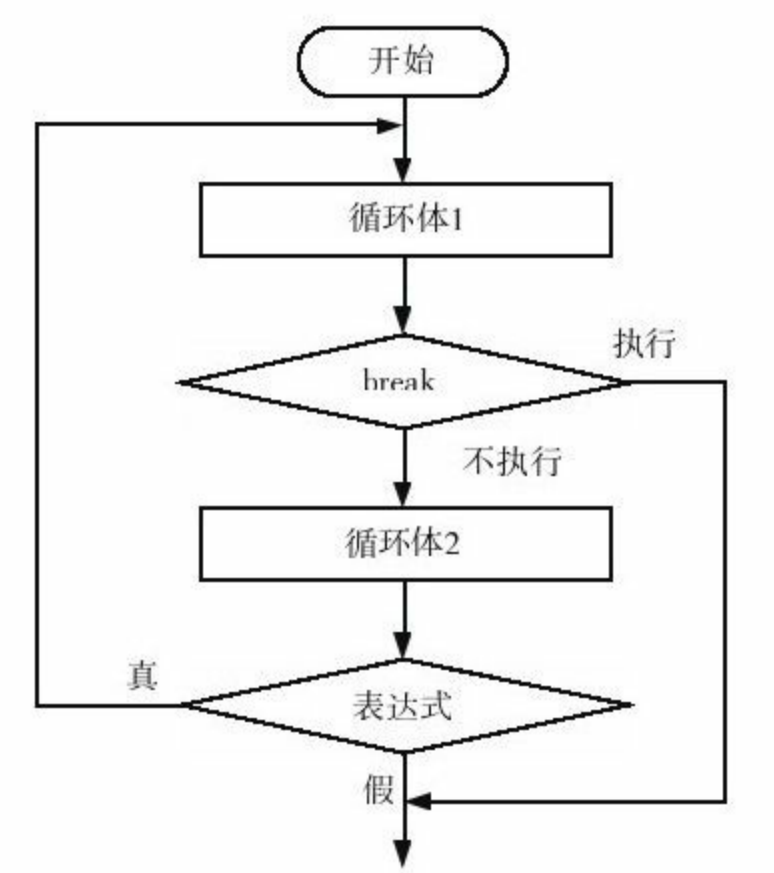

图　3-15　含有break语句的do-while循环流程图

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。

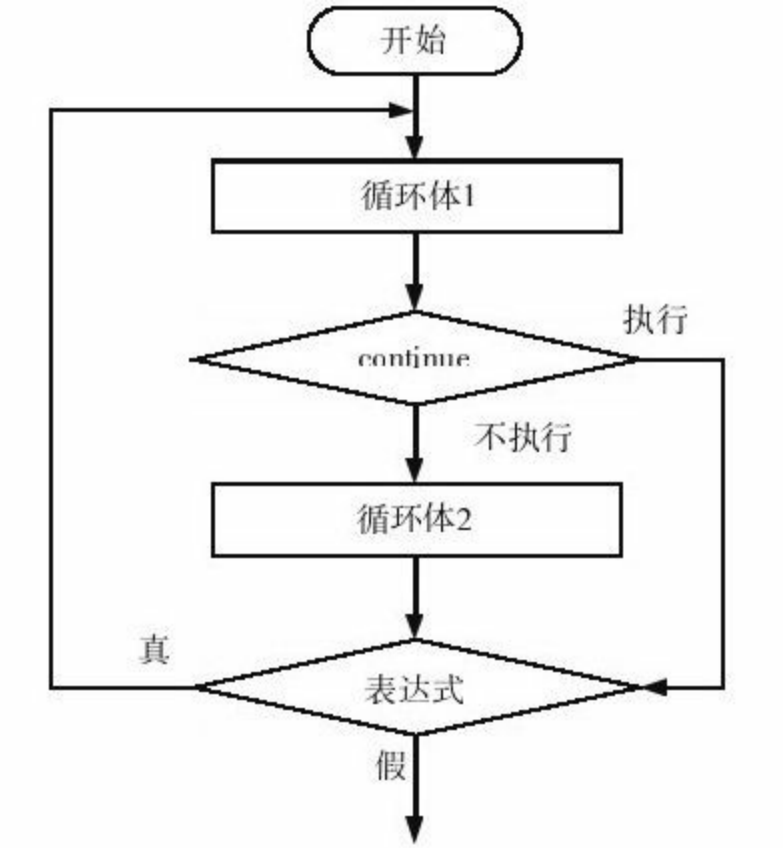

图　3-16　含有continue语句的do-while循环流程图

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。

```
#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。

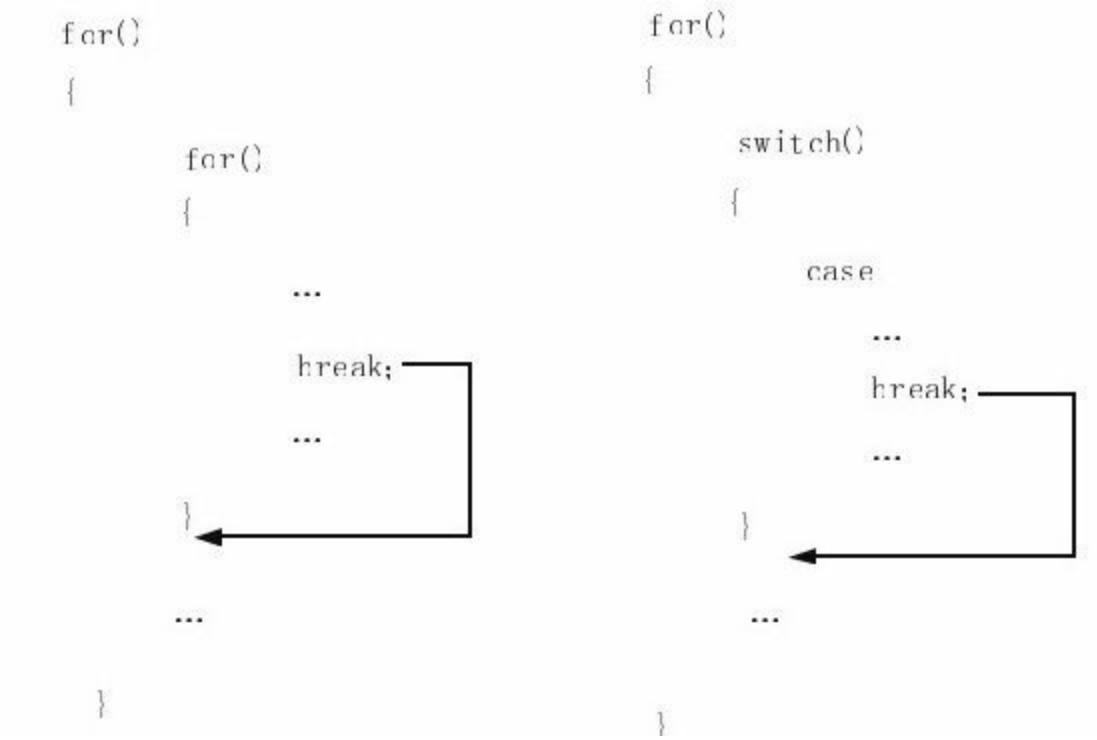

图　3-17　break语句

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。

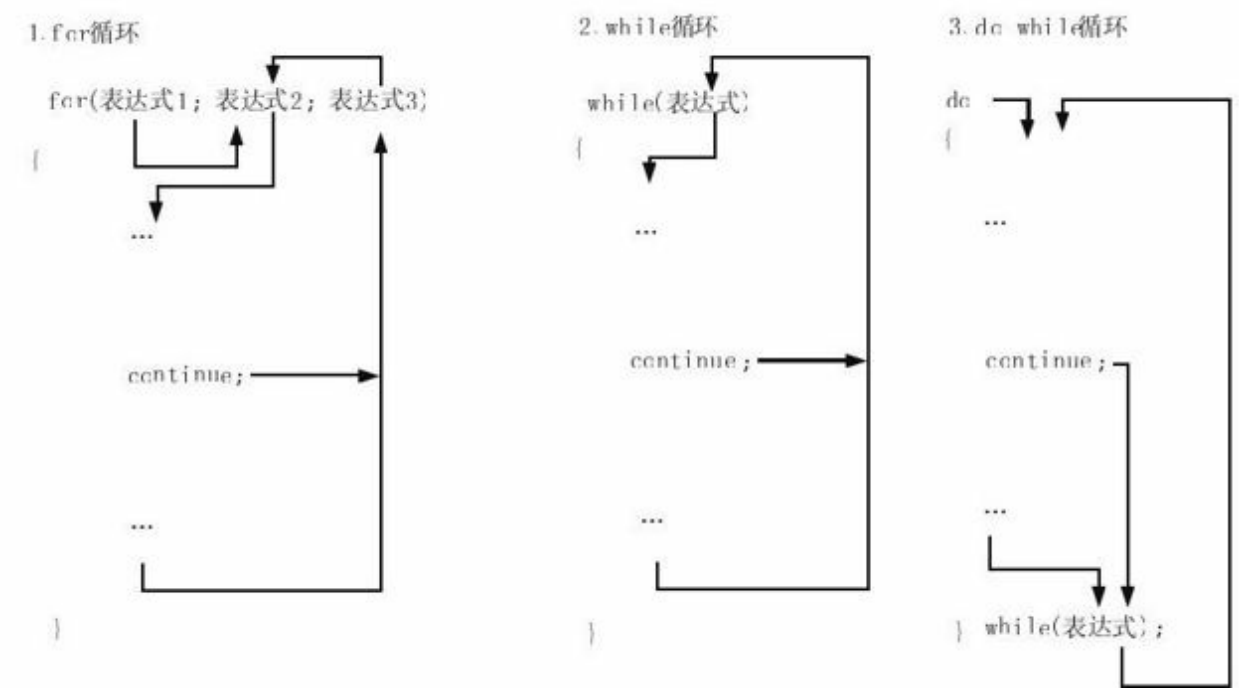

图　3-18　continue语句

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。

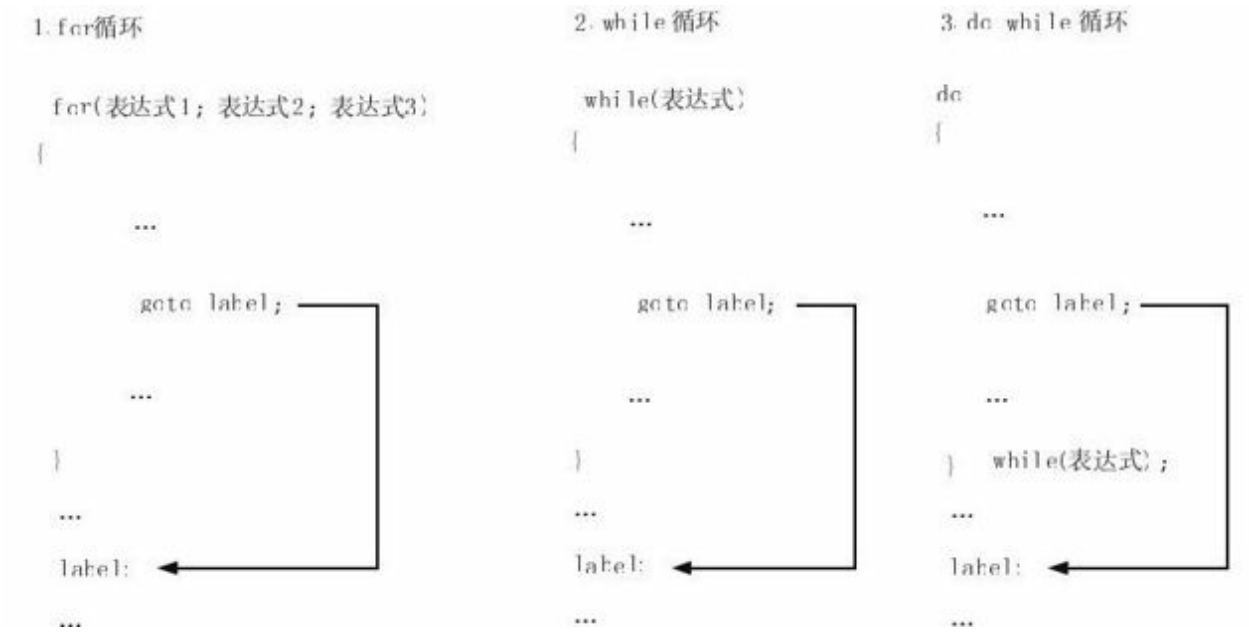

图　3-19　goto语句

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。

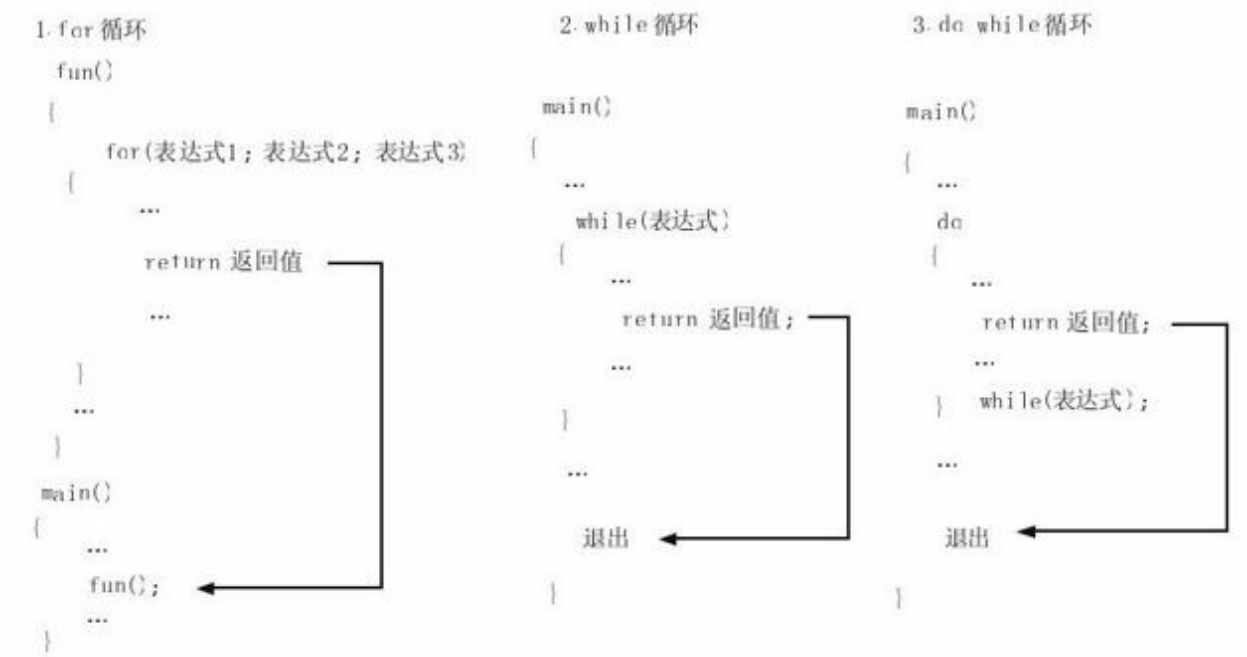

图　3-20　return语句

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
```

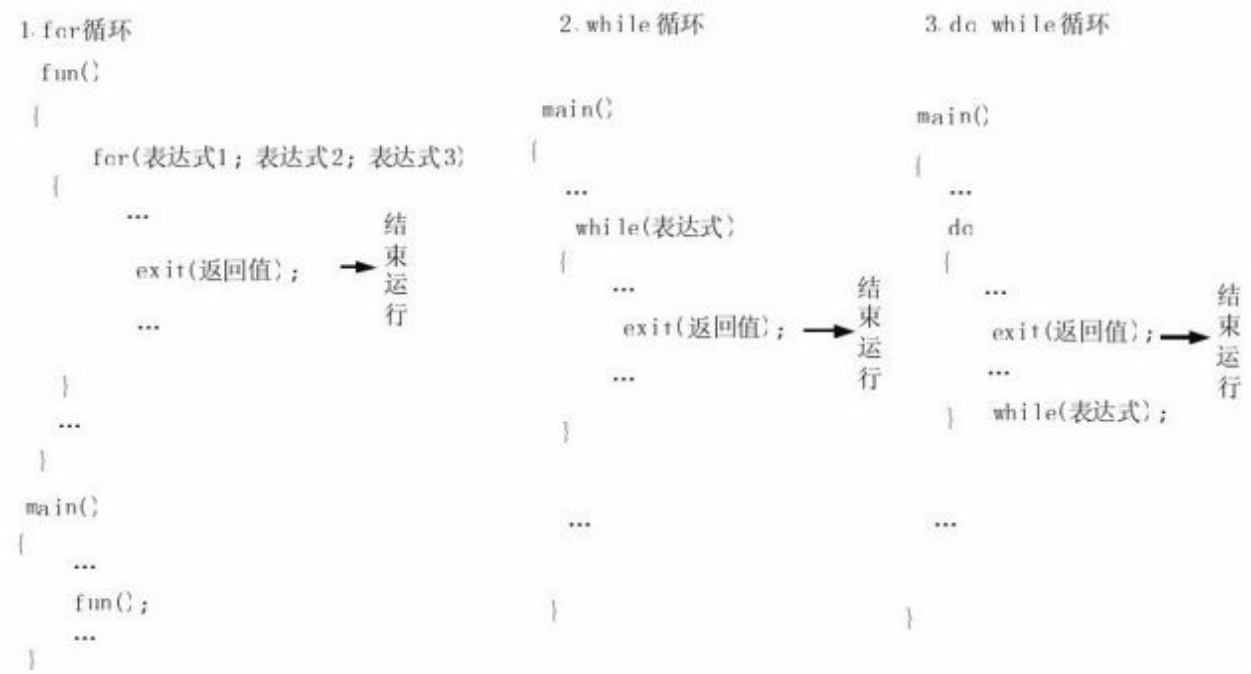

图　3-21　exit（）函数

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。

第4章　数组

在现实生活中，我们经常会遇到类似这样的问题：计算某个班级的学生数学平均成绩。学生可能有几十人甚至上百人，我们不可能在编程时直接定义那么多的变量名来分别表示每个学生的数学成绩，因为这样不仅使代码的可读性很差，还不便于代码的维护。这就需要采用一种简单的办法将这些相同类型的数据表示出来，这时就会用到数组。但是如果需要处理的是一幅图片，采用一般静态数组的方法就可能因为数据量过大而导致栈的溢出，这时就需要通过定义动态数组来实现。本章主要学习静态数组和动态数组的使用，按照由易到难的顺序讲解在数组使用中都有哪些易错点，以及如何在编程中避免类似的错误。

4.1 一维数组的定义及引用

在讲解一维数组之前，先来看数组的定义。所谓数组，是指将那些具有相同类型的、数量有限的若干个变量通过有序的方法组织起来的一种便于使用的形式。数组属于一种构造类型，其中的变量被称为数组的元素。数组元素的类型可以是基本数据类型，也可以是特殊类型和构造类型。

先来看一维数组，它是最简单的数组类型，其定义的一般形式为：

```
类型说明符 数组名[常量表达式];
```

其中，类型说明符是数组中每个元素的类型，常量表达式是数组元素的个数。

在使用一维数组的时候需要留意以下两个要点。

常量表达式的值必须是正整数，常量表达式中不允许含有变量，而数组的命名同样要遵循标识符的命名规则。我们可以通过下面的代码来具体分析。

```
#include<stdio.h>
void main(void)
{
int n=9;
int arr[n];
return;
```

```
}
```

以上代码定义了一个整型变量n=9，接下来把n作为数组定义过程中的常量表达式，编译时会出现“error C2133：'arr'：unknown size”错误信息，但是使用下面的方法可以避免上述错误。

```
#include＜stdio.h＞
#define N 9
void main（void）
{
int arr[N];
return;
}
```

以上代码在预处理时将N的值替换为9，所以编译时不会出现错误。

数组元素的引用。引用数组元素的一般形式为：

```
数组名[下标]
```

在C语言中，由于数组的起始元素下标为0，所以“数组名[N]”所引用的是数组中第N+1个元素。下面通过一个简单的代码来看数组元素的引用。

```
#include＜stdio.h＞
#define N 9
void main（void）
{
int arr[N];
int i;
```

```
for（i=0; i＜N; i++）
{
arr[i]=i+1;
printf（"arr[%d]=%d\t", i,arr[i]）;
if（0==（i+1）%3）
printf（"\n"）;
}
return;
}
```

运行结果：

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

在上面的代码中，定义了一个含有9个元素的一维数组arr，在引用数组中的元素时，采用“数组名[下标]”的方式，将其中的每个元素视为一个普通的变量来进行操作。需要注意的是，因为定义的数组arr仅含有9个元素，所以在使用的过程中，下标值不能超过8，否则就会出现下标越界的错误，示例如下：

```
#include＜stdio.h＞
#define N 9
void main（void）
{
int arr[N];
arr[9]=9;
return;
}
```

以上代码在编译时不会提示任何错误信息，因为编译器并不会做数组元素下标是否越界的检测，所以在运行的时候就会出现错误。

讲解完一维数组的注意要点，接下来通过一段代码来了解一维数组在内存中是如何存放的。

```
#include＜stdio.h＞
#define N 4
void main（void）
{
int arr[N];
int i;
for（i=0; i＜N; i++）
{
arr[i]=i;
printf（"&arr[%d]=%d\n", i, &arr[i]）;
}
return;
}
```

运行结果：

```
&arr[0]=1245040
&arr[1]=1245044
&arr[2]=1245048
&arr[3]=1245052
```

在以上代码中定义了含有4个整型元素的一维数组，每个元素占用4字节，下标从0开始，到3结束。arr数组元素在内存中的存储结构如图4-1所示。

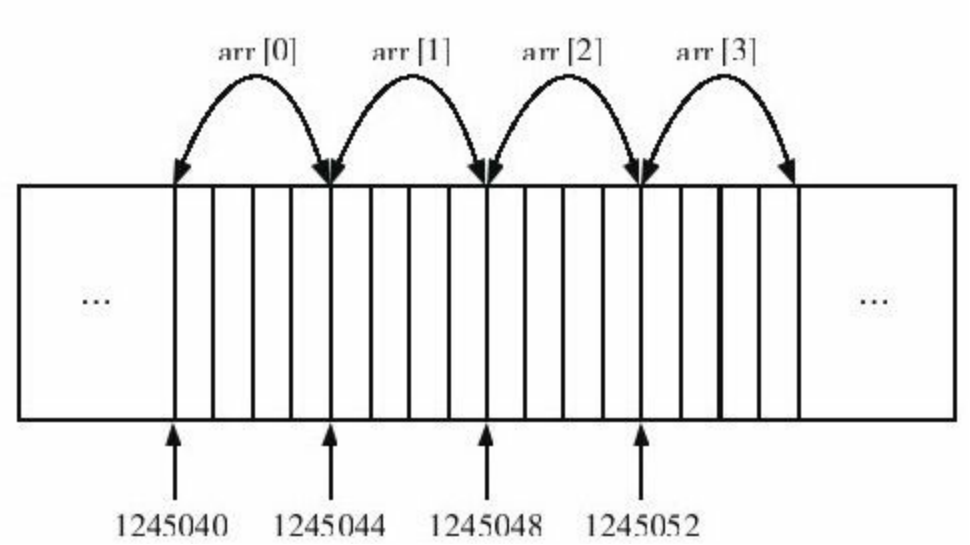

图　4-1　arr数组元素的内存结构

从图4-1中可以看出，数组元素从第一个存储地址到最后一个存储地址呈依次递增的趋势，每个数组元素所占用的内存大小为元素类型所占用的内存大小，这里，元素类型为整型，占用4字节。当然，数组类型也可以是自定义的数据类型，示例如下：

```
#include＜stdio.h＞
#define N 4
struct_stu
{
char name[20];
int score;
}stu[N];
void main（void）
{
int i;
printf（"自定义结构类型_stu占用的内存大小为：%d\n"，sizeof（_stu））；
for（i=0；i＜N；i++）
{
printf（"＆stu[%d]=%d\n"，i，＆stu[i]）；
}
return;
}
```

运行结果：

```
自定义结构类型_stu占用的内存大小为：24
&stu[0]=4375128
&stu[1]=4375152
&stu[2]=4375176
&stu[3]=4375200
```

这里的自定义数据类型在内存中的存储结构如图4-2所示。

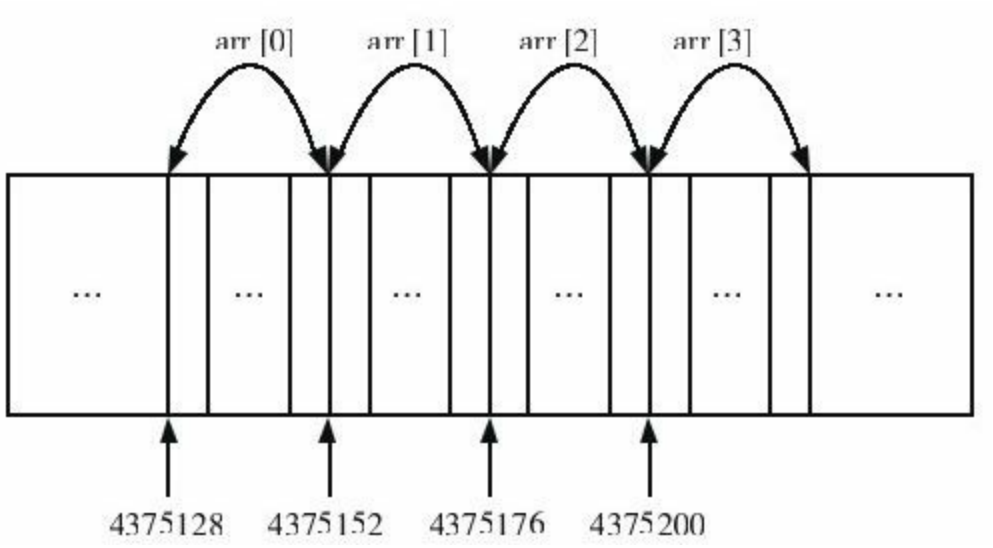

图　4-2　stu数组元素的内存结构

从图4-2中看到，自定义的类型所占用的内存大小为24字节，所以数组stu中每个数组元素占用内存的大小为24字节，存储方式同样从低地址到高地址，并且所有数组元素都存储在一个连续的内存单元中，数组所占用的内存大小为数组元素类型所占用的内存大小乘以数组元素的个数。

对于一维数组还需要注意的就是它的初始化问题，接下来看看一维

数组初始化的注意事项。一维数组初始化的一般形式为：

```
数组类型 数组名[N]={数组元素1，数组元素2，……数组元素N}；
```

下面通过代码来看一维数组的初始化。

```
#include＜stdio.h＞
#define N 4
void main（void）
{
int i;
int arr[N]={1，2，3，4};
for（i=0; i＜N; i++）
{
printf（"arr[%d]=%d\n", i,arr[i]）;
}
return;
}
```

运行结果：

```
arr[0]=1
arr[1]=2
arr[2]=3
arr[3]=4
```

从上面的运行结果可以看出，这里在定义数组时对数组元素进行的初始化，和前面采用循环的方式单独对数组中每个元素进行初始化的方法一样，但是，切不可在定义数组后再对整个数组进行赋值操作，例如：

```
#include＜stdio.h＞
#define N 4
```

```
void main（void）
{
int i;
int arr[N];
arr[N]={12，34};
return;
}
```

或者

```
#include＜stdio.h＞
#define N 4
void main（void）
{
int i;
int arr[N];
arr={12，34};
return;
}
```

这两种方式都是错误的，C语言不允许在定义了数组之后再采用以上方式对整个数组赋值，定义了数组之后，只能对数组元素中的单个元素进行赋值操作。

在对数组元素进行整体赋值的时候，如果初始化的元素的个数小于数组的长度，那么结果会怎么样呢？看看下面的代码：

```
#include＜stdio.h＞
#define N 4
void main（void）
{
int i;
int arr[N]={1，2};
for（i=0；i＜N；i++）
{
printf（"arr[%d]=%d\n"，i,arr[i]）；
}
```

```
return;
}
```

运行结果：

```
arr[0]=1
arr[1]=2
arr[2]=0
arr[3]=0
```

从上面对数组的初始化方式和运行结果发现，如果初始化的数组元素个数小于数组长度，那么编译时会将那些没有初始化的数组元素赋值为0。如果初始化元素的个数大于数组长度，又会怎么样呢？看看下面的代码：

```
#include＜stdio.h＞
#define N 4
void main（void）
{
int i;
int arr[N]={1，2，3，4，5，6};
for（i=0; i＜N; i++）
{
printf（"arr[%d]=%d\n", i,arr[i]）;
}
return;
}
```

这段代码在编译时会提示“error C2078：too many initializers”错误，提示初始化元素过多。

在用这种方法进行初始化时也可以省略数组长度，此时数组长度由

初始化的数组元素个数来决定，例如：

```
#include＜stdio.h＞
void main（void）
{
int i;
int arr[]={1，2，3，4，5，6};
for（i=0; i＜6; i++）
{
printf（"arr[%d]=%d\t", i,arr[i]）;
if（0==（i+1）%3）
printf（"\n"）;
}
return;
}
```

运行结果：

```
arr[0]=1 arr[1]=2 arr[2]=3
arr[3]=4 arr[4]=5 arr[5]=6
```

在上面的代码中，在定义数组arr的时候省略了数组长度，编译器会根据初始化的数组元素个数来决定所采用的数组的长度。

4.2 二维数组的定义及引用

前面介绍了一维数组，接下来介绍如何定义和使用二维数组。

二维数组定义的一般形式如下：

```
类型说明符 数组名[常量表达式1][常量表达式2];
```

与一维数组的定义唯一的不同是多了一个常量表达式2，其中，常量表达式1为第一维的长度，常量表达式2为第二维的长度。通常在处理二维数组的时候，为了便于理解，都将数组视为一个矩阵，常量表达式1表示矩阵的行数，而常量表达式2表示矩阵的列数。与一维数组一样，在定义二维数组时，常量表达式同样不能为变量。下面先通过一段代码来看二维数组的定义。

```
#include<stdio.h>
#define M 4
#define N 3
void main(void)
{
int i,j;
int arr[M][N];
for(i=0; i<M; i++)
{
for(j=0; j<N; j++)
{
printf("&arr[%d][%d]=%d\t", i,j, &arr[i][j]);
}
printf("\n");
}
return;
}
```

运行结果：

```
&arr[0][0]=1245000&arr[0][1]=1245004&arr[0][2]=1245008
&arr[1][0]=1245012&arr[1][1]=1245016&arr[1][2]=1245020
&arr[2][0]=1245024&arr[2][1]=1245028 arr[2][2]=1245032
&arr[3][0]=1245036&arr[3][1]=1245040&arr[3][2]=1245044
```

将二维数组arr视为一个矩阵，图4-3显示了数组中每个元素在矩阵中的存放位置。

由图4-3可知，数组中各个元素在矩阵中对应的位置由二维数组的两个下标决定。我们可以将定义的二维数组int arr[4][3]视为由arr[4]和int[3]两部分构成，将arr[4]视为一个整型一维数组，其中含有4个元素arr[0]、arr[1]、arr[2]、arr[3]，每个元素都是int[3]类型的，也就是说，每个元素又是一个一维数组，每个一维数组含有3个元素，如arr[0]含有arr[0][1]、arr[0][1]、arr[0][2]三个元素。

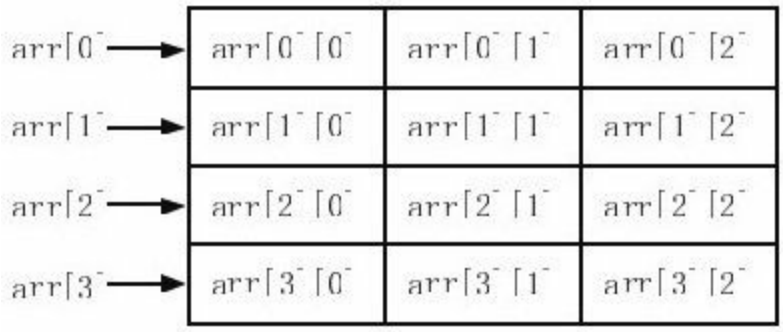

图　4-3　二维数组arr元素的矩阵存放方式

知道了二维数组的这种特殊结构之后，接下来通过图4-4来了解二

维数组在内存中的存储结构。

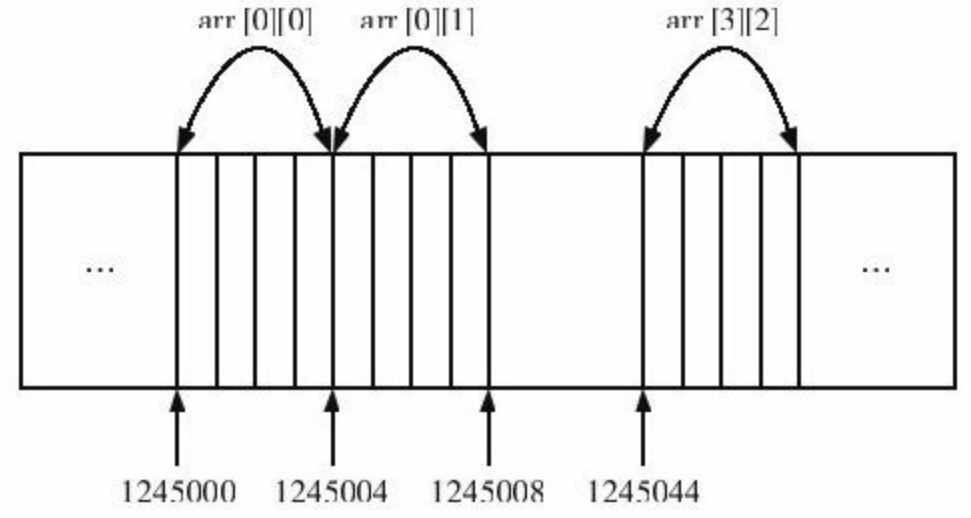

图　4-4　二维数组arr在内存中的存储结构

通过上述二维数组在内存中的存储结构图可以发现，二维数组中的所有元素都存储在一片连续的内存单元中，所占用的内存大小为元素类型所占用的内存大小乘以第一维及第二维的长度。如果以矩阵的方式来分析二维数组的存储方式，那么先从矩阵第一行从左往右依次存储完所有元素，然后按照同样的方法存储第二行的所有元素，直到存储完所有数组元素为止。

接下来看二维数组的一般引用形式：

```
数组名[表达式1][表达式2]
```

如果用矩阵的方式来描述二维数组，那么这里的“表达式1”就是行下标，“表达式2”就是列下标，它们的取值均从0开始。例如定义一个二

维数组a[M][N]，它的行下标的取值范围为0～M-1，列下标的取值范围为0～N-1。与之前讲解的一维数组一样，在使用二维数组元素的过程中也要避免下标越界的情况出现，看看下面的代码：

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[M][N];
for（i=0; i＜M; i++）
{
for（j=0; j＜N; j++）
{
arr[i][j]=i+j;
printf（"arr[%d][%d]=%d\t", i,j, arr[i][j]）;
}
printf（"\n"）;
}
return;
}
```

运行结果：

```
arr[0][0]=0 arr[0][1]=1 arr[0][2]=2
arr[1][0]=1 arr[1][1]=2 arr[1][2]=3
arr[2][0]=2 arr[2][1]=3 arr[2][2]=4
arr[3][0]=3 arr[3][1]=4 arr[3][2]=5
```

在上面的代码中，虽然在数组定义的过程中不含有变量，但是在数组引用的时候却含有变量。对数组中元素的引用只需采用“数组名[行下标][列下标]”即可，值得注意的是行下标和列下标的取值范围，使用数组元素时同样可以将数组中的每个元素看成一个简单的变量进行处理。

在上面代码的二重循环中，内层循环控制的是二维数组的列下标，外层循环控制的是二维数组的行下标。通过循环变量的改变来逐一取出二维数组中的每个元素，并对其进行赋值和打印操作。接下来看看二维数组的初始化方法，以下代码是在定义二维数组的过程中对二维数组元素进行赋值操作。

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[M][N]={
{1，2，3}，
{4，5，6}，
{7，8，9}，
{10，11，12}};
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
printf（"arr[%d][%d]=%d\t"，i,j，arr[i][j]）；
}
printf（"\n"）；
}
return;
}
```

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=5 arr[1][2]=6
arr[2][0]=7 arr[2][1]=8 arr[2][2]=9
arr[3][0]=10 arr[3][1]=11 arr[3][2]=12
```

上面代码对二维数组进行初始化时，将每一行数组元素的初始化值都放到一个花括号内，在花括号内采用逗号来分隔数组元素的初始化值，这是一种直观的初始化方法。就如前面分析的，二维数组在内存中的存储方式与一维数组是相同的，所有元素都存储在同一片连续的内存单元中，所以也可以采用一维数组的方式来对其进行初始化，例如：

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[M][N]={1，2，3，4，5，6，7，8，9，10，11，12};
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
printf（"arr[%d][%d]=%d\t"，i,j，arr[i][j]）；
}
printf（"\n"）；
}
return;
}
```

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=5 arr[1][2]=6
arr[2][0]=7 arr[2][1]=8 arr[2][2]=9
arr[3][0]=10 arr[3][1]=11 arr[3][2]=12
```

两种初始化方法的运行结果完全一致。当然，在初始化的过程中也可以对其中的部分元素进行赋值，只不过采用这两种方法对部分元素进

行赋值操作所产生的效果是不同的，下面通过代码来进行简单对比。

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[M][N]={1，2，3，4，5，6，7};
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
printf（"arr[%d][%d]=%d\t"，i,j，arr[i][j]）；
}
printf（"\n"）；
}
return;
}
```

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=5 arr[1][2]=6
arr[2][0]=7 arr[2][1]=0 arr[2][2]=0
arr[3][0]=0 arr[3][1]=0 arr[3][2]=0
```

采用一维数组的初始化方式对二维数组的部分元素进行赋值，如图4-5所示。

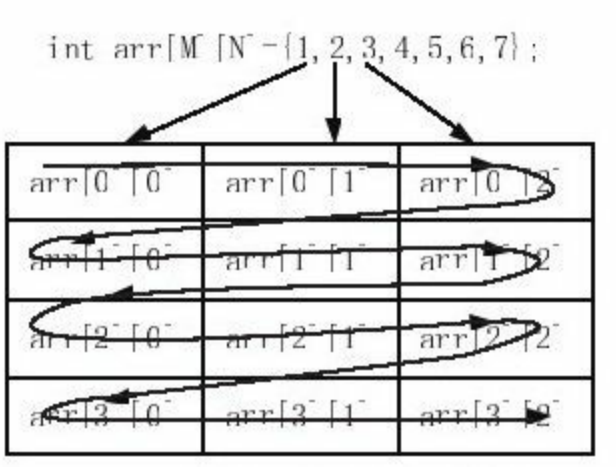

图　4-5　二维数组的初始化方式（1）

采用这种方式对二维数组进行赋值操作时，按照如图4-5所示的方法从arr[0][0]开始依次对数组中的每个元素进行赋值，如果赋值的个数小于数组元素的个数，那么剩余部分的元素初始值为0；而如果给定的初始值的个数大于二维数组元素的个数，那么编译时就会出现“errorC2078：too many initializers”错误，所以在初始化时要注意给定的初始值个数不要多于二维数组本身所能容纳的元素的个数。下面来看采用括号的方式对二维数组中的每行元素进行赋值的情况。

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[M][N]={{1，2，3},
{4，5},
{7},
{10}};
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
```

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

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=5 arr[1][2]=0
arr[2][0]=7 arr[2][1]=0 arr[2][2]=0
arr[3][0]=10 arr[3][1]=0 arr[3][2]=0
```

采用花括号的方法来对每行的数组元素单独赋值时要注意，花括号中的内容不能为空，至少要给定一个初始值，同时，初始值的个数也不能多于每行元素的个数。如图4-6所示为用花括号的方法对每行的数组元素单独赋值的初始化方式。

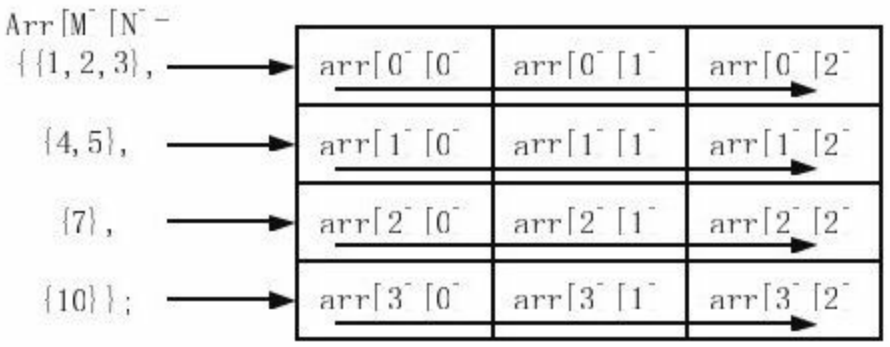

图　4-6　二维数组arr的初始化（2）

由图4-6可知，采用这种方式对二维数组中的元素进行赋值时，花括号中的每个值与相对应行的数组元素对应，初始值的个数小于行中数

组元素的个数时，将剩余数组元素的初始值赋值为0。

当然，还可以采用一种特殊的方式来定义并初始化二维数组，那就是在定义的过程中可以不指定表达式1，即不指定行的长度，但是必须通过表达式2来指定列的长度。同样可以使用前面介绍的两种方法对这种不指定表达式1的二维数组进行初始化，例如：

```
#include＜stdio.h＞
#define M 4
#define N 3
void main（void）
{
int i,j;
int arr[][N]={{1，2，3}，{4}，{7，8}，{10}};
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
printf（"arr[%d][%d]=%d\t"，i,j，arr[i][j]）；
}
printf（"\n"）；
}
return;
}
```

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=0 arr[1][2]=0
arr[2][0]=7 arr[2][1]=8 arr[2][2]=0
arr[3][0]=10 arr[3][1]=0 arr[3][2]=0
```

如果在定义的时候没有指定第一维的长度，编译器会在编译时根据花括号的个数来判断第一维的长度。以下代码为没有使用花括号的情

况。

```
#include＜stdio.h＞
#define N 3
void main（void）
{
int i,j，M;
int arr[][N]={1，2，3，4，5，6，7，8，9，10};
if（0==10%N）
M=10/N;
else
M=10/N+1;
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
printf（"arr[%d][%d]=%d\t"，i,j，arr[i][j]）；
}
printf（"\n"）；
}
return;
}
```

运行结果：

```
arr[0][0]=1 arr[0][1]=2 arr[0][2]=3
arr[1][0]=4 arr[1][1]=5 arr[1][2]=6
arr[2][0]=7 arr[2][1]=8 arr[2][2]=9
arr[3][0]=10 arr[3][1]=0 arr[3][2]=0
```

由于既没有指定第一维的长度，又没有给出与行相对应的花括号的数目，因此编译器会根据给出的初始值来判断行数，如果给出的初始值的个数为n，指定的二维数组的第二维的长度为N，那么此时的行数可以通过以下方式得到。

```
if（0==n%N）
```

```
M=n/N;
else
M=n/N+1;
```

通过初始值的个数n与第二维的长度N之间关系就可以知道二维数组的第一维长度M。

4.3　多维数组的定义及引用

讲解完了一维数组和二维数组，接下来看看多维数组，其定义的一般形式为：

```
类型说明符数组名[常量表达式1][常量表达式2][常量表达式3]……;
```

在此以三维数组为例来进行讲解，定义一个三维整型数组int arr[2][3][4]，将其视为由arr[2][3]和int[4]两部分所组成，arr[2][3]为一个二维整型数组，数组中的每个元素都是一个含有4个整型元素的一维数组。按照同样的方法，将arr[2][3]视为由arr[2]和int arr[3]两部分所组成，arr[2]为含有两个元素的一维数组，元素的类型为int[3]，即每个元素都是含有3个整型元素的一维数组，如图4-7所示为arr[2][3][4]的结构。

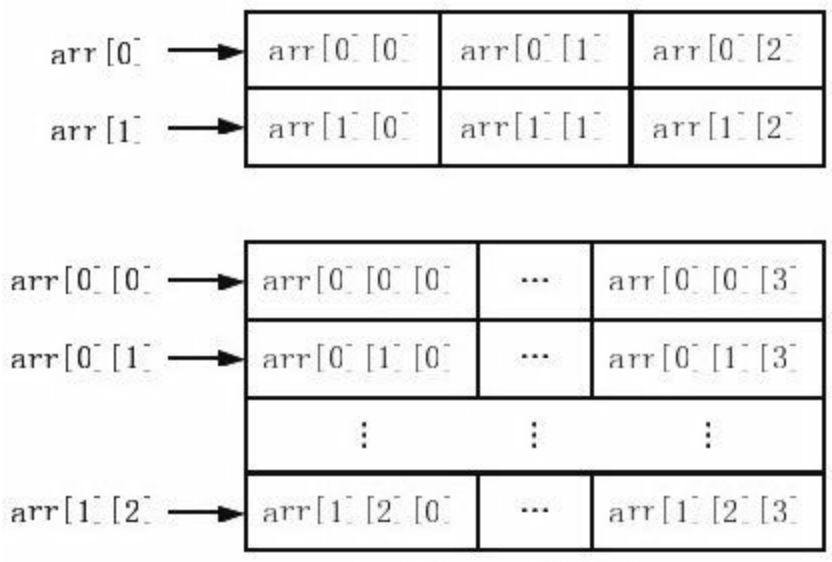

图　4-7　三维数组arr的结构

三维数组中元素的个数为第一维长度×第二维长度×第三维长度，数组元素同样存储在一片连续的内存单元中，存储方式与二维数组非常类似。一维数组的存储是从下标为0的元素开始，按照下标递增的顺序，直到下标为数组长度减1，一维数组的下标能够很好地显示出数组元素在内存中的存放次序。如果将在内存中存储的三维数组视为一个一维数组，那么三维数组元素的存储次序就可以用一维数组的下标来描述，内存中存储的三维数组的任意一个元素arr[m][n][k]可以转换为与之相对应的一维数组元素arr[m*N*K+n*K+k]，其中，N为第二维的长度，K为第三维的长度，而m、n、k分别为第一维、第二维、第三维的下标。三维数组占用的内存大小为数组类型所占用的内存大小乘以它所含元素的个数。三维以上数组的分析与此类似，将其逐层拆分开进行分析即可，在此不再做过多讲解。下面通过一段简单的代码来学习三维数组的定义和引用。

```
#include＜stdio.h＞
#define M 2
#define N 3
#define K 4
void main（void）
{
int i,j，k;
int arr[M][N][K]={1，2，3，4，5，6，7，8，9，10，11，12};
int a[M*N*K];
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
for（k=0；k＜K；k++）
{
printf（"arr[%d][%d][%d]=%d\t"，i,j，k,arr[i][j][k]）；
}
```

```
printf ("\n") ;
}
printf ("\n") ;
}
return;
}
```

运行结果：

```
arr[0][0][0]=1 arr[0][0][1]=2 arr[0][0][2]=3 arr[0][0][3]=4
arr[0][1][0]=5 arr[0][1][1]=6 arr[0][1][2]=7 arr[0][1][3]=8
arr[0][2][0]=9 arr[0][2][1]=10 arr[0][2][2]=11 arr[0][2][3]=12
arr[1][0][0]=0 arr[1][0][1]=0 arr[1][0][2]=0 arr[1][0][3]=0
arr[1][1][0]=0 arr[1][1][1]=0 arr[1][1][2]=0 arr[1][1][3]=0
arr[1][2][0]=0 arr[1][2][1]=0 arr[1][2][2]=0 arr[1][2][3]=0
```

由上面的运行结果可知，三维数组元素的引用与之前讲解的一维数组和二维数组非常类似，都是通过数组名和下标来对数组元素进行操作。三维数组的初始化方式也与二维数组类似，在此就不再详细讲解了。

4.4 字符数组的定义及引用

字符数组的特殊之处在于，它是数组元素为字符的数组。其定义的一般形式和注意事项与之前讲解的一般数组类似，只是其中的类型说明符是char。当然，并不是说类型说明符只能是char，也可以是long、int等，但是由于char型只占用一个字节的大小，使用long型和int型来定义字符数组会造成资源的浪费，因此一般选择使用char型来定义字符数组。

1.一维字符数组

首先通过下面一段代码来看看一维字符数组的定义。

```
    #include<stdio.h>
    #define M 20
    void main(void)
    {
    int i;
    long arr_l[M]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
    char arr_c[M]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
    printf("long型字符数组占用的内存大小为：%d\n", sizeof(arr_l));
    printf("char型字符数组占用的内存大小为：%d\n", sizeof(arr_c));
    return;
    }
```

运行结果：

```
    long型字符数组占用的内存大小为：80
    char型字符数组占用的内存大小为：20
```

在上面的代码中定义了不同类型的字符数组来存放相同的字符，可以看出，它们占用的内存大小相差很大，long型字符数组所占用内存大小是char型数组占用内存大小的4倍。从这点可以看出，选用char型作为数组类型避免了内存空间的浪费。下面通过一段代码来了解字符数组的初始化特点。

```
    #include＜stdio.h＞
    #define M 20
    void main（void）
    {
    int i;
    char arr[M]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
    for（i=0; i＜20; i++）
    printf（"%c", arr[i]）;
    return;
    }
```

运行结果为“Hello World！”，其中有一些空字符。看看上面代码中定义的arr数组，其数组长度为20，而初始化的字符元素的个数为12，初始化的字符元素个数小于数组长度，编译器在编译过程中将后面没有初始化的数组元素赋值为‘\0’，这也正是打印输出中含有空字符的原因。在打印的时候也可以将数组中的元素‘\0’视为数组结束的标志，例如：

```
    #include＜stdio.h＞
    #define M 20
    void main（void）
    {
    int i;
    char arr[M]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
```

```
for (i=0; arr[i]! ='\0'; i++)
printf ("%c", arr[i]) ;
printf ("\n") ;
return;
}
```

运行结果：

```
Hello World!
```

这时的输出结果中就不含有任何空字符了，因为巧妙地使用了字符数组中的‘\0’标志。当然，也可以采用字符串常量的方式来对一维字符数组进行初始化，例如：

```
#include＜stdio.h＞
#define M 20
void main (void)
{
int i;
char arr[M]={"Hello World! "};
for (i=0; arr[i]! ='\0'; i++)
printf ("%c", arr[i]) ;
printf ("\n") ;
return;
}
```

运行结果：

```
Hello World!
```

在对一维字符数组进行定义和初始化的过程中，可以不指定其长度。使用字符常量列表和字符串常量的方式进行初始化的结果是不同

的，例如：

```
    #include＜stdio.h＞
    void main（void）
    {
    int i;
    char arr_s[]={"Hello World! "};
    char arr_c[]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
    printf（"采用字符串常量进行初始化的arr_s数组的长度为：%d\n",
sizeof（arr_s））;
    printf（"采用字符常量列表进行初始化的arr_c数组的长度为：%d\n",
sizeof（arr_c））;
    return;
    }
```

运行结果：

```
    采用字符串常量进行初始化的arr_s数组的长度为：13
    采用字符常量列表进行初始化的arr_c数组的长度为：12
```

从运行结果发现，采用这两种方式得到的数组长度并不相同，在采用字符串常量对字符数组进行初始化的过程中，在内存中进行存储时会自动在字符串的后面添加一个结束符‘\0’，所以得到的字符数组长度是字符串常量的长度加1；而采用字符常量列表的方式对字符数组进行初始化就不会在最后添加一个结束符，所以利用这种方式定义的字符数组的长度就是字符常量列表中字符的个数。

在对字符数组进行初始化时需要注意，不能先定义字符数组再对字符数组进行一次性初始化，例如：

```
char arr[];
arr={'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
arr={"Hello World! "};
```

以上这两种初始化方法都是不正确的，因为arr代表的是数组的首地址，不能将常量的值赋给数组地址。当然，数组之间也不可以进行赋值操作，例如：

```
char arr_a[20]={"safdsafdsa"};
char arr_b[20];
arr_b=arr_a;
```

这种赋值方式同样是错误的，因为两个数组名都是一个常量地址，不可以对其进行任何修改，先来看下面这段代码。

```
#include<stdio.h>
void main（void）
{
char arr[]={"Hello World! "};
printf（"字符数组的首地址为：%d\n"，&arr[0]）；
printf（"字符数组的首地址为：%d\n"，arr）；
return;
}
```

运行结果：

```
字符数组的首地址为：1245036
字符数组的首地址为：1245036
```

由上面的运行结果发现，数组名就是数组元素的首地址，所以不能够对其进行前面那些错误操作。

虽然&arr[0]和arr的值相同，但是所指的内容并不相同，可以通过下面的代码来进一步加深印象。

```
#include<stdio.h>
void main(void)
{
char arr[]={"Hello World! "};
printf("&arr[0]占用内存大小：%d\n", sizeof(&arr[0]));
printf("arr占用内存大小为：%d\n", sizeof(arr));
return;
}
```

运行结果：

```
&arr[0]占用内存大小：4
arr占用内存大小为：13
```

从运行结果可以清楚地知道，&arr[0]和arr所占用的内存大小并不相同，&arr[0]代表一个地址变量，由于在32位计算机中地址变量是由4字节大小来表示的，因此&arr[0]的大小为4字节，而arr的大小却是13字节，这是因为它代表的是整个数组，在采用字符串常量进行初始化时会在字符串常量的后面添加一个串结束符‘\0’，所以相当于定义一个数组长度为13的字符数组char arr[13]，进而可以将arr视为int[13]这种特殊的类型，所以它占用内存是13字节的大小。

接下来通过下面的代码来看串结束符在字符数组输出中的作用。

```
#include<stdio.h>
void main(void)
{
```

```
    char arr[]={"Hello World! "};
    printf ("%s\n", arr) ;
    return;
    }
```

运行结果：

```
    Hello World!
```

修改一下代码，然后再看看运行结果。

```
    #include<stdio.h>
    void main (void)
    {
    char arr[]=
{'H', 'e', 'l', 'l', 'o', '', 'W', 'o', 'r', 'l', 'd', '! '};
    printf ("%s\n", arr) ;
    return;
    }
```

运行结果：

```
    Hello World! ?↕
```

由运行结果发现：在输出的字符串中出现了初始化时没有的字符。这是因为在使用%s格式输出的时候，如果想要得到正确的结果，那么字符必须以结束符‘\0’结尾。当然也可以采用%c格式分别输出字符数组中的每个元素。

2.二维字符数组

接下来看二维字符数组，其定义的一般形式为：

```
char 数组名[常量表达式1][常量表达式2];
```

以char str[M][N]为例，可以将其视为由两部分组成，str[M]和char[N]，相当于定义了一个数组长度为M的一维数组，其中的每个元素又是含有N个字符长度的一维数组。对于多维字符数组，可以采用前面讲解的二维数组的方法来进行分析。接下来了解二维字符数组的使用，下面代码实现的是通过基姆拉尔森计算公式求解输入的某天是星期几。

```
#include＜stdio.h＞
int main（）
{
char weekname[][10]={
"Monday",
"Tuseday",
"Wednesday",
"Thursday",
"Friday",
"Saturday",
"Sunday",
};
int year;
int month;
int day;
printf（"请输入年份："）；
scanf（"%d", ＆year）；
printf（"请输入月份："）；
scanf（"%d", ＆month）；
printf（"请输入日期："）；
scanf（"%d", ＆day）；
if（（month==1）||（month==2））
{
month+=12;
year--;
}
int index;
index=（day+2*month+3*（month+1）/5+year+year/4-
```

```
year/100+year/400）%7;
    printf（"%d/%d/%d这一天是%s\n"，year,month,day,weekname[index]）;
    return 0;
    }
```

运行结果：

```
请输入年份：2011
请输入月份：8
请输入日期：24
2011/8/24这一天是Wednesday
```

上面定义的二维数组没有指定第一维的长度，这时该长度由初始化的字符串常量的个数来决定。通过数组下标index来决定当前日期是星期几，然后用weekname[index]来输出。值得注意的是，weekname[index]是一个一维字符数组的首地址，其字符数组的长度为10。如图4-8所示为二维数组weekname在内存中的存储方式。

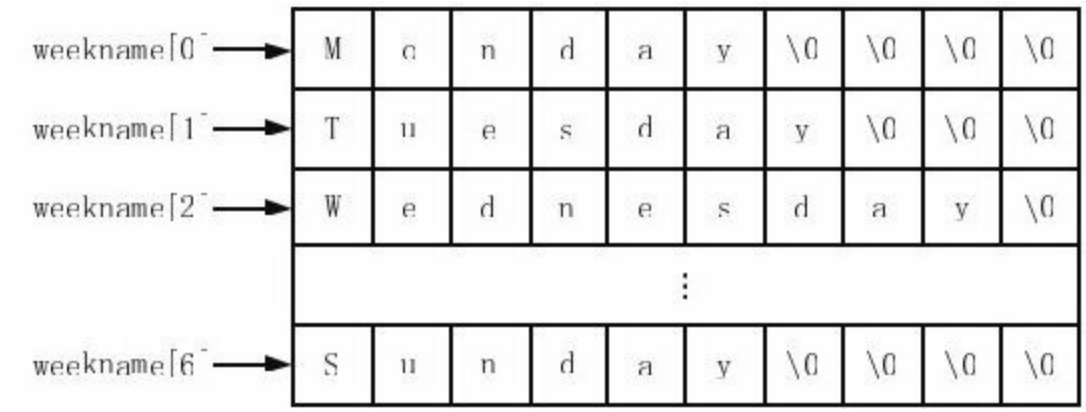

图　4-8　二维数组weekname在内存中的存储方式

由二维数组weekname在内存中的存储方式可知，没有指定初始化值的数组元素被赋值为串结束符‘\0’，在每个字符串常量的后面都有结

束符，所以可以使用%s格式来进行输出。

4.5 数组作为函数参数的易错点解析

对于函数调用过程中数组的使用，最常用的方式是把数组作为参数进行传递。下面通过一段代码来了解数组在函数调用中的使用，此段代码的功能为产生8个0～99之间的随机整数，这里采用冒泡法升级排序输出。

```
#include<stdio.h>
#include<stdlib.h>
#define N 8
void sort(int arr[], int n)
{
int i,j;
int temp;
int flag;
for(i=0; i<n-1; i++)
{
flag=1;
for(j=0; j<n-i-1; j++)
{
if(arr[j]>arr[j+1])
{
temp=arr[j];
arr[j]=arr[j+1];
arr[j+1]=temp;
flag=0;
}
}
}
for(i=0; i<n; i++)
{
printf("%d\t", arr[i]);
}
return;
}
int main(void)
{
int i;
int arr[N];
```

```
printf（"排序前：\n"）；
for（i=0；i＜N；i++）
{
arr[i]=rand（）%100；
printf（"%d\t"，arr[i]）；
}
printf（"\n排序后：\n"）；
sort（arr,N）；
return 0；
}
```

运行结果：

排序前：

```
41 67 34 0 69 24 78 58
```

排序后：

```
0 24 34 41 58 67 69 78
```

在sort（）函数中，我们将main（）函数中随机生成的数组通过参数传递给sort（）函数，然后在sort（）函数中对传递过去的数组元素进行排序。但是在使用数组名作为参数进行传递时需要注意，传递的是数组的首地址，即在函数sort（）中操作的数组与在main（）函数中定义的数组共享一个存储空间，所以在sort（）函数中对arr数组元素进行排序也就是对在main（）函数中定义的数组元素进行排序，所以也可以在main（）函数中执行打印arr数组元素的操作，得到的同样是进行排序之后的数组，例如：

```
#include<stdio.h>
#include<stdlib.h>
#define N 8
void sort(int arr[], int n)
{
int i,j;
int temp;
int flag;
for(i=0; i<n-1; i++)
{
flag=1;
for(j=0; j<n-i-1; j++)
{
if(arr[j]>arr[j+1])
{
temp=arr[j];
arr[j]=arr[j+1];
arr[j+1]=temp;
flag=0;
}
}
}
return;
}
void print(int arr[], int n)
{
int i;
for(i=0; i<n; i++)
{
printf("%d\t", arr[i]);
}
}
int main(void)
{
int i;
int arr[N];
printf("排序前: \n");
for(i=0; i<N; i++)
{
arr[i]=rand()%100;
printf("%d\t", arr[i]);
}
sort(arr,N);
printf("\n排序后: \n");
print(arr,N);
return 0;
}
```

运行结果：

```
排序前：
41 67 34 0 69 24 78 58
排序后：
0 24 34 41 58 67 69 78
```

从运行结果可以发现，这样的操作能够返回想要的结果。但是如果在一个函数中生成了一个数组，那么这个数组能否在其他函数中使用呢？看看下面的代码。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define N 8
int*creat ()
{
int i;
printf ("creat () 函数中产生数组元素\n") ;
int arr[N];
for (i=0; i＜N; i++)
{
arr[i]=rand () %100;
printf ("%d\t", arr[i]) ;
}
return arr;
}
void print (int arr[], int n)
{
int i;
for (i=0; i＜n; i++)
{
printf ("%d\t", arr[i]) ;
}
}
int main (void)
{
int i;
int*p;
p=creat () ;
```

```
printf（"\n通过函数调用返回后得到的数组\n"）；
print（p,N）；
return 0;
}
```

运行结果：

```
creat（）函数中产生数组元素
41 67 34 0 69 24 78 58
通过函数调用返回后得到的数组
-858993460-858993460-858993460-858993460-858993460
-858993460 6 1245056
```

上面的运行结果表明，函数并没有成功地返回想要的数组元素，这是因为在creat（）函数中定义的数组arr在creat（）函数调用结束后被自动释放掉了，所以不可能得到正确的结果。正如前面讲解的，数组arr的生存周期随着函数调用的结束而结束了，那么有什么办法能够成功地从数组中返回一个数组呢？当然是想办法使数组在函数调用结束之后不被释放。具体实现方法有以下两种。

将数组定义为静态数组。静态数组的定义与普通数组的定义的区别就在于要在数组的前面加上关键字static，例如：

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define N 8
int*creat（）
{
int i;
printf（"creat（）函数中产生数组元素\n"）；
static int arr[N];
for（i=0; i＜N; i++）
{
```

```
arr[i]=rand（）%100;
printf（"%d\t", arr[i]）;
}
return arr;
}
void print（int arr[], int n）
{
int i;
for（i=0; i＜n; i++）
{
printf（"%d\t", arr[i]）;
}
}
int main（void）
{
int i;
int*p;
p=creat（）;
printf（"\n通过函数调用返回后得到的数组\n"）;
print（p,N）;
return 0;
}
```

运行结果：

```
creat（）函数中产生数组元素
41 67 34 0 69 24 78 58
通过函数调用返回后得到的数组
41 67 34 0 69 24 78 58
```

定义为静态数组后发现成功返回了数组，这是因为静态数组的生存周期是从程序开始运行到程序运行结束，而不像普通数组那样随着函数调用的结束而结束其生命周期，所以数组不会随着函数调用的结束而被释放掉。

申请一个动态数组。对动态数组不了解的读者，可以先学习本书第

4.6节的知识点，下面学习具体实现方法。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define N 8
int*creat ()
{
int i;
printf ("creat () 函数中产生数组元素\n") ;
int*arr= (int*) malloc (sizeof (int*) *N) ;
for (i=0; i＜N; i++)
{
arr[i]=rand () %100;
printf ("%d\t", arr[i]) ;
}
return arr;
}
void sort (int arr[], int n)
{
int i,j;
int temp;
int flag;
for (i=0; i＜n-1; i++)
{
flag=1;
for (j=0; j＜n-i-1; j++)
{
if (arr[j]＞arr[j+1])
{
temp=arr[j];
arr[j]=arr[j+1];
arr[j+1]=temp;
flag=0;
}
}
}
return;
}
void print (int arr[], int n)
{
int i;
for (i=0; i＜n; i++)
{
printf ("%d\t", arr[i]) ;
}
```

```
}
int main（void）
{
int i;
int*p;
p=creat（）；
sort（p,N）；
printf（"\n通过函数调用返回进行排序后的数组\n"）；
print（p,N）；
free（p）；
return 0;
}
```

运行结果：

```
creat（）函数中产生数组元素
41 67 34 0 69 24 78 58
通过函数调用返回后得到的数组
41 67 34 0 69 24 78 58
```

采用这种方式同样可以得到正确的结果，只是，这个时候要注意在main（）函数中采用free（）函数进行内存释放时，不能使用arr作为参数，因为尽管为它申请的内存空间是从程序运行开始到程序运行结束，但是它并不是一个全局变量。在main（）函数中使用指针p指向了内存单元的首地址，所以可以采用指针变量p作为free（）函数的参数对申请的内存空间进行释放。

4.6 动态数组的创建及引用

在讲解动态数组之前，先来看下面这段代码，其功能为将一幅像素为320×320的灰度图像扩展为480×480，扩展方式为：每隔四行空两行，每隔两列空两列。出于代码篇幅的考虑，在此主要给出算法的核心部分。

```
#include<stdio.h>
#include<stdlib.h>
#define N 320
#define M 480
int main(void)
{
int i,j;
int a[N][N];
int b[M][M];
for(i=0; i<N; i++)
{
for(j=0; j<N; j++)
{
a[i][j]=rand() %256;
printf("%d\t", a[i][j]);
}
printf("\n");
}
for(i=0; i<M; i++)
{
for(j=0; j<M; j++)
{
if(4==(i%6)||5==(i%6)||4==(j%6)||5==(j%6))
{
b[i][j]=0;
continue;
}
b[i][j]=a[i/6*4+i%6][j/6*4+j%6];
}
}
return 0;
}
```

由于运行结果篇幅过大，在此就不再给出运行结果，有兴趣的读者可以按照比例缩小M和N的值，然后查看运行结果会发现，在编译的时候不会有任何错误，而运行程序时会直接崩溃掉，调试一下会发现“stack overflow”错误，这说明栈溢出了。我们发现这个时候使用静态数组没法解决问题，那么该如何解决这个问题呢？这就需要使用接下来要讲解的动态数组，尤其是在进行图像处理的时候，必须用动态数组来对大量数据进行处理。

先来看什么是动态数组。动态数组是相对于静态数组而言的，从“动”字可以看出它的灵活性。静态数组的长度是预先定义好的，在整个程序中，一旦给定数组大小后就无法改变。而动态数组则不然，它可以根据程序需要重新指定数组大小。动态数组的内存空间是由堆动态分配的，通过执行代码为其分配存储空间，只有程序执行到分配语句时，才为其分配存储空间。

对于静态数组，其创建非常方便，使用完也无需释放，引用起来也简单，但是创建后无法改变数组大小是其致命的弱点。对于动态数组，其创建麻烦，使用完必须由程序员自己释放，否则会引起内存泄露，但是其使用非常灵活，能根据程序需要动态分配大小。因此相对于静态数组来说，使用动态数组的自由度更大。

在创建动态数组的过程中要遵循一个原则，那就是从外层向里层逐

层创建，从里层向外层逐层释放。如果要创建一个N维的动态数组，那么要从第一维开始创建，直到第N维为止；而释放时与创建时相反，即从第N维开始释放，直到第一维为止。下面结合代码来讲解。1.一维动态数组创建一维动态数组的一般形式为：

```
类型说明符*数组名=（类型说明符*）malloc（数组长度*sizeof（类型说明符））；
```

下面通过一段代码来了解一维动态数组的使用方法。

```
#include＜stdio.h＞
#include＜stdlib.h＞
int main（）
{
int n,i;
int*arr;
printf（"请输入所要创建的一维动态数组的长度："）；
scanf（"%d", &n）；
if（（arr=（int*）malloc（sizeof（int）*n））==NULL）
{
printf（"分配内存空间失败！\n"）；
return 0;
}
for（i=0; i＜n; i++）
{
arr[i]=i+1;
printf（"%d\t", arr[i]）；
if（0==（i+1）%4）
printf（"\n"）；
}
free（arr）；
return 0;
}
```

运行结果：

```
请输入所要创建的一维动态数组的长度：12
```

```
1 2 3 4
5 6 7 8
9 10 11 12
```

在以上动态数组的创建过程中，先使用了malloc（）函数向系统动态申请分配了sizeof（int）*n个字节的内存空间，然后将申请的内存空间视为一个一维数组进行操作。当然，一维数组没有体现出动态数组的分配原则。

2.二维动态数组

创建二维动态数组的一般形式为：

```
类型说明符**数组名=（类型说明符**）malloc（第一维长度*sizeof（类型说明符*））；
```

例如：

```
for（i=0；i＜第一维长度；i++）
{
数组名[i]=（类型说明符**）malloc（第二维长度*sizeof（类型说明符*））；
}
```

接下来通过一段代码来了解二维动态数组的创建方法。

```
#include＜stdio.h＞
#include＜stdlib.h＞
int main（）
{
int n1，n2；
int**arr,i，j；
printf（"请输入所要创建的动态数组的第一维长度："）；
```

```
scanf（"%d", &n1）;
printf（"请输入所要创建的动态数组的第二维长度："）;
scanf（"%d", &n2）;
if（（arr=（int**）malloc（n1*sizeof（int*）））==NULL）//第一维的创建
{
printf（"分配内存空间失败！\n"）;
return 0;
}
for（i=0; i＜n1; i++）
{
if（（arr[i]=（int*）malloc（n2*sizeof（int）））==NULL）//第二维的创建
{
printf（"分配内存空间失败！\n"）;
return 0;
}
}
for（i=0; i＜n1; i++）
{
for（j=0; j＜n2; j++）
{
arr[i][j]=i*n2+j+1;
printf（"%d\t", arr[i][j]）;
}
printf（"\n"）;
}
for（i=0; i＜n1; i++）
{
free（arr[i]）; //释放第二维
}
free（arr）; //释放第一维
return 0;
}
```

运行结果：

```
请输入所要创建的动态数组的第一维长度：4
请输入所要创建的动态数组的第二维长度：3
1  2  3
4  5  6
7  8  9
10  11  12
```

以上动态数组的创建过程直观地体现出创建时所要遵循的基本原则，先通过下面一行代码来创建最外层，即第一维。

```
array=（int**）malloc（n1*sizeof（int*））；//第一维的创建
```

接下来创建第二维，在创建第二维的过程中需要注意，这里使用了一个for循环语句来进行创建，要创建的二维数组为arr[n1][n2]，第一维的大小为n1，第二维的大小为n2，利用前面分析静态数组时的方法将其拆分为arr[n1]和int[n2]两部分，定义第一维的大小为n1，其中每个元素arr[i]都指向含有n2个int型数组元素的内存区域，即指向一片内存大小为n2*sizeof（int）字节的空间，所以在创建第二维的时候要采用下面的方法来实现。

```
for（i=0；i＜n1；i++）
{
arr[i]=（int*）malloc（n2*sizeof（int））；//第二维的创建
}
```

创建好数组元素之后，接下来对其中的数组元素进行赋值操作，并打印输出。创建动态数组之后千万别忘了释放内存，否则会导致内存泄露，释放时要遵循从内向外的原则，即先释放最高维，然后是次高维，直到第一维为止。这里创建的是二维数组，所以先释放的是第二维。与之前讲的创建方法类似，同样采用下面的循环来释放。

```
for（i=0；i＜n1；i++）
{
```

```
free（arr[i]）；//释放第二维
}
```

释放完第二维后采用下面的方式释放第一维。

```
free（arr）；//释放第一维
```

所以，对动态数组的使用要有始有终，要牢记在使用完之后要及时释放申请的内存区域，避免造成内存泄露。经过上面的讲解，相信读者对于动态数组的创建规则有了一定的了解。

3.三维动态数组

创建三维动态数组的一般形式为：

类型说明符***数组名=（类型说明符***）malloc（第一维长度*sizeof（类型说明符**））；

例如：

```
for（i=0；i＜第二维长度；i++）
{
数组名[i]=（类型说明符**）malloc（第二维长度*sizeof（类型说明符*））；
for（j=0；j＜第三维长度；j++）
{
数组名[i][j]=（类型说明符*）malloc（第三维长度*sizeof（类型说明符））；
}
}
```

从上面的三维动态数组可以看出，虽然三维动态数组的创建可以由

二维动态数组推导出来，但是与二维数组的创建相比还是要复杂一些，所以在此特地给出三维动态数组创建方法。接下来通过具体的实例来了解三维动态数组的创建，以进一步加深读者对动态数组的理解。

```
    #include＜stdlib.h＞
    #include＜stdio.h＞
    int main（）
    {
    int n1，n2，n3；
    int***arr；
    int i,j，k；
    printf（"请输入所要创建的动态数组的第一维长度："）；
    scanf（"%d"，＆n1）；
    printf（"请输入所要创建的动态数组的第二维长度："）；
    scanf（"%d"，＆n2）；
    printf（"请输入所要创建的动态数组的第三维长度："）；
    scanf（"%d"，＆n3）；
    if（（arr=（int***）malloc（n1*sizeof（int**）））==NULL）//第一维的创
建
    {
    printf（"分配内存空间失败！\n"）；
    return 0；
    }
    for（i=0；i＜n1；i++）
    {
    if（（arr[i]=（int**）malloc（n2*sizeof（int*）））==NULL）//第二维的创
建
    {
    printf（"分配内存空间失败！\n"）；
    return 0；
    }
    for（j=0；j＜n2；j++）
    {
    if（（arr[i][j]=（int*）malloc（n3*sizeof（int）））==NULL）//第三维的
创建
    {
    printf（"分配内存空间失败！\n"）；
    return 0；
    }
    }
    }
    for（i=0；i＜n1；i++）
    {
```

```
for（j=0; j＜n2; j++）
{
for（k=0; k＜n3; k++）
{
arr[i][j][k]=i+j+k+1;
printf（"%d\t", arr[i][j][k]）;
}
printf（"\n"）;
}
printf（"\n"）;
}
for（i=0; i＜n1; i++）
{
for（j=0; j＜n2; j++）
{
free（arr[i][j]）; //释放第三维
}
}
for（i=0; i＜n1; i++）
{
free（arr[i]）; //释放第二维
}
free（arr）; //释放第一维
return 0;
}
```

运行结果：

```
请输入所要创建的动态数组的第一维长度：3
请输入所要创建的动态数组的第二维长度：3
请输入所要创建的动态数组的第三维长度：3
1 2 3
2 3 4
3 4 5
2 3 4
3 4 5
4 5 6
3 4 5
4 5 6
5 6 7
```

从上面的代码可以看出，三维动态数组的创建方法与前面讲解的二

维动态数组的创建方法类似，在此就不再展开分析了。

4.可扩展动态数组

细心的读者可能发现了一个问题，那就是前面所讲的动态数组都是一次性创建好的，如果在接下来使用数组时需要扩展或删减一些不再使用的数组元素，该怎么办呢？这个时候就要想办法扩展动态数组，这就是接下来要讲解的——可扩展动态数组。在讲解可扩展动态数组之前，先看一段关于动态数组扩展的代码，在此以一维动态数组的扩展为例，其他类型的数组以此类推。

```
#include＜stdio.h＞
#include＜stdlib.h＞
int main（）
{
int*n，*p;
int i,n1，n2;
printf（"请输入所要创建的动态数组的长度："）；
scanf（"%d"，＆n1）；
if（（n=（int*）malloc（n1*sizeof（int）））==NULL）
{
printf（"分配内存空间失败！\n"）；
return 0;
}
for（i=0; i＜n1; i++）
{
n[i]=n1-i;
printf（"%d\t"，n[i]）；
if（0==（i+1）%4）
printf（"\n"）；
}
printf（"\n请输入所要扩展的动态数组的长度："）；
scanf（"%d"，＆n2）；
if（（p=（int*）realloc（n，（n2）*sizeof（int））==NULL）//动态扩充数组
{
printf（"分配内存空间失败！\n"）；
```

```
return 0;
}
for（i=0; i＜n2; i++）
{
p[i]=n2-i;
printf（"%d\t", p[i]）;
if（0==（i+1）%4）
printf（"\n"）;
}
free（p）;
return 0;
}
```

运行结果：

```
请输入所要创建的动态数组的长度：8
8 7 6 5
4 3 2 1
请输入所要扩展的动态数组的长度：12
12 11 10 9
8 7 6 5
4 3 2 1
```

在上面的代码中，一开始分配的数组长度是n1，接下来把数组长度扩展为n2。这时要注意，有的读者可能会采用如下方式来释放定义的动态数组：

```
free（p）;
free（n）;
```

因为在代码中定义了两个整型指针n和p，所以释放的时候需要分别对n和p进行内存分配，所以有的读者认为应该采用这种释放方式。事实上，这是不正确的，至于具体的错误原因在本书10.7节关于

malloc（）、calloc（）、realloc（）三者区别的讲解部分有详细分析，不明白的读者一看便知其中缘由了。

上面讲解了动态数组的扩展，而动态数组的缩减该如何操作呢？只要在运行上面的代码时输入的n2小于n1，就可以实现对动态数组的缩减。在此就不再给出代码了，读者可以在运行程序时通过控制n1和n2的值来实现对动态数组的扩展和缩减。

第5章　指针

懂得C语言的人都知道，C语言的强大和良好的自由性绝大部分体现在指针的灵活运用上。因此，说指针是C语言的灵魂，一点都不为过。指针又是C语言学习中的一个难点，有不少人始终对指针怀有一种畏惧心理，如何来理解指针，指针到底意味着什么，都将在本章得到答案，让我们一起揭开C语言指针的神秘面纱。

5.1 不同类型指针之间的区别和联系

在本书的第1章中就对指针做了初步介绍，同时也交代了不可能用一个小节的内容来讲解指针，因此这里特地用一章的篇幅讲解指针。

前面已初步介绍了不同类型指针之间的区别，但是并没有进行详细分析，那么不同类型的指针之间究竟有什么样的区别和联系呢？

1.不同类型的指针

在讲解不同类型指针的区别和联系之前先来看下面的一段代码，然后对其区别和联系加以总结。

```
    #include<stdio.h>
    int main(void)
    {
    char a=1, *pa;
    int b=2, *pb;
    double c=3, *pc;
    pa=&a;
    pb=&b; pc=&c;
    printf("char型指针pa占用内存大小为：%d字节\n", sizeof(pa));
    printf("int型指针pb占用内存大小为：%d字节\n", sizeof(pb));
    printf("double型指针pc占用内存大小为：%d字节\n\n", sizeof(pc));
    printf("char型指针pa所指向内存区域的大小为：%d字节\n", sizeof(*pa));
    printf("int型指针pb所指向内存区域的大小为：%d字节\n", sizeof(*pb));
    printf("double型指针pc所指向内存区域的大小为：%d字节\n\n",
sizeof(*pc));
    printf("char型变量a的地址为：%d\n", pa);
    printf("int型变量b的地址为：%d\n", pb);
    printf("double型变量c的地址为：%d\n", pc);
    return 0;
    }
```

运行结果：

```
char型指针pa占用内存大小为：4字节
int型指针pb占用内存大小为：4字节
double型指针pc占用内存大小为：4字节
char型指针pa所指向内存区域的大小为：1字节
int型指针pb所指向内存区域的大小为：4字节
double型指针pc所指向内存区域的大小为：8字节
char型变量a的地址为：1245052
int型变量b的地址为：1245044
double型变量c的地址为：1245032
```

从上面的运行结果可以看出，虽然定义了不同类型的指针，但是它们在内存中都占有4字节的大小，这也进一步验证了前面所讲的内容，指针变量占用内存的大小与它本身的类型无关，而是由使用的计算机决定的。但是不同类型的指针之间也是有区别的，因为不同类型的指针变量所指向内存区域的大小并不相同，可以通过图5-1来了解不同类型的指针在内存中的结构。

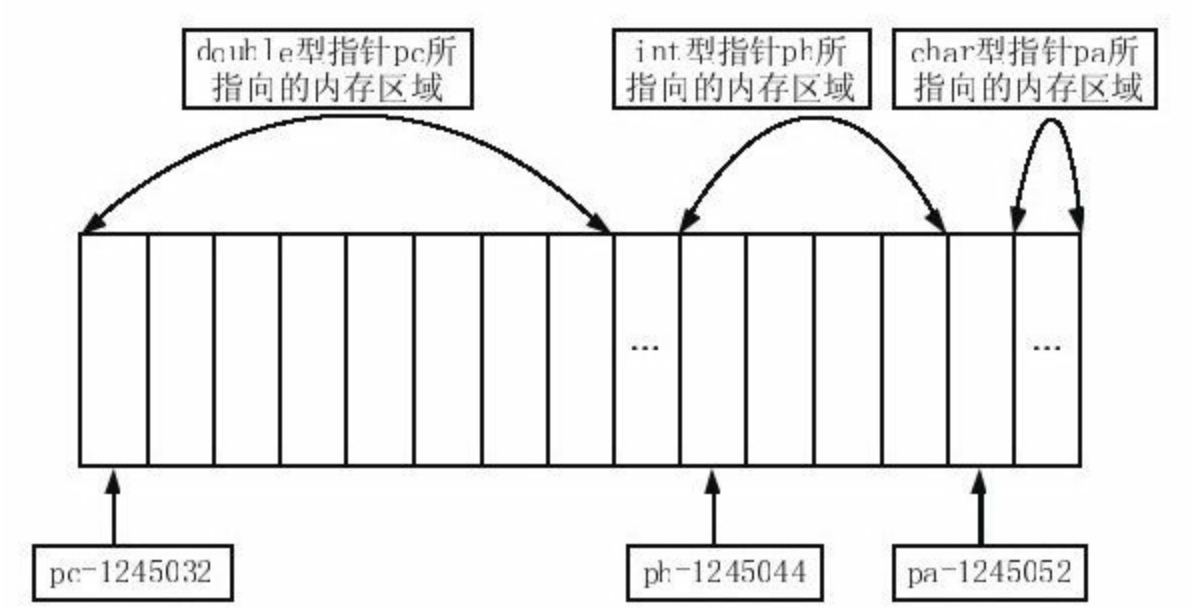

图　5-1　不同类型指针的内存结构

结合上面的运行结果和图5-1可以清楚地发现，不同类型的指针在内存中所指向的内存区域的大小并不相同。在第1章中也强调过，不能把指针简单地理解为地址，虽然时常听到“指针就是地址，地址就是指针”这样的说法，但是我们应该将指针与所指向的内存区域结合起来，这样就可以对指针有更加深入的认识，在编程的时候才能够对指针的运用做到心中有数。

2.普通指针和数组指针

普通指针和数组指针之间又有什么样的区别和联系呢？在给出区别之前，先一起来看看下面的一段代码。

```
#include＜stdio.h＞
int main（void）
{
char a[4];
char（*pa）[4]，*pb;
pa=&a;
pb=&a[0];
printf（"char型数组指针pa做占用的内存大小为：%d\n"，sizeof（*pa））；
printf（"char型指针pb做占用的内存大小为：%d\n\n"，sizeof（*pb））；
printf（"pa=%d\tpa+1=%d\n"，pa,pa+1）；
printf（"pb=%d\tpb+1=%d\n"，pb,pb+1）；
return 0;
}
```

运行结果：

```
char型数组指针pa做占用的内存大小为：4
```

```
char型指针pb做占用的内存大小为：1
pa=1245052 pa+1=1245056
pb=1245052 pb+1=1245053
```

在上面的代码中，＆a和＆a[0]都表示char型数组a的首地址，但是它们的类型并不相同，这是因为＆a[0]仅表示数组中一个char型变量的地址，与一个普通的char型变量无异。值得注意的是＆a，在此之前也对其进行过相应的分析，先将char a[4]变形为char*（＆a）[4]，这时会清楚地发现＆a并不是一个简单的char型指针，而是一个char*[4]型的指针，所以接下来定义了一个数组指针char（*pa）[4]来存放＆a，如果不定义数组指针char（*pa）[4]，而定义一个char*pa，使用pa=＆a就会出现“error C2440：'='：cannot convert from'char（*）[4]'to'char*'”错误，这是因为两者的类型不匹配。如图5-2所示为指针pa、pa+1、pb、pb+1的内存结构。

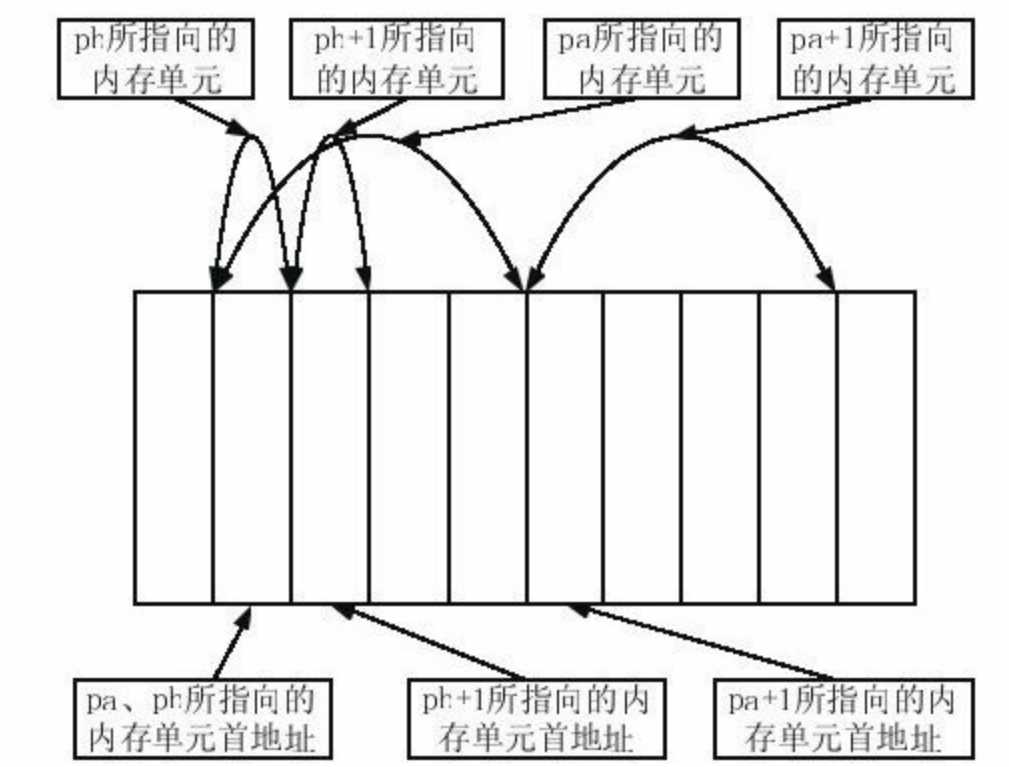

图　5-2　指针pa、pa+1、pb、pb+1的内存结构

通过上面的运行结果和图5-2发现，数组指针pa和pb所指向的是同一个起始地址，但是由于指针的类型不同，所指向的内存单元的大小也不一样，因此在进行指针运算时得到的结果也不相同。pb到pb+1的变化大小由它所指向的类型决定，由于指针是字符指针，所以变化为一个字节，如果是整型指针，那么变化是4字节；而pa到pa+1的变化大小为4字节，正好为pa指针所指向的类型char*[4]，所以在做相应的指针运算时尤其要注意指针所指向的类型。

3.不同类型指针间的强制转换

如果强制转化指针的类型，那会出现什么情况呢？先一起来看看下

面的一段代码。

```
#include＜stdio.h＞
int main（void）
{
int a;
int*pa;
pa=＆a;
a=0x12345678;
printf（"int型指针pa的值为：%x\n"，pa）；
printf（"char型指针pa的值为：%x\n\n"，（char*）pa）；
printf（"int型指针pa所指内存单元的值为：%x\n"，*pa）；
printf（"char型指针pa所指内存单元的值为：%x\n"，*（char*）pa）；
return 0;
}
```

运行结果：

```
int型指针pa的值为：12ff7c
char型指针pa的值为：12ff7c
int型指针pa所指内存单元的值为：12345678
char型指针pa所指内存单元的值为：78
```

从上面的运行结果可以看出。int型指针变量pa中存放变量a的地址，通过（char*）pa将int型指针变量强制转换为char型指针变量，指针的值并没有发生变化，因为任何指针在32位计算机中都用4个字节来表示，所以指针值不会随指针类型的变化而改变。值得注意的是，强制转换后指针所指的内存单元发生了变化，之前pa是整型指针，占用4字节的大小，转换之后变为char型指针，占用1字节的大小，所以接下来采用printf（）函数打印出pa指针变化前后所指内存单元的内容时得到的结果不同，如图5-3所示。

通过图5-3可以更加直观地了解指针转换前后pa指针所指向的内存单元的大小。

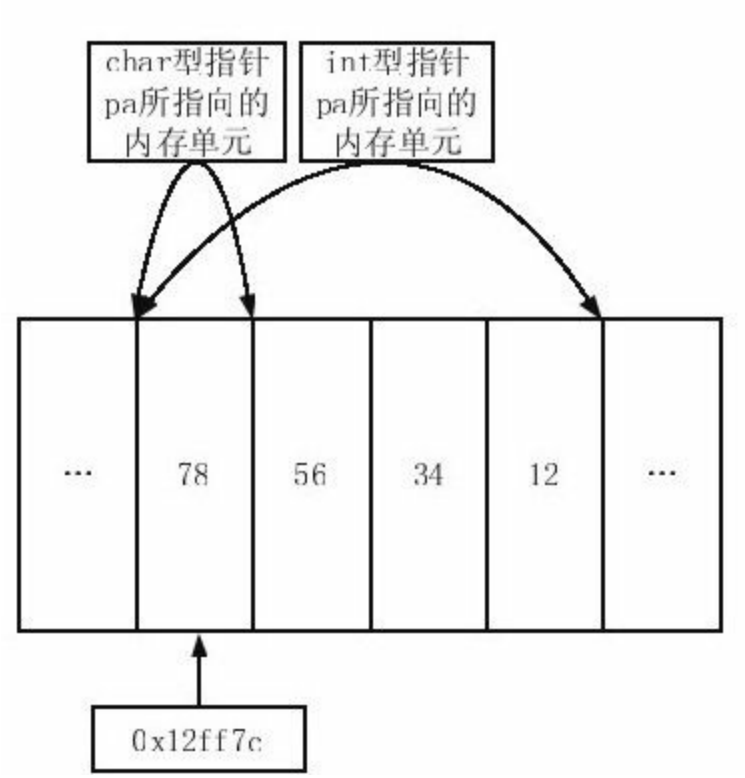

图　5-3　char型指针pa和int指针pa的内存结构

这里要插入一个知识点，那就是计算机中的大小端模式。所谓小端模式，是数据的低位保存在内存的低地址中，高位保存在内存的高地址中；大端模式就是数据的低位保存在内存的高地址中，而高位保存在内存的低地址中。

很多人对于大小端模式不是太在意，因为大多数情况下在计算机上编程时好像不需要了解它。但是现在很多芯片，如ARM、DSP等，在使用的过程中要尤其注意是大端模式还是小端模式，这直接影响编写代码

的方式。如果计算机是大端模式，而我们按照小端模式的方法来编写，那么对于数据的处理就不会按照预期的方式进行，最终得到的也不是想要的结果。在此讲解一下如何判断使用的计算机是小端模式还是大端模式。

```
#include＜stdio.h＞
int main（void）
{
int a;
int*pa;
pa=&a;
a=0x11223344;
if（0x44==*（char*）pa）
printf（"小端模式！\n"）；
else
printf（"大端模式！\n"）；
return 0;
}
```

运行结果：

```
小端模式！
```

在上面的代码中，先定义了一个整型变量a，其初始值为0x11223344，同时定义了一个整型指针指向变量a的内存地址，然后通过（char*）pa将整型指针转换为字符指针，如果转换后的字符指针的值为0x44，那么计算机为小端模式，否则为大端模式。这样判断的依据是指针指向的内存单元的起始地址，即低地址，如果转换为字符指针后，其存储单元的值为0x44，那么说明低地址部分保存的是低位，即为小端

模式，否则为大端模式。

5.2 指针的一般性用法及注意事项

在学习指针的一般性用法时，读者首先要清楚地知道指针是地址，而这个地址的特殊性在于指向的是一片连续内存区域的起始地址。下面通过代码来看指针的一般性用法及注意事项。

```
#include<stdio.h>
void main ()
{
int a,b;
int*pa, *pb;
a=100;
b=200;
pa=&a;
pb=&b;
printf ("--------------------变换前-------------------\n") ;
printf ("a=%d\tb=%d\n", a,b) ;
printf ("*pa=%d\t*pb=%d\n", *pa, *pb) ;
*pa=300;
int c=500;
pb=&c;
printf ("--------------------变换后-------------------\n") ;
printf ("a=%d\t*pa=%d\n", a, *pa) ;
printf ("c=%d\tb=%d\t*pb=%d\n", c,b, *pb) ;
return;
}
```

运行结果：

```
--------------------变换前-------------------
a=100 b=200
*pa=100*pb=200
--------------------变换后-------------------
a=300*pa=300
c=500 b=200*pb=500
```

上面的代码可以通过图5-4来展示变换前后各变量的值。在一开始时定义了两个整型变量a和b，然后定义了两个整型指针，分别指向变量a和b的内存单元。所以接下来直接打印出的a和b与采用*pa和*pb打印出来的结果完全一样。但是，在接下来的代码中通过“*pa=300；”来对pa所指向的内存单元进行赋值，而pa所指向的内存单元也就是变量a所指向的内存单元，所以在接下来打印出的a的值不再是初始值了，而是后来对*pa赋的值。定义了一个整型变量c，然后将pb指向c在内存中的存储单元，而不是变量b在内存中的存储单元，所以接下来输出的变量b的值仍是之前对b的赋值，而指针pb所指向的内存单元的值变为此时它所指向的变量c在内存单元中的值。

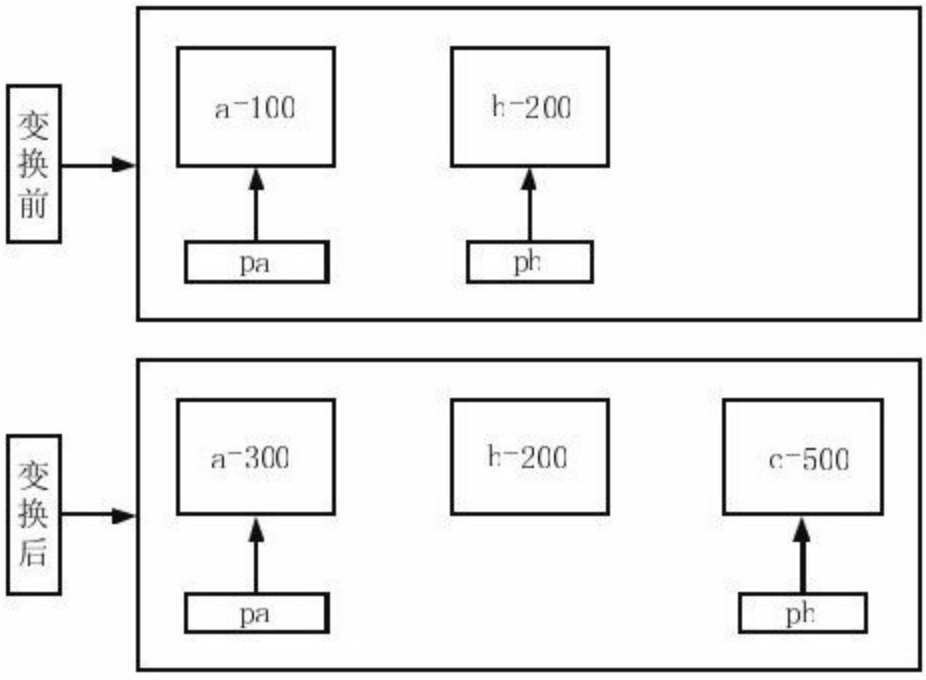

图　5-4　变换前后的内存结构图

读者要注意上面的代码中的“*pa=300；”，在这里可以使用这句代码的原因是在此之前通过pa=＆a使pa指向变量a在内存中的存储单元。如果定义了一个指针并直接使用，会发生什么现象呢？看看下面的代码。

```
#include＜stdio.h＞
void main ()
{
int*pa;
*pa=10;
return;
}
```

在上面的代码中简简单单地定义了一个整型指针变量pa，然后直接对其进行赋值操作，编译的时候没有发现任何错误，但是运行程序时发现程序直接崩溃了。为什么会出现这样的现象呢？这是因为一开始定义的指针变量指向的是一个“不可用”的内存，不能够对其进行赋值操作，所以在使用的时候要尤其注意那些没有初始化的指针。对于那些不使用的指针，读者要养成将其赋值为NULL的习惯。现在将上面的代码修改如下：

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main ()
{
int*pa= (int*) malloc (sizeof (int) *4) ;
int i;
for (i=0; i＜4; i++)
{
* (pa+i) =i+1;
printf ("* (pa+%d) =%d\n", i, * (pa+i) ) ;
```

```
}
free（pa）；
return;
}
```

运行结果：

```
*（pa+0）=1
*（pa+1）=2
*（pa+2）=3
*（pa+3）=4
```

在上面的代码中，通过malloc（）函数为int型指针pa分配了内存，使得pa在内存中有了固定的存储区域，所以在接下来的代码中可以对其进行赋值。值得注意的是，pa使用时占用的内存空间不要超过所分配的内存大小。最后使用了一个free（）函数来释放申请的内存空间，接下来，如果想要再次使用pa指针，该怎么办呢？很多人可能想到了使用if语句来判断当前pa指针是否可用，相应的代码如下：

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main（）
{
int*pa=（int*）malloc（sizeof（int）*9）；
int i;
for（i=0；i＜9；i++）
{
*（pa+i）=rand（）%100;
printf（"*（pa+%d）=%d\t"，i，*（pa+i））；
if（（i+1）%3==0）
printf（"\n"）；
}
free（pa）；
int a=12;
if（pa！=NULL）
```

```
{
pa=&a;
printf（"%d\n", *pa）;
}
return;
}
```

运行结果：

```
*（pa+0）=41*（pa+1）=67*（pa+2）=34
*（pa+3）=0*（pa+4）=69*（pa+5）=24
*（pa+6）=78*（pa+7）=58*（pa+8）=62
12
```

在上面的代码中，在使用free（）函数释放掉pa所指向的内存单元之后，接下来使用if语句判断当前的pa是否为空。从运行结果可以发现，虽然使用free（）函数释放了pa所指向的内存单元，但是pa的值并不为NULL，而是打印输出了12。使用free（）函数前后，pa指针究竟有什么变化呢？看看下面的代码。

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main（）
{
int*pa=（int*）malloc（sizeof（int））;
*pa=726;
printf（"*pa=%d\n", *pa）;
printf（"使用free（）函数之前pa=%d\n", pa）;
free（pa）;
printf（"使用free（）函数之后pa=%d\n", pa）;
return;
}
```

运行结果：

```
*pa=726
使用free（）函数之前pa=3681952
使用free（）函数之后pa=3681952
```

从上面的运行结果发现，在使用free（）函数前后，指针变量pa所指向的内存单元的地址并没有发生变化，所以在前面的代码中不能通过if语句来判断是否对pa使用了free（）函数，而要人为指定pa的值。因此我们要养成将那些使用了free（）函数的指针赋值为NULL的习惯，这样还能够很好地防止使用完free（）函数之后再次使用free（）函数而导致程序崩溃的情况出现，如：

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main（）
{
int*pa=（int*）malloc（sizeof（int））；
*pa=726;
printf（"使用free（）函数之前*pa=%d\n"，*pa）；
free（pa）；
printf（"使用free（）函数之后*pa=%d\n"，*pa）；
pa=NULL;
free（pa）；
free（pa）；
return;
}
```

运行结果：

```
使用free（）函数之前*pa=726
使用free（）函数之后*pa=-572662307
```

由上面的运行结果可以看出，在使用完free（）函数之后，变量pa

所指向的内存单元中的数据变为“垃圾数据”，不再是之前保存的数据。同时，在使用了free（）函数之后将pa指针的值设置为NULL，之后又两次对pa使用了free（）函数，这时程序并没有因此而崩溃，所以在指针的操作中，要养成将那些不再使用了的指针赋值为NULL的习惯，避免由于粗心带来的错误。

5.3 指针与地址之间的关系

在C语言中，我们所说的地址就是计算机中内存单元的编号，通过这个编号，我们可以访问相应的内存单元。就像学生宿舍一样，每个宿舍都有一个编号，通过编号可以找到相应的宿舍。在此之前已经讲过指针，它的本质就是地址，也就是一个地址编号，它指向一片连续的内存单元，就好比安排同一个班级的学生在相连的宿舍中，只需要知道安排宿舍的多少和安排的第一个宿舍号就可以逐一找出班级所有的学生。相信读者对*和＆已经不再陌生了，*可以作为定义指针时的形式说明符和取出指针变量保存的地址所指向的内存单元的值。我们可以通过*结合地址的方式来访问内存单元中的数据并存入数据；而＆是取地址运算符，通过它得到变量的地址。接下来通过下面的一段代码来演示如何通过地址取出该地址单元中的内容。

```
#include<stdio.h>
void main()
{
int a;
a=9;
printf("a=%d\n", *(&a));
return;
}
```

运行结果：

```
a=9
```

在代码的开始部分，申请了一个int的存储单元，之后在该单元中放入一个整数9，为了演示上面所说的指针与地址间的关系，接下来通过&a取得变量a在内存中的编号，也就是地址，然后通过*来得到地址单元中的内容。通过打印输出可以发现，取出的该单元的值就是在开始部分对该单元赋的值，所以可以通过*结合地址的方式来访问内存单元。接下来通过下面的代码来了解指针变量与地址之间的关系。

```
#include＜stdio.h＞
void main ()
{
int a;
int*pa;
a=12;
pa=&a;
printf（"指针变量pa的地址：%d\n\n"，&pa）；
printf（"int型变量a的地址为：%d\n"，&a）；
printf（"int型指针的pa的值为：%d\n\n"，*（&pa））；
printf（"a的值为：%d\n"，a）；
printf（"int型指针pa所指的内存单元的值为：%d\n"，*pa）；
return;
}
```

运行结果：

```
指针变量pa的地址：1245048
int型变量a的地址为：1245052
int型指针的pa的值为：1245052
a的值为：12
int型指针pa所指的内存单元的值为：12
```

可以通过图5-5演示上面的运行结果，指针变量同样有一个内存地址，将变量a的地址存放到int型指针pa中，即int指针pa指向变量a在内存

中的存储单元，所以通过*pa来打印pa所指内存单元的值时得到的是变量a的值。

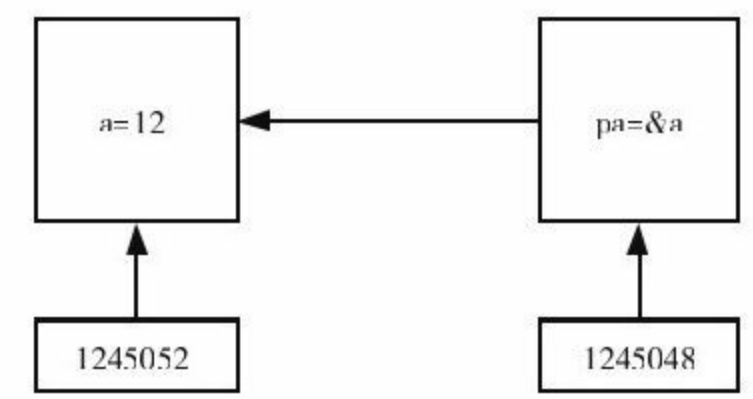

图　5-5　int型指针pa与变量a地址间的关系

使用不同类型的指针指向同一片内存单元的首地址，由于不同类型的指针所指向内存单元的长度不同，所以尽管它们指向同一个起始地址，但是其指向的内存单元的值并不相同。为了便于理解，我们在此给出一段代码加以分析。

```
#include＜stdio.h＞
void main ()
{
int a;
short*pa=NULL;
char*pb;
a=0x12345678;
pa=（short*）&a;
pb=（char*）&a;
printf（"short类型占用的内存大小为：%d\n"，sizeof（short））；
printf（"char类型占用的内存大小为：%d\n\n"，sizeof（char））；
printf（"int型指针的pa的值为：%x\n"，pa）；
printf（"int型指针的pb的值为：%x\n\n"，pb）；
printf（"int型指针的pa所指向内存单元的值为：%x\n"，*pa）；
printf（"int型指针的pb所指向内存单元的值为：%x\n"，*pb）；
return;
```

```
}
```

运行结果：

```
short类型占用的内存大小为：2
char类型占用的内存大小为：1
int型指针的pa的值为：12ff7c
int型指针的pb的值为：12ff7c
int型指针的pa所指向内存单元的值为：5678
int型指针的pb所指向内存单元的值为：78
```

可以通过图5-6来演示上面的运行结果，pa和pb都指向变量a在内存中的首地址，但是它们所指向的内存单元的值并不一样，这是因为定义的pa和pb属于不同的指针类型，由代码中的sizeof（）操作符求得的short型和char型在内存中所占的大小可知，定义的short类型指针变量pa在内存中占有2字节的大小，输出的是从变量a的起始地址开始的2字节的内容，而char类型指针变量pb在内存中占有1字节的大小，输出的是从变量a的起始地址开始的1字节的内容。

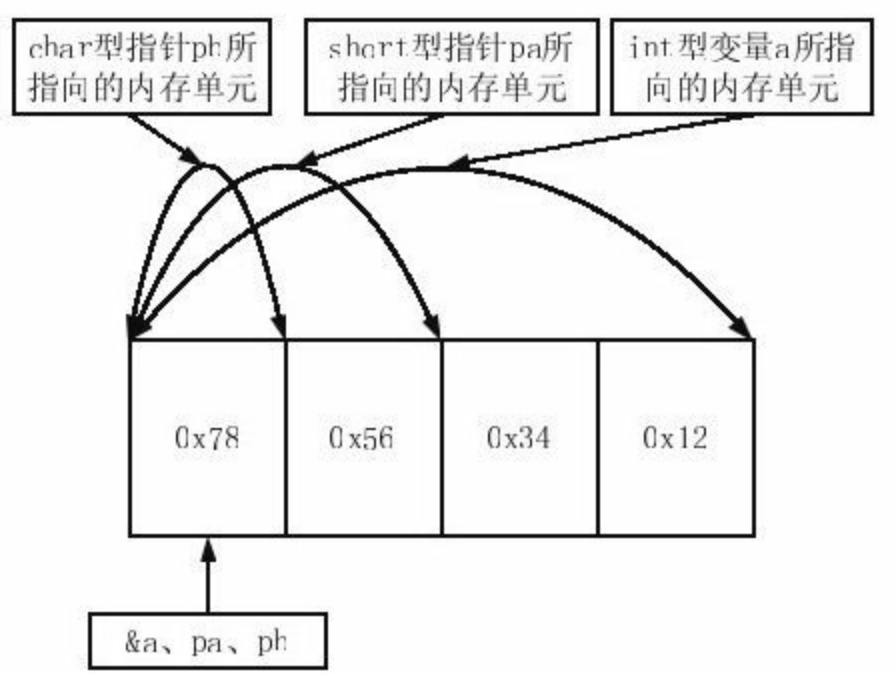

图　5-6　不同类型指针间的内存结构

值得注意的是，定义的指针变量可以进行自加和自减运算，而地址的值是一个常量，不能进行自加和自减运算。

在使用指针变量进行运算时要特别留意运算的优先级，因为这样的错误在编译的过程中不会出现任何警告信息，这样会大大增加查错时间。接下来通过下面的一段代码加以分析。

```
#include＜stdio.h＞
void main ()
{
int a[9];
int*pa;
int i;
for (i=0; i＜9; i++)
{
a[i]=i+1;
}
pa=a;
```

```
for（i=0; i＜9; i++）
{
printf（"a[%d]=%d\t", i, *pa++）;
if（（i+1）%3==0）
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
```

在程序中定义了一个int型数组，其长度为9，分配在一片连续的内存区域中，同时定义了一个int型的指针，它指向数组a在内存中的首地址，然后通过指针pa间接地取出内存中数组元素的值。值得注意的是，执行*pa++运算时先取出pa所指内存单元的值，然后将指针移动到下一个位置，切记不可将其写为（*pa）++。这里暂且不讨论这种做法的对与错，先通过下面一段代码来一窥究竟。

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main（）
{
int a[4];
int*pa;
int*pb;
int i;
printf（"\n通过数组a来直接打印其中的元素值\n"）;
for（i=0; i＜4; i++）
{
a[i]=rand（）%100;
printf（"a[%d]=%d\t", i,a[i]）;
```

```
}
pa=a;
printf（"\n通过int型指针pa来间接打印数组a中的元素值\n"）；
for（i=0; i＜4; i++）
{
printf（"a[%d]=%d\t", i, *pa++）；
}
pb=a;
printf（"\n通过int型指针pb来间接打印数组a中的元素值\n"）；
for（i=0; i＜4; i++）
{
printf（"a[%d]=%d\t", i, （*pb）++）；
}
return;
}
```

运行结果：

```
通过数组a来直接打印其中的元素值
a[0]=41 a[1]=67 a[2]=34 a[3]=0
通过int型指针pa来间接打印数组a中的元素值
a[0]=41 a[1]=67 a[2]=34 a[3]=0
通过int型指针pb来间接打印数组a中的元素值
a[0]=41 a[1]=42 a[2]=43 a[3]=44
```

从上面的运行结果发现，两种间接打印出来的数组元素的值并不相同。现在来分析一下，pb指向的是数组a的首地址，所以通过（*pa）得到的是数组元素a[0]，可以将（*pb）++等价转换为a[0]++，从而轻易地发现出错的原因，使用（*pb）++这样的方式并不能够间接引用数组a中的元素。

在使用指针变量进行运算的时候，要清楚地知道参与运算后的指针变量的值是否改变，如果所使用的指针发生了变化，却还是按照之前的方式来引用，那么就会发生不可预测的错误，因此对于指针变量的值是

否发生变化要有一个清楚的认识。看看下面的代码：

```
#include＜stdio.h＞
#include＜stdlib.h＞
void main（）
{
int a[4];
int i;
int*pa;
int*pb;
for（i=0; i＜4; i++）
{
a[i]=i*2;
}
printf（"数组a的首地址为%d\n"，a）；
pa=a;
for（i=0; i＜4; i++）
printf（"a[%d]=%d\t"，i，*（pa+i））；
printf（"\n通过*（pa+i）间接引用数组a中元素之后pa的值为：%d\n"，pa）；
pb=a;
for（i=0; pb＜pa+4; ）
printf（"a[%d]=%d\t"，i++，*pb++）；
printf（"\n通过*pb++间接引用数组a中元素之后pb的值为：%d\n"，pb）；
return;
}
```

运行结果：

```
a[0]=0 a[1]=2 a[2]=4 a[3]=6
通过*（pa+i）间接引用数组a中元素之后pa的值为：1245040
a[0]=0 a[1]=2 a[2]=4 a[3]=6
通过*pb++间接引用数组a中元素之后pb的值为：1245056
```

从上面的运行结果发现，采用两种引用方式之后指针变量的值并不相同，这是因为在前一种引用方式中始终没有移动指针变量的位置，pa始终指向数组元素a的首地址，所以在用pb引用数组中的元素时，在for循环中巧妙地利用pa变量建立一个是否执行循环体的判断条件。而pb指

针变量进行自加运算相当于pb=pb+1，所以pb指针变量的值每次都在改变，使最终退出循环之后pb已经不再指向整型数组a在内存中的存储单元，这时如果使用printf打印它所指向的内存单元的数据，就会发现其中存储的是一些随机数。

5.4 指针与数组之间的关系

指针和数组之间究竟有什么样的关系，接下来就通过以下几个方面一一加以讲解。

1.指针对于数组的引用

在前面讲解数组时也提到了指针，但是并没有进行具体的分析，通过指针来引用一维数组也已经多次讲到了，在此就不过多讲解，接下来先通过一段代码来看如何通过指针来引用二维数组。

```
#include<stdio.h>
void main ()
{
int a[3][3];
int i,j;
int*pa;
printf ("直接引用二维数组a中的元素\n") ;
for (i=0; i<3; i++)
{
for (j=0; j<3; j++)
{
a[i][j]=i*3+j+1;
printf ("a[%d][%d]=%d\t", i,j, a[i][j]) ;
}
printf ("\n") ;
}
pa=a[0];
printf ("通过指针引用二维数组a中的元素\n") ;
for (i=0, j=0; pa<=&a[2][2]; pa++)
{
printf ("a[%d][%d]=%d\t", i,j, *pa) ;
j++;
if (3==j)
{
j=0;
```

```
i++;
printf ("\n") ;
}
}
return;
}
```

运行结果：

```
直接引用二维数组a中的元素
a[0][0]=1 a[0][1]=2 a[0][2]=3
a[1][0]=4 a[1][1]=5 a[1][2]=6
a[2][0]=7 a[2][1]=8 a[2][2]=9
通过指针引用二维数组a中的元素
a[0][0]=1 a[0][1]=2 a[0][2]=3
a[1][0]=4 a[1][1]=5 a[1][2]=6
a[2][0]=7 a[2][1]=8 a[2][2]=9
```

以上代码如何通过指针来成功地引用二维数组中的元素的呢？下面通过图5-7来分析。由于计算机的内存结构是一维的，即线性的，因此计算机在内存中存储多维数组时要将其转换为一维来进行存储。根据一维数组存储时按照下标来标识存储的先后顺序，下标从0开始，多维数组转换为一维数组的方法是从最外维开始转换，之后是次外维，依此类推，直到第一维。

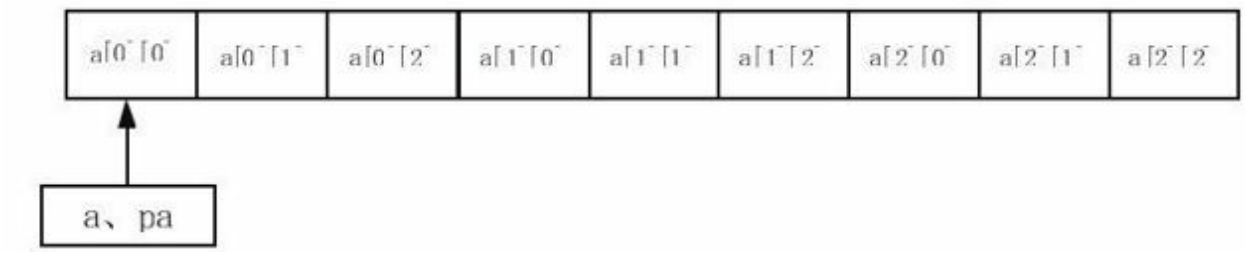

图　5-7　指针对二维数组的引用

前面讲解数组时提到了动态数组的概念，在创建多维动态数组的时候，采用的是多级指针的方法，但是没有提到另外两种动态数组的创建方式，即通过指针数组和数组指针的方式，下面就来看看如何通过它们来创建动态数组。

2.通过指针数组创建动态数组

接下来通过代码介绍如何通过一个指针数组来创建一个动态的二维数组，读者可将其创建方法与本书4.6节介绍的创建方法进行对比。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define N 3
#define M 4
void main ()
{
int*a[N];
int i,j;
for (i=0; i＜3; i++)
{
if ( (a[i]= (int*) malloc (M*sizeof (int) ) ) ==NULL)
{
printf ("分配失败! ") ;
exit (0) ;
}
}
for (i=0; i＜N; i++)
{
for (j=0; j＜M; j++)
{
a[i][j]=i*M+j+1;
printf ("a[%d][%d]=%d\t", i,j, a[i][j]) ;
}
printf ("\n") ;
}
for (i=0; i＜N; i++)
free (a[i]) ;
return;
```

```
}
```

运行结果：

```
a[0][0]=1 a[0][1]=2 a[0][2]=3 a[0][3]=4
a[1][0]=5 a[1][1]=6 a[1][2]=7 a[1][3]=8
a[2][0]=9 a[2][1]=10 a[2][2]=11 a[2][3]=12
```

上面的代码中，在起始部分定义了一个整型指针数组，其第一维的长度是已知的，第二维的长度是在运行过程中指定的，相当于定义了一个含有3个元素的数组，每个元素又是一个指针，为其分配一片连续的内存单元，在此为每个元素分配了4个整型变量的存储空间。使用完动态数组之后，最重要的一点就是将其申请的内存空间通过free（）函数释放，否则将会造成内存泄露。当然，利用指针数组也可以定义多维数组，定义方法与二维数组类似。

3.通过数组指针创建动态数组

学习了如何通过指针数组创建动态数组后，接下来学习如何通过数组指针创建动态数组。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define N 3
#define M 4
void main ()
{
int (*a) [N];
int i,j;
if ( (a= (int (*) [N]) malloc (N*M*sizeof (int) ) ) ==NULL)
```

```
{
printf（"分配失败！"）；
exit（0）；
}
printf（"当以4X3二维数组方式引用创建的动态数组时\n"）；
for（i=0；i＜M；i++）
{
for（j=0；j＜N；j++）
{
a[i][j]=i*N+j+1;
printf（"a[%d][%d]=%d\t"，i,j，a[i][j]）；
}
printf（"\n"）；
}
printf（"当以3X4二维数组方式引用创建的动态数组时\n"）；
for（i=0；i＜N；i++）
{
for（j=0；j＜M；j++）
{
a[i][j]=i*M+j+1;
printf（"a[%d][%d]=%d\t"，i,j，a[i][j]）；
}
printf（"\n"）；
}
free（a）；
return;
}
```

运行结果：

```
当以4X3二维数组方式引用创建的动态数组时
a[0][0]=1 a[0][1]=2 a[0][2]=3
a[1][0]=4 a[1][1]=5 a[1][2]=6
a[2][0]=7 a[2][1]=8 a[2][2]=9
a[3][0]=10 a[3][1]=11 a[3][2]=12
当以3X4二维数组方式引用创建的动态数组时
a[0][0]=1 a[0][1]=2 a[0][2]=3 a[0][3]=4
a[1][0]=5 a[1][1]=6 a[1][2]=7 a[1][3]=8
a[2][0]=9 a[2][1]=10 a[2][2]=11 a[2][3]=12
```

从上面的运行结果发现，在采用数组指针创建动态数组的时候，第

一维和第二维的分配不是固定的，可以人为地指定，只要第一维和第二维的乘积不超过动态申请时的大小即可，这是用数组指针创建动态数组的特殊之处。

了解了如何通过指针数组和数组指针创建动态数组之后，接下来分析指针数组和数组指针对数组的引用方式。

4.指针数组对数组的引用

关于指针数组怎样引用数组，可以通过下面的代码来加以分析。

```
#include<stdio.h>
#include<stdlib.h>
#define N 3
#define M 4
void main ()
{
int*a[M], b[M][N];
int i,j;
for (i=0; i<M; i++)
for (j=0; j<N; j++)
{
b[i][j]=i+j;
}
for (i=0; i<M; i++)
{
a[i]=b[i];
}
for (i=0; i<M; i++)
{
for (j=0; j<N; j++)
{
printf ("a[%d][%d]=%d\t", i,j, a[i][j]) ;
}
printf ("\n") ;
}
return;
}
```

运行结果：

```
a[0][0]=0 a[0][1]=1 a[0][2]=2
a[1][0]=1 a[1][1]=2 a[1][2]=3
a[2][0]=2 a[2][1]=3 a[2][2]=4
a[3][0]=3 a[3][1]=4 a[3][2]=5
```

在使用指针数组时要清楚，指针数组的本质是数组，数组中的每个元素都是一个指针，我们可以通过图5-8来了解指针数组的引用方式。通过指针数组来引用数组时，只要把数组中每行起始元素的首地址指给指针数组中的每个元素，就可以实现对数组的引用了。

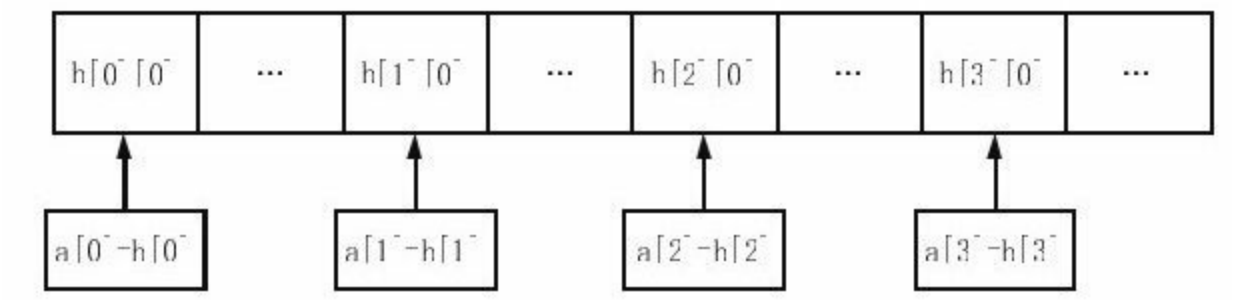

图　5-8　指针数组对数组的引用

5.数组指针对数组的引用

看完指针数组对于数组的引用，接下来通过下面一段代码来看数组指针对数组的引用。

```
#include<stdio.h>
#include<stdlib.h>
#define N 3
#define M 4
```

```
void main（）
{
int（*a）[N]，b[M][N];
int i,j;
for（i=0; i＜M; i++）
for（j=0; j＜N; j++）
{
b[i][j]=i+j;
}
a=b;
for（i=0; i＜M; i++）
{
for（j=0; j＜N; j++）
{
printf（"a[%d][%d]=%d\t"，i,j，a[i][j]）;
}
printf（"\n"）;
}
return;
}
```

运行结果：

```
a[0][0]=0 a[0][1]=1 a[0][2]=2
a[1][0]=1 a[1][1]=2 a[1][2]=3
a[2][0]=2 a[2][1]=3 a[2][2]=4
a[3][0]=3 a[3][1]=4 a[3][2]=5
```

由上面的代码可知，在使用数组指针引用数组时，只需要将数组元素的首地址赋值给定义的数组指针即可，但是要注意，表示数组首地址的方法很多，但不是每种表示方法都可以。＆b是数组的首地址，如果执行“a=＆b；”这样的赋值操作，那么会出现“error C2440：'='：cannot convert from'int（*）[4][3]'to'int（*）[3]'”错误。将“int b[4][3]”变形为“int*（＆b）[4][3]”，就会清楚地发现，＆b相当于int（*）[4][3]类型，而定义的数组指针是int（*）[3]类型，显然不匹配，因此会出错。

采用“a=＆b[0]；”这样的赋值方式可以吗？答案是肯定的。与前面分析多维数组的方法相同，b[0]元素相当于一个包含了3个整型变量的一维数组，可将b[0]视为一种特殊的类型int[3]，这时再进行变形，将＆b[0]视为int（*）[3]类型，与定义的a刚好一致，所以可以进行这样的赋值。有人可能会有疑惑，仅仅将第一行的首地址＆b[0]给了a，如何实现取出数组中所有的元素值呢？前面讲过，多维数组在计算机内存中是按照一维的方式进行存储的，并且数组元素存储在一片连续的内存区域中，所以能够逐一取出数组中的元素值。需要注意的是，数组中两个下标的含义是不一样的，在此以二维数组为例进行讲解。

```
#include＜stdio.h＞
#define N 2
#define M 3
void main（）
{
int a[N][M];
int i,j;
for（i=0; i＜N; i++）
for（j=0; j＜M; j++）
{
a[i][j]=i+j;
}
printf（"直接求取数组元素的地址\n"）;
for（i=0; i＜N; i++）
{
for（j=0; j＜M; j++）
{
printf（"＆a[%d][%d]=%d\t", i,j, ＆a[i][j]）;
}
printf（"\n"）;
}
printf（"间接求取数组元素的地址\n"）;
for（i=0; i＜N; i++）
{
for（j=0; j＜M; j++）
{
```

```
printf（"&a[%d][%d]=%d\t", i,j, a[i]+j）;
}
printf（"\n"）;
}
return;
}
```

运行结果：

```
直接求取数组元素的地址
&a[0][0]=1245032&a[0][1]=1245036&a[0][2]=1245040
&a[1][0]=1245044&a[1][1]=1245048&a[1][2]=1245052
间接求取数组元素的地址
&a[0][0]=1245032&a[0][1]=1245036&a[0][2]=1245040
&a[1][0]=1245044&a[1][1]=1245048&a[1][2]=1245052
```

直接求取数组元素地址和间接求取数组元素地址得到的结果一样，下面通过图5-9来分析。数组名和第一维下标决定了一行元素的起始地址，而第二维下标决定了数组元素存储单元相对于行起始地址的偏移量，所以在代码中通过a[i]+j同样可以存储没有数组元素的起始地址。注意，如果写成“a[i][j]”这样的形式，那么就意味着当前取出的是距离起始地址a[i]的偏移量为j的存储单元的数组元素的值，所以第二维的[]的作用和前面所说的*有着相同的作用，都用于得到地址所指向内存单元存储的值。

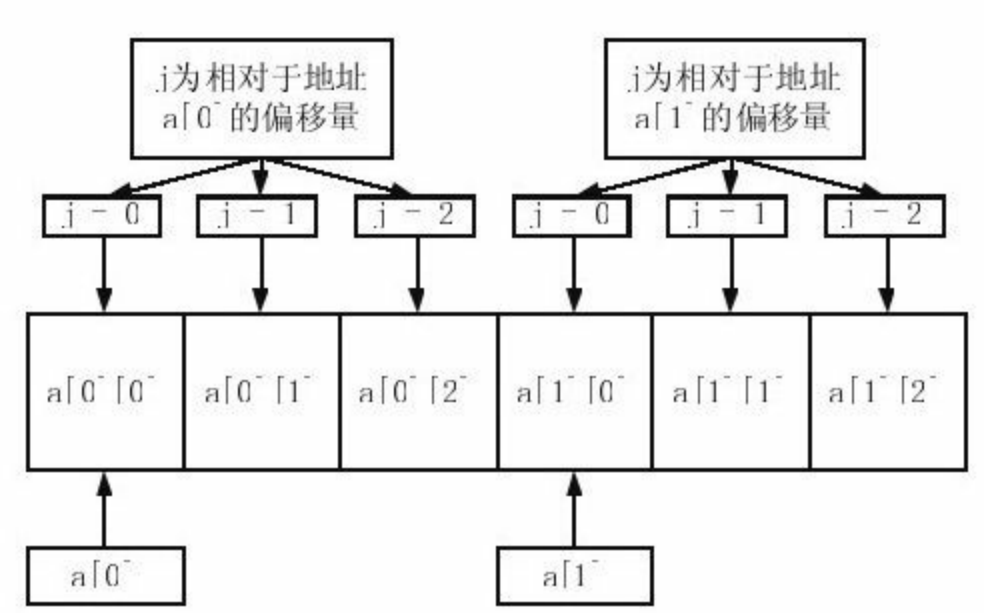
j为相对于地址
a[0]的偏移量
j为相对于地址
a[1]的偏移量
j－0
j－1
j－2
j－0
j－1
j－2
a[0][0]
a[0][1]
a[0][2]
a[1][0]
a[1][1]
a[1][2]
a[0]
a[1]

图　5-9　数组元素的引用

5.5 指针与字符串之间的关系

在C语言中可以采用两种方式来定义和访问一个字符串，一种是数组，另外一种是指针。其中采用数组的方式定义和访问字符串在第4章中已经介绍过了，接下来了解一下通过指针定义字符串的特点。

```
#include<stdio.h>
void main ()
{
char*str="Hello Bigloomy! ";
printf ("%s\n", str) ;
return;
}
```

运行结果：

```
Hello Bigloomy!
```

前面在介绍数组与字符串时特别强调过，使用printf（）函数时字符串的后面必须有结束符‘\0’，否则就会出错。上面的代码运行时正确打印出字符串说明通过指针定义字符串时会自动在字符串后面添加一个结束符‘\0’，我们可以通过下面的代码来验证。

```
#include<stdio.h>
void main ()
{
char*str="Hello Bigloomy";
while (*str! ='\0')
printf ("%c", *str++) ;
printf ("\n") ;
return;
```

```
}
```

运行结果：

```
Hello Bigloomy!
```

我们可以通过图5-10来分析上面的代码，没有采用%s格式一次性打印出字符串，而是按照单个字符的方式打印出其中的字符串。需要注意的是，通过指针定义字符串时，字符存储在一片连续的内存单元中，一开始，str指针指向的是字符串的首地址，通过*str++来逐一取出字符同时将指针移到下一个位置，通过判断字符后面是否有结束符‘\0’来确定是否结束打印。

接下来通过一段代码了解通过数组初始化的字符串和通过字符指针初始化的字符串有什么区别，先来看通过数组初始化的字符串。

```
#include＜stdio.h＞
void main（）
{
char str[20]="Hello World! ";
str[6]='B', str[7]='i', str[8]='g', str[9]='l';
str[10]='o', str[11]='o', str[12]='m', str[13]='y';
printf（"%s\n", str）;
return;
}
```

运行结果：

```
Hello Bigloomy
```

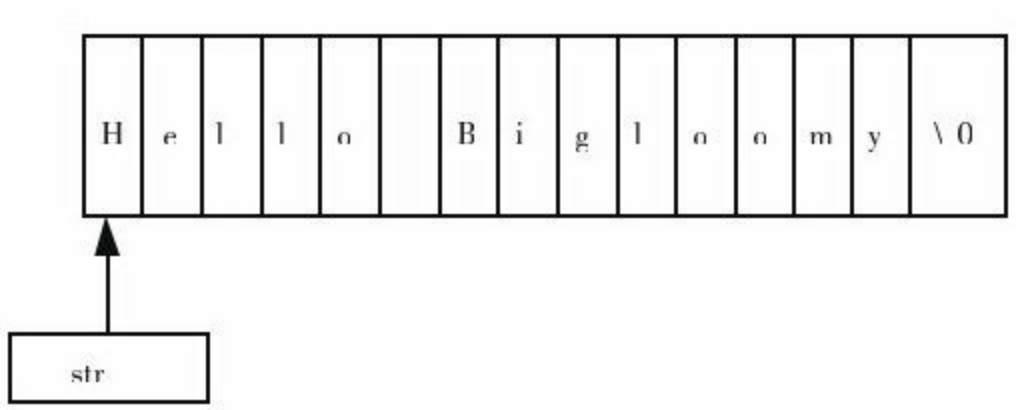

图　5-10　char型指针对字符串中字符的引用

在上面的代码中，先通过数组来定义字符串，在随后的代码中可以通过数组元素来引用和修改字符串中的值。但是如果使用的是指针定义的字符串，能否进行修改呢？看看下面的代码。

```
#include＜stdio.h＞
void main ()
{
char*str="Hello World! "; * (str+6) ='B';
printf ("%s\n", str) ;
return;
}
```

这段代码在编译时不会出现任何问题，但是在运行的时候就直接崩溃了，这是为什么呢？这是因为通过指针定义的字符串在内存中具有只读属性，不能在其后的代码中进行任何修改，只可以引用。如果指针指向的字符串不具有只读属性，那么可以对其进行修改。如：

```
#include＜stdio.h＞
void main ()
{
char str[20]="Hello World! ";
```

```
char*ptr;
ptr=str;
ptr[6]='B', ptr[7]='i', ptr[8]='g', ptr[9]='l';
ptr[10]='o', ptr[11]='o', ptr[12]='m', ptr[13]='y';
printf ("%s\n", ptr) ;
return;
}
```

运行结果：

```
Hello Bigloomy
```

在上面的代码中，由于字符串是采用数组的方式定义的，不具备只读属性，因此可以对其进行修改。定义一个字符指针ptr，然后将字符数组元素的首地址传递给ptr，接下来就可以通过字符指针来对其所指向的内存单元进行相应的修改了，ptr[i]表示的是取出距离ptr地址的偏移量为i的内存单元的值，这与采用*（ptr+i）的效果一样。

之前讲解指针的时候曾指出，不能够直接对未经初始化的指针进行赋值操作，所以赋值之前需要使指针指向一个可用的地址空间，否则指针的指向是不确定的，虽然编译的时候不会出现错误，但是运行程序时就会直接崩溃并退出，如：

```
#include＜stdio.h＞
void main ()
{
int*a;
*a=9;
return;
}
```

以上程序在编译时不会发生任何错误，但运行时程序直接崩溃退出，解决这个问题的办法是先使指针a指向一个确定的地址，通常的做法是，让a指向某个已经分配了的内存地址或者为a在内存中分配一个存储空间。在采用指针定义字符串时没有显示地为指针分配内存，而编译器在编译的过程中会在内存中为字符串分配一个内存区域，同时将分配的内存区域的首地址传递给指针变量，因此在通过指针定义字符串时不必为其分配内存空间，编译器在编译的过程中会自动完成对字符串内存的分配，这与之前讲解的一般数值指针不同。

5.6 指针与函数之间的关系

在第1章中简单介绍了指针与函数的关系，但是并没有进行深入的讲解。关于函数指针和指针函数的内容，将会在第7章进行深入讲解，在此重点介绍指针作为函数参数的使用情况。

指针作为函数参数，最典型的莫过于main（）函数，所以先通过main（）函数来看看指针作为参数的使用情况。

```
#include<stdio.h>
void main(int argc,char*argv[])
{
while(argc>0)
{
printf("%s\t",*argv++);
argc--;
}
return;
}
```

运行结果：

```
C:\>fdsa.exe hello world
fdsa.exe hello world
```

先介绍main（）函数中参数的含义，其中，argc表示命令行输入参数的个数，在此命令行参数有fdsa.exe、hello、world三个，所以argc的值为3；而argv表示一个指针数组，用来存放命令行输出的字符串，相当于“char*argv[]={"fdsa.exe"，"hello"，"world"}；”。细心的读者可能发

现了一个问题，前面说过，数组名不能够进行自加和自减等运算，但是上面的代码中却利用参数中的数组名进行自加运算，而且成功输出了。难道前面说错了吗？看下面这段代码。

```
#include<stdio.h>
void main()
{
char*argv[]={"fdsa.exe", "hello", "world"};
int i=0;
while(i<3)
{
printf("%s\t", *argv++);
i++;
}
return;
}
```

以上程序编译的时候出错了，看来，数组名的确不可以进行自加和自减运算。其实，在前面的程序中，数组作为参数时成功地实现了输出，是因为数组作为参数时已经不再是数组了，而是一个指针变量，所以能够对其进行自加运算。我们可以通过下面几段代码来逐一分析。

```
#include<stdio.h>
void print(char str[])
{
printf("print函数中的sizeof(str)=%d\n", sizeof(str));
printf("%s\n", str);
return;
}
void main()
{
char str[]="Hello Bigloomy!";
printf("main函数中的sizeof(str)=%d\n", sizeof(str));
print(str);
return;
}
```

运行结果：

```
main函数中的sizeof（str）=16
print函数中的sizeof（str）=4
Hello Bigloomy!
```

从前面的代码中可以发现，通过sizeof操作符从main（）函数和print（）函数中得到的str大小并不一样，在main（）函数中是16字节，与前面讲解字符串数组时分析的一样，会在字符串的后面自动添加一个串结束符‘\0’，所以大小是16。而在print（）函数中的str大小为4字节，由此可以看出，在参数str传递的过程中传递的并不是整个数组的大小，而是数组的首地址，这时print（）函数中的参数不是一个数组，而是一个指针变量。如果读者觉得上面的解释不好理解，那么再来看下面这段代码。

```
#include＜stdio.h＞
void print（char str[]）
{
printf（"print函数中的str=%d\n"，str）；
printf（"print函数中的＆str=%d\n"，＆str）；
printf（"%s\n"，str）；
return；
}
void main（）
{
char str[]="Hello Bigloomy";
printf（"main函数中的str=%d\n"，str）；
printf（"main函数中的＆str=%d\n\n"，＆str）；
print（str）；
return；
}
```

运行结果：

```
main函数中的str=1245040
main函数中的&str=1245040
print函数中的str=1245040
print函数中的&str=1244960
Hello Bigloomy!
```

在main（）函数中打印的str和&str结果相同，都是数组的首地址；而在print（）函数中打印的str和&str并不相同，str与在main（）函数中打印的结果相同，表示数组的首地址，而&str却不一样，这也进一步说明这里的str是一个指针变量，而不再是一个数组。如图5-11所示，print（）函数中的str是一个指针变量，所以它拥有自己的地址。这就是在两个函数中得到的结果不一致的原因，参数传递过去的仅仅是数组的首地址，所以在print（）函数中的str指针变量中保存的是数组的首地址。为了便于理解，通常将参数传递中的“char str[]”改写为“char*str”。

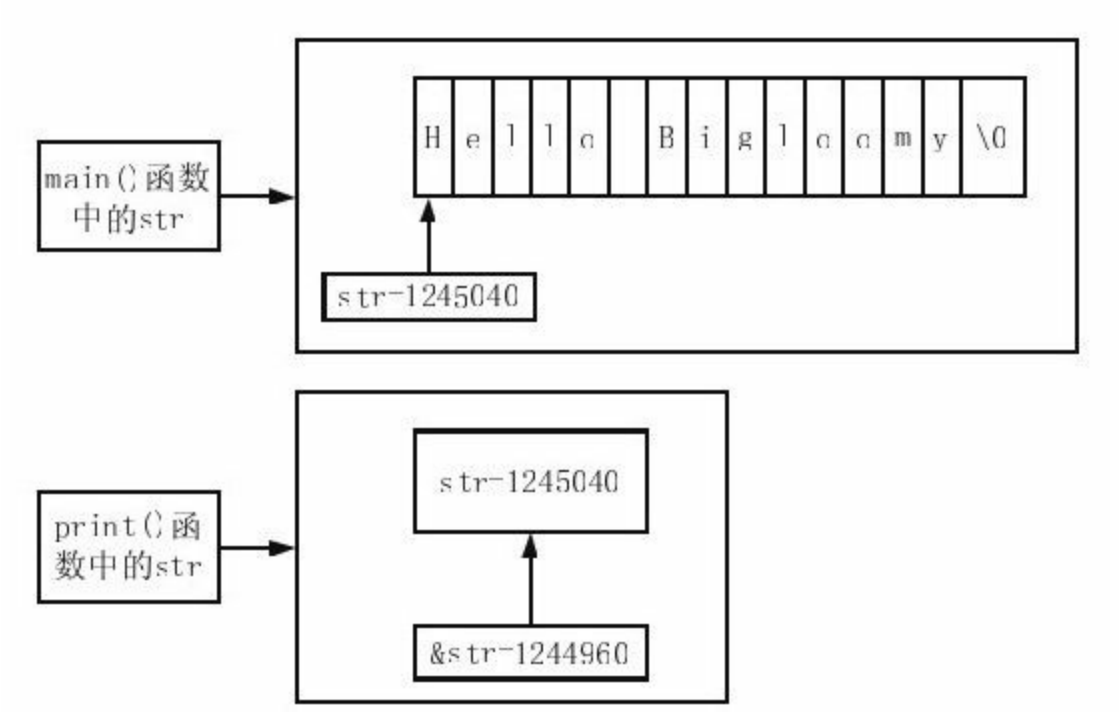

图　5-11　main（）函数和print（）函数中str的内存结构

下面再通过一段代码来看指针作为参数的使用情况。

```
#include<stdio.h>
void copy_string(char from[], char to[])
{
while(*to++=*from++);
return;
}
void main()
{
char str[]="this is a string! ";
printf("%s\n", str);
char dec_str[20];
copy_string(str,dec_str);
printf("%s\n", dec_str);
return;
}
```

运行结果：

```
this is a string!
this is a string!
```

通过函数copy_string（）成功地将str中的字符串复制到dec_str数组中，虽然copy_string（）函数中的参数是数组形式的，但是它的实质是指针变量。再看copy_string（）函数的实现方式，这里通过while循环来实现，由于str字符数组的后面有结束符‘\0’，因此我们巧妙地将这一条件作为复制是否结束的标志。需要注意的是，在mian（）函数中传递的两个参数都是已经在内存中分配了的。如果没有在main（）函数中分配目标变量的内存大小，会出现什么问题呢？看看下面的代码。

```
#include＜stdio.h＞
#include＜stdlib.h＞
void copy_string（char*from,char*to）
{
to=（char*）malloc（sizeof（char）*40）；
char*to_start=to;
for（；*to=*from；from++，to++）；
printf（"%s\n"，to_start）；
return;
}
void main（）
{
char*str="Hello World! ";
printf（"%s\n"，str）；
char*dec_str;
copy_string（str,dec_str）；
//printf（"%s\n"，dec_str）；
return;
}
```

运行结果：

```
Hello World!
Hello World!
```

在上面的代码中，在main（）函数中没有为dec_str指针分配内存单元，但是在copy_string（）函数中为参数to分配了内存单元，成功地在函数copy_string（）中实现了字符串的复制。那么这个复制好的字符串能否像上面的字符串那样返回呢？取消注释最后一行printf语句来验证是否成功返回复制后的字符串，我们会发现，运行到这一行时直接崩溃掉了。这是什么原因所导致的呢？我们通过下面一段代码来加以分析。

```
#include＜stdio.h＞
#include＜stdlib.h＞
void copy_string（char from[]，char to[]）
{
printf（"分配前to=%x\n"，to）；
to=（char*）malloc（sizeof（char）*20）；
printf（"分配后to=%x\n"，to）；
char*start_to=to;
while（*to++=*from++）；
printf（"copy_string函数：%s\n\n"，start_to）；
return;
}
void main（）
{
char str[]="Hello World! ";
printf（"main函数：%s\n"，str）；
char*dec_str;
printf（"调用函数copy_string前dec_str=%x\n\n"，dec_str）；
copy_string（str,dec_str）；
printf（"调用函数copy_string后dec_str=%x\n"，dec_str）；
return;
}
```

运行结果：

```
main函数：Hello World!
调用函数copy_string前dec_str=cccccccc
分配前to=cccccccc
分配后to=380fe0
copy_string函数：Hello World!
调用函数copy_string后dec_str=cccccccc
```

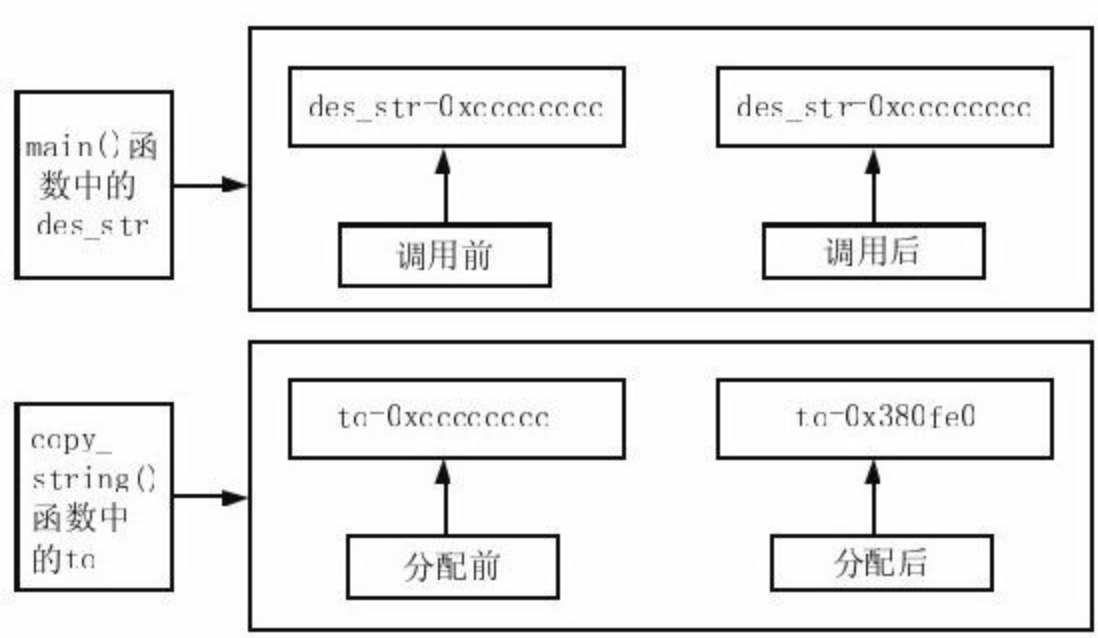

图　5-12　char型指针des_str和to的内存结构

通过图5-12来演示上面的运行结果，在main函数中没有为目标字符指针变量申请内存空间，在调用函数之前发现编译器对指针进行初始化，该指针指向一个不可用的内存；在copy_string（）函数中，在对字符指针变量to分配内存前后它的指向发生了变化，从之前的cccccccc，变为指向一片可用的内存单元，所以接下来成功地实现了字符串的复制，但是在调用copy_string（）函数之后，指针变量des_str的值并没有发生变化，还是指向地址为cccccccc的存储单元，因此在执行printf语句的时候导致程序崩溃。

由上面的分析可知，实参的地址并没有改变，导致dec_str指针还是指向一个不可用的内存地址，从而在使用printf语句打印时出错。如果通过函数返回分配后的起始地址，那么在main（）函数中同样可以打印出通过函数处理后的字符串。

5.7 指针与指针之间的关系

到现在为止，对于指针的知识点基本上可以告一段落了，但是还有一个知识点没有讲解，那就是指针和指针之间的关系，首先通过下面的一段代码来了解最简单的相同类型指针之间的赋值。

```
#include<stdio.h>
void main ()
{
char*str="aaaabbbbccccdddd";
char*ptr;
ptr=str;
printf ("%s\n", ptr) ;
return;
}
```

运行结果：

```
aaaabbbbccccdddd
```

在上面的代码中，先通过一个字符指针定义了一个字符串，然后定义了一个字符指针，通过把字符串的首地址赋值给定义的字符指针，通过printf语句成功打印出定义的字符串。如果定义的不是相同类型的字符指针，那么结果会怎样呢？看看下面的代码。

```
#include<stdio.h>
void main ()
{
char*str="aaaabbbbccccdddd";
int*ptr;
ptr= (int*) str;
```

```
while (*ptr! ='\0')
{
printf ("%s\n", ptr) ;
ptr++;
}
return;
}
```

运行结果：

```
aaaabbbbccccdddd
bbbbccccdddd
ccccdddd
dddd
```

通过图5-13来演示上面代码的运行结果，先通过字符指针定义一个字符串，接下来并没有定义一个相同类型的字符指针，而是定义了一个整型指针ptr，通过强制转换将字符指针的地址转换为整型指针，然后通过整型指针的移动来打印出其中的字符串。需要注意的是，整型指针的自加和自减的移动是每次4字节，而字符指针自加和自减的移动是每次1字节，同时指针所指向的内存区域是从起始地址开始到串结束符为止。

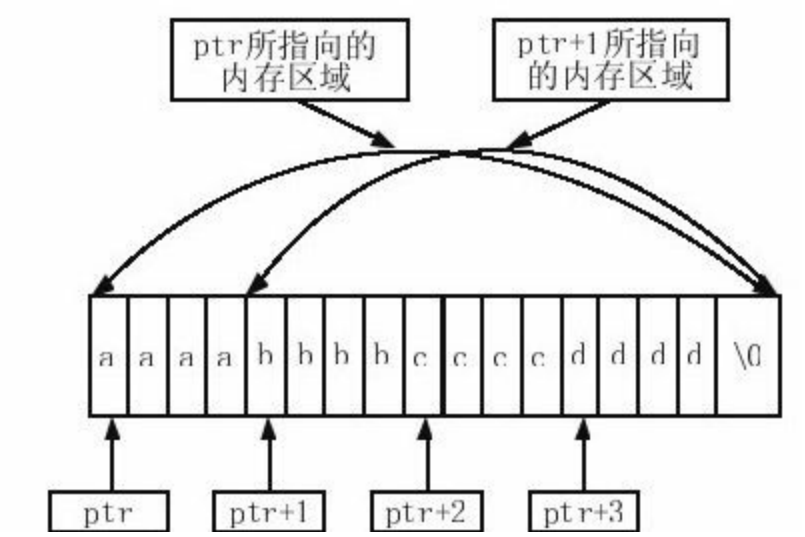

图　5-13　int型变量ptr所指向的内存结构

接下来看多重指针。在讲解多重指针之前，读者要明白的一点是，多重指针变量只能保存同级指针变量的值或它的第一级指针变量的地址。看看下面的代码。

```
#include＜stdio.h＞
void main ()
{
int a;
int*pa;
int**pb;
a=9;
pa=&a;
pb=&pa;
printf ("&a=%d\n\n", &a) ;
printf ("pa=%d\n", pa) ;
printf ("&pb=%d\n", &pa) ;
printf ("*pa=%d\n\n", *pa) ; printf ("pb=%d\n", pb) ;
printf ("&pb=%d\n", &pb) ;
printf ("*pb=%d\n", *pb) ;
printf ("**pb=%d\n", **pb) ;
return;
}
```

运行结果：

```
&a=1245052
pa=1245052
&pa=1245048
*pa=9
pb=1245048
&pb=1245044
*pb=1245052
**pb=9
```

通过图5-14来对上面代码的运行结果加以分析。先定义一个整型变量a，然后分别定义一个一维整型指针pa和二维指针pb，将a的地址保存在指针变量pa中，再将一维指针变量pa的地址保存在pb中，二维指针变量pb同样有自己的地址。从图5-14也可以看出，通过一个*与pb结合可以得到变量a的地址。因为pb中保存的是指针变量pa的地址，所以与*结合即可取出pa地址单元的值，即变量a的地址。*pb再与*结合就可得到变量a的值。注意，虽然定义二维指针pb时使用了两个*号，但是这并不意味着可以对pb使用两个取地址运算符&，两个*号的意思仅仅是指定义的指针变量保存的是一个指针的地址，而不是一个普通常量的地址，当然也可以将相同类型二维指针变量的值赋值给它。

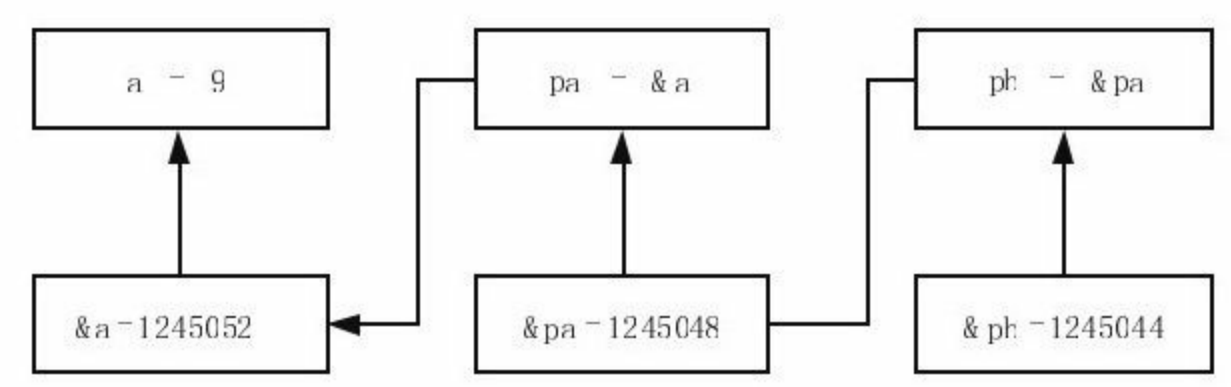

图　5-14　int型变量a、指针pa、二重指针pb的内存结构

第6章　数据结构

数据结构的重要性不仅仅体现在C语言的学习中，对于任何一门编程语言，数据结构的重要性都是不言而喻的。数据结构是计算机存储和组织数据的方式，是相互之间存在一种或多种特定关系的数据元素的集合。一种良好的数据结构可以带来更高的运行或存储效率。可以说，数据结构是脱离语言本身而存在的，不是某一门语言所特有的。本章将通过C语言来讲解如何实现一种典型的数据结构，在讲解链表之前先来了解C语言中一些常见的构造类型，这些知识对于在编程中熟练掌握和使用数据结构尤其重要。

6.1 枚举类型的使用及注意事项

第1章讲解核心概念时初步介绍了枚举类型，但是仅做了一般性介绍，并未介绍其具体的使用，接下来就介绍枚举类型的使用，以及在使用中的一些注意事项。

先来看下面的一段代码。

```
#include<stdio.h>
#include<conio.h>
enum Bool{
True,
False
};
enum Bool is_number(char c)
{
if(c>='0'&&c<='9')
return True;
else
return False;
}
void main()
{
char c;
enum Bool ret;
while(1)
{
printf("\n请输入:");
c=getch();
putchar(c);
ret=is_number(c);
if(ret)
printf("\n输入的是数字字符!");
else
printf("\n输入的是非数字字符!");
}
return;
}
```

运行结果：

```
请输入：4
输入的是非数字字符!
请输入：d
输入的是数字字符!
请输入：*
输入的是数字字符!
请输入：
```

上面的运行结果与我们想要的结果不符，通过仔细观察会发现实现方法很明确，没有什么问题，那么问题出在哪儿呢？看看定义的枚举类型，第1章讲解枚举类型时多次提到，枚举类型的默认初始值是从0开始的，因此对于上面代码中定义的枚举类型，True的值相当于0，而False的值相当于1。当输入数字的时候，返回True，所以打印输出的是else部分的信息。要修改这个错误，可以采用如下两种方法。

方法一：将上面定义的枚举类型修改为如下形式。

```
enum Bool{
False,
True
};
```

这样使定义的枚举类型的值为1，而false的值为0，满足编译器的逻辑处理方法，不会再犯逻辑错误。

方法二：将main（）函数中的实现方法修改为如下形式。

```
void main（）
{
char c;
while（1）
{
printf（"\n请输入："）；
c=getch（）；
putchar（c）；
if（is_number（c）==True）
printf（"\n输入的是数字字符！"）；
else
printf（"\n输入的是非数字字符！"）；
}
return;
}
```

修改main（）函数之后，同样可以得到正确的结果。在if判断语句中，不再简单地通过返回值来打印输出信息，而是通过判断返回值与所需的返回值是否一致来打印输出信息。方法二比方法一更加严格，不易出错，所以建议读者在编写类似的代码时采用方法二。接下来通过一段完整的代码来演示方法二的使用，其功能为判断用户的选择。

```
#include＜stdio.h＞
#include＜conio.h＞
enum answ{
Yes，
No，
Error
};
enum answ answer（）
{
enum answ ans;
char c;
printf（"\n请输入你的选择："）；
c=getch（）；
putchar（c）；
if（'Y'==c||'y'==c）
ans=Yes;
else if（'N'==c||'n'==c）
ans=No;
```

```
else
ans=Error;
return ans;
}
void main ()
{
enum answ ans;
while (1)
{
ans=answer ();
if (ans==Yes)
printf ("\n你的选择是Yes! ");
else if (ans==No)
printf ("\n你的选择是No! ");
else
printf ("\n输入有误! ");
}
return;
}
```

运行结果：

```
请输入你的选择：y
你的选择是Yes!
请输入你的选择：n
你的选择是No!
请输入你的选择：d
输入有误!
请输入你的选择：
```

上面的代码采用了第二种方法，成功实现了对用户输入的判断。

在使用枚举类型的过程中需要注意，虽然枚举类型中的值均为整数值，但是不能对定义的枚举类型变量直接赋整数值，而应该进行强制转换，如：

```
#include<stdio.h>
void main ()
```

```
{
enum num{
one,
two,
three
}a;
a=（enum num）0;
if（a==one）
printf（"将整数值通过强制转换成功赋值给枚举类型变量a\n"）;
else
printf（"赋值失败！\n"）;
return;
}
```

运行结果：

```
将整数值通过强制转换成功赋值给枚举类型变量a
```

在上面的代码中，通过强制转换将整数值0赋值给枚举类型变量a，等价于a=one。如果不使用强制转换，只是简单地通过“=”号进行赋值操作，那么就会出现“error C2440：'='：cannot convert from'const int'to'enum main：num'”错误。

第1章讲解枚举类型时就提到了它和共用体有很大的相似之处，下面通过代码来看看它们之间的区别。

```
#include＜stdio.h＞
void main（）
{
enum compass{
East,
South,
West,
North
}a,b;
```

```
    printf（"枚举类型占用的内存大小为：%d个字节\n"，sizeof（enum
compass））；
    printf（"枚举类型变量a占用的内存大小为：%d个字节\n"，sizeof（a））；
    printf（"枚举类型变量b占用的内存大小为：%d个字节\n"，sizeof（b））；
    printf（"枚举类型变量a的地址为：%d\n"，&a）；
    printf（"枚举类型变量b的地址为：%d\n"，&b）；
    return;
    }
```

运行结果：

```
    枚举类型占用的内存大小为：4个字节
    枚举类型变量a占用的内存大小为：4个字节
    枚举类型变量b占用的内存大小为：4个字节
    枚举类型变量a的地址为：1245052
    枚举类型变量b的地址为：1245048
```

从上面的运行结果可以看出，对于定义的枚举类型，不管它包含多少个枚举常量，所占用的内存大小都为4字节，与所使用的整型类似；对于所定义的枚举类型变量，其占用的内存大小也是4字节，而不同枚举类型的变量所占用的是不同的内存单元，与之前讲解的共用体变量不同，共用体类型的变量占用同一片内存区域。上面定义的枚举类型变量a和枚举类型变量b所占用的是不同的内存区域。

6.2 结构体变量的初始化方法及引用

前面的代码中已经多次用到了结构体，但是并没有细讲结构体变量的初始化方法和如何引用结构体成员，接下来针对这两方面进行介绍。

6.2.1 结构体的初始化

先来看下面的一段代码。

```
#include<stdio.h>
struct stu{
char name[10];
char num[10];
double score;
char sex[4];
int age;
};
void main ()
{
struct stu per={
"小明",
"21009012",
456.9,
"男",
22,
};
printf ("name: %s\nnum: %s\nscore: %lf\nsex: %s\nage: %d\n",
per.name,per.num,per.score,per.sex,per.age) ;
return;
}
```

运行结果：

```
name: 小明
num: 21009012
```

```
score：456.900000
sex：男
age：22
```

在上面的代码中，初始化方法是根据结构体中成员定义的先后顺序进行赋值。如果不知道结构体成员的先后顺序情况，那么就不能采用这种方法了，例如，对于封装好的结构体，在知道结构体成员类型而不知道先后顺序的情况下，一般采用下面的这种初始化方法。

```
#include＜stdio.h＞
struct stu{
char*name;
char*num;
double score;
char*sex;
int age;
};
void main ()
{
struct stu per;
per.name="王梅梅";
per.age=21;
per.sex="女";
per.sex="fd";
per.num="20204";
per.score=765.9;
printf ("name：%s\nnum：%s\nscore：%.3lf\nsex：%s\nage：%d\n",
per.name,per.num,per.score,per.sex,per.age) ;
return;
}
```

运行结果：

```
name：王梅梅
num：20204
score：765.900
sex：fd
age：21
```

采用上面这种方式同样可以实现对结构体变量的初始化。细心的读者会发现一个问题，在上面的代码中把前面结构体中的数组类型变成了指针类型。为什么不采用数组类型呢？如果把上面结构体中的指针类型还原为最初的数组类型，而且还采用上面的初始化方式，会出现什么情况呢？为了重点讲解指针与数组的区别，修改上面的代码如下。

```
#include＜stdio.h＞
struct stu{
char name[10];
};
void main ()
{
struct stu per;
per.name="张晓笑";
printf ("%s\n", per.name) ;
return;
}
```

上面的代码在编译时出现了“error C2440：'='：cannot convert from'char[7]'to'char[10]'”错误，为什么会出现这样的错误呢？在对结构体中的名字进行初始化的过程中，一个汉字占2字节，最后还有一个结束符‘\0’，共占用7字节，是不是只要修改结构体中name成员的数组长度为7就可以了呢？将其修改为7后，再次编译时出现了“error C2106：'='：left operand must be l-value”错误，这是什么原因呢？这种错误归根于对数组的初始化方式不清楚。在讲解字符数组的初始化方式时特地强调过，不能在定义数组后再对数组名进行一次性字符串赋值，而指针可以采用这种方式进行初始化。为了加深理解，在此不妨再通过一段代码对

指针和数组的字符串初始化进行比较。

```
#include＜stdio.h＞
void main（）
{
char*namep;
namep="范德萨";
char name[10]="张欣欣";
//name="张欣欣"；切记不可采用这样的初始化方式
printf（"%s\n"，namep）；
printf（"%s\n"，name）；
return;
}
```

运行结果：

```
范德萨
张欣欣
```

那么，是不是不能将结构体的成员变量定义为数组类型呢？当然不是，如果将结构体的成员变量定义为数组类型，那么可以利用strcpy（）函数对其进行初始化，例如：

```
#include＜stdio.h＞
#include＜string.h＞
struct stu{
char name[10];
char num[10];
double score;
char sex[4];
int age;
};
void main（）
{
struct stu per;
strcpy（per.name，"小明"）；
strcpy（per.num，"21009012"）；
```

```
    strcpy（per.sex，"男"）；
    per.score=456.9;
    per.age=22;
    printf（"name：%s\nnum：%s\nscore：%lf\nsex：%s\nage：%d\n",
per.name,per.num,per.score,per.sex,per.age）；
    return;
    }
```

运行结果：

```
    name：小明
    num：21009012
    score：456.900000
    sex：男
    age：22
```

从上面的运行结果可以看出，如果结构体中存在数组类型的成员，可以利用strcpy（）函数对数组进行初始化。需要注意的是，这种方式实现的只是对字符数组的初始化。

6.2.2 结构体的引用

对于结构体成员的引用，从前面的代码也可以看出，其一般形式为：

```
结构体变量名.结构体成员名
```

如果定义的是结构体数组，那么就把数组中的每个元素作为一个结构体变量来引用结构体成员，其一般形式为：

```
结构体变量名[下标].结构体成员名
```

对于结构体数组的引用，在第1章讲解结构体时已经给出了相应的代码，在此重点介绍结构体指针的使用。先来看下面的一段代码。

```
#include<stdio.h>
#include<string.h>
struct stu{
char name[10];
char num[10];
double score;
char sex[4];
int age;
};
void main ()
{
struct stu per, *p_per;
strcpy (per.name, "小明") ;
strcpy (per.num, "21009012") ;
strcpy (per.sex, "男") ;
per.score=456.9;
per.age=22;
p_per=&per;
```

```
    printf（"通过结构体变量per引用结构体成员\n"）；
    printf（"name：%s\tnum：%s\tscore：%lf\tsex：%s\tage：%d\n",
per.name,per.num,per.score,per.sex,per.age）；
    printf（"\n通过结构体指针p_per引用结构体成员方法一\n"）；
    printf（"name：%s\tnum：%s\tscore：%lf\tsex：%s\tage：%d\n",
（*p_per）.name，（*p_per）.num，（*p_per）.score，（*p_per）.sex，
（*p_per）.age）；
    printf（"\n通过结构体指针p_per引用结构体成员方法二\n"）；
    printf（"name：%s\tnum：%s\tscore：%lf\tsex：%s\tage：%d\n"，p_per-
＞name,p_per-＞num,p_per-＞score,p_per-＞sex,p_per-＞age）；
    return;
    }
```

运行结果：

```
    通过结构体变量per引用结构体成员
    name：小明num：21009012 score：456.900000 sex：男age：22
    通过结构体指针p_per引用结构体成员方法一
    name：小明num：21009012 score：456.900000 sex：男age：22
    通过结构体指针p_per引用结构体成员方法二
    name：小明num：21009012 score：456.900000 sex：男age：22
```

在上面的代码中，通过指针引用得到的结果和直接采用结构体变量引用得到的结果完全相同，这说明可以通过指针引用来替代结构体变量对结构体成员的引用。下面介绍指针引用的两种方法。

方法一：

```
    （*结构体指针变量名）.结构体成员名
```

采用该方法时要注意其中的括号不能省略，因为“*”的优先级低于“.”的优先级。如果没有加括号，以上面的代码为例，在此介绍一种错误情况，“error C2228：left of'.name'must have class/struct/union type”，

其余的错误与此类似。出现这种错误是因为使用p_per引用结构体成员时出错了。p_per并不能够引用结构体成员，它本身并不是一种结构体类型，只是一个保存结构体变量地址的变量，要想通过它来引用结构体成员，必须使用括号先将p_per和*号结合，表示一种结构体类型，等价于p_per所指向的结构体变量，进而成功引用结构体成员。

方法二：

结构体指针变量名->结构体成员名

采用这种方式时不需要再使用*和指针名结合的方式来引用结构体成员，只需要通过指针名直接引用结构体成员即可。需要注意的是，在使用这种方法引用结构体成员的时候，结构体类型的指针必须已经指向保存的结构体类型的变量。接下来看看直接定义的结构体类型的指针变量如何使用。

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
struct stu{
char name[10];
char num[10];
double score;
char sex[4];
int age;
};
void main ()
{
struct stu*p_per1=(struct stu*)malloc(sizeof(struct stu));
strcpy((*p_per1).name,"夏美芳");
strcpy((*p_per1).num,"21009012");
```

```
strcpy（（*p_per1）.sex，"女"）；
    （*p_per1）.score=456.9；
    （*p_per1）.age=22；
   printf（"name：%s\tnum：%s\tscore：%.2lf\tsex：%s\tage：%d\n"，
p_per1->name,p_per1->num,p_per1->score,p_per1->sex,p_per1->age）；
    struct stu*p_per2=（struct stu*）malloc（sizeof（struct stu））；
    strcpy（p_per2->name，"艾美丽"）；
    strcpy（p_per2->num，"21009011"）；
    strcpy（p_per2->sex，"女"）；
    p_per2->score=568.4；
    p_per2->age=20；
    printf（"name：%s\tnum：%s\tscore：%.2lf\tsex：%s\tage：%d\n"，
（*p_per2）.name，（*p_per2）.num，（*p_per2）.score，（*p_per2）.sex，
（*p_per2）.age）；
    return；
    }
```

运行结果：

```
    name：夏美芳num：21009012 score：456.90 sex：女age：22
    name：艾美丽num：21009011 score：568.40 sex：女age：20
```

在上面的代码中，定义了结构体类型的指针变量p_per1和p_per2，采用了两种引用方法对其进行初始化，同时使用了两种方法进行打印输出。在此分别采用不同的引用方法来对结构体的的指针变量进行初始化和引用，主要是为了说明指针变量可以采用两种方法中的任何一种进行引用。需要留意的是，讲解指针时也特别强调过，没有分配内存的指针变量指向的是不可使用的内存区域，所以如果要对指针变量进行初始化，必须先在内存单元中为其分配一块可用的内存区域。

接下来我们再来看看一个综合性的使用，这段代码实现对已经初始化了的结构体数组按照学生的总分进行排名，分数由高到低。

```
#include<stdio.h>
#include<stdlib.h>
#define N 4
struct stu{
char*name;
char*num;
double score;
char*sex;
int age;
}per[N]={
{"张欣欣", "210025", 345.23, "女", 20},
{"范德萨", "210026", 386.56, "男", 22},
{"王大鹏", "210027", 312.45, "男", 19},
{"郭丽丽", "210028", 364.25, "女", 24}
};
void sort (struct stu*p_per,int n)
{
int i,j, k;
struct stu temp;
for (i=0; i<n-1; i++)
{
k=i;
for (j=i+1; j<n; j++)
{
if (p_per[j].score>p_per[k].score)
k=j;
}
if (k! =i)
{
temp=p_per[i];
p_per[i]=p_per[k];
p_per[k]=temp;
}
}
return;
}
void print (struct stu*p_per,int n)
{
int i;
for (i=0; i<n; i++)
{
printf ("name: %s\tnum: %s\tscore: %.2lf\tsex: %s\tage: %d\n",
p_per->name,p_
per->num,p_per->score,p_per->sex,p_per->age) ;
p_per++;
}
return;
```

```
}
void main ()
{
sort (per,N) ;
print (per,N) ;
return;
}
```

运行结果：

```
name: 范德萨num: 210026 score: 386.56 sex: 男age: 22
name: 郭丽丽num: 210028 score: 364.25 sex: 女age: 24
name: 张欣欣num: 210025 score: 345.23 sex: 女age: 20
name: 王大鹏num: 210027 score: 312.45 sex: 男age: 19
```

对比上面的初始化顺序和打印出来的结果可知，我们成功地实现了代码的功能，下面对其中的重点进行分析。在对结构体数组进行初始化的时候，是按照声明的先后顺序进行的，所以此时要注意结构体成员的先后顺序；在接下来对结构体数组进行排序的时候，传递的是结构体数组的首地址，然后在sort（）函数中对其进行排序，由于采用的是传址方式，因此在sort（）函数中对结构体指针p_per所指向的空间进行的交换操作就是在起始部分对结构体数组per所做的操作；接下来通过print（）函数来打印此时结构体数组中的信息，得到的是经过排序后的数组，而不再是之前初始化的数组。

6.3 结构体字节对齐详解

“字节对齐”这个概念在第1章中已经初步介绍过，这里主要介绍结构体在内存中是如何进行字节对齐操作的。

1.sizeof操作符

在讲解结构体的字节对齐之前，不得不提的一个操作符是sizeof，其功能为返回一个对象或类型所占用的内存大小。可能有人会将sizeof操作符当成函数，为了加深sizeof是一个操作符的概念，来看下面的代码。

```
#include<stdio.h>
void print()
{
printf("hello world!\n");
return;
}
void main()
{
printf("%d\n", sizeof(print()));
return;
}
```

对于上面这段代码，在Linux环境下采用gcc编译是没有任何问题的，返回值为void类型，其输出值为1，而在VC++6.0环境下运行就会出现“error C2070：illegal sizeof operand”，从错误提示可知，sizeof是一个操作符，并非函数。对于sizeof操作符，它的使用方式有如下几种。

```
sizeof（object）；//sizeof（对象）；
sizeof（type_name）；//sizeof（类型）；
sizeof object；//sizeof对象；这种写法通常不会在代码中使用，所以很少见到。
```

接下来通过一段代码来演示以上3种方式的使用。

```
#include＜stdio.h＞
void main（）
{
int i;
printf（"sizeof（i）：\t%d\n"，sizeof（i））；
printf（"sizeof（4）：\t%d\n"，sizeof（4））；
printf（"sizeof（4+2.5）：\t%d\n"，sizeof（4+2.5））；
printf（"sizeof（int）：\t%d\n"，sizeof（int））；
printf（"sizeof 5：\t%d\n"，sizeof 5）；
return;
}
```

运行结果：

```
sizeof（i）：4
sizeof（4）：4
sizeof（4+2.5）：8
sizeof（int）：4
sizeof 5：4
```

代码中给出了每种方式的使用方法，注意，sizeof在计算对象占用内存大小时也转换成对对象类型的计算，也就是说，对于同种类型的不同对象，执行了sizeof后的值都是一样的。从给出的代码中也可以看，sizeof可以对一个表达式求值，编译器根据表达式的最终结果类型来确定大小，但是一般不会对表达式进行计算，当表达式为函数时，并不执行函数体，例如：

```
#include<stdio.h>
void main ()
{
int i;
i=0;
printf ("sizeof (i+++1.23) : %d\n", sizeof (i+++1.23) ) ;
printf ("sizeof (float) : \t%d\n", sizeof (float) ) ;
printf ("使用过sizeof操作符之后的i值为: %d\n", i) ;
return;
}
```

运行结果：

```
sizeof (i+++1.23) : 8
sizeof (float) : 4
使用过sizeof操作符之后的i值为: 0
```

在上面的运行结果中，通过sizeof操作符求得的i++与1.23相加之后的值所占用的内存大小为8而不是4，说明i++与1.23相加后被隐式转换为double类型，而不是float类型。接下来的i值与初始值一样，说明在操作符中的i++并没有被执行，也就是说，sizeof操作符不会对表达式进行计算。再来看看下面的一段代码：

```
#include<stdio.h>
int print ()
{
printf ("Hello bigloomy! ") ;
return 0;
}
void main ()
{
printf ("sizeof (print () ) : \t%d\n", sizeof (print () ) ) ;
return;
}
```

运行结果：

```
sizeof（print（））：4
```

运行结果并没有输出“Hello bigloomy！”信息，即print（）函数并没有被调用，这表明当操作符中的表达式为函数时，函数体并不会被执行，仅得到函数的返回类型所占用的内存大小。

2.offsetof宏的实现

为了能够更好地理解字节对齐的概念，接下来了解一下Linux内核链表里的一个宏。

```
#define offsetof（TYPE,MEMBER）（（size_t）&（（TYPE*）0）->MEMBER）
```

这个宏大致可以分为以下4部分：

通过（TYPE*）0）将0地址强制转换为TYPE结构类型的指针；

通过（（TYPE*）0）->MEMBER访问TYPE结构中的MEMBER数据成员；

通过&（（（TYPE*）0）->MEMBER）取出TYPE结构中的数据成员MEMBER的地址；

通过（size_t）（&（（（TYPE*）0）->MEMBER））将结果转

换为size_t类型。

宏offsetof的巧妙之处在于将0地址强制转换为TYPE结构类型的指针，TYPE结构以内存空间首地址0作为起始地址，成员地址自然为偏移地址。有的读者可能会想是不是非要用0呢？当然不是，这里仅仅是为了使计算简便。也可以使用其他值，只是算出来的结果还要减去该值才是偏移地址。看下面的代码：

```
#include<stdio.h>
#define offsetof(TYPE,MEMBER) ((size_t)&((TYPE*)4)->MEMBER)
typedef struct stu1
{
int a;
int b;
}stu1;
void main()
{
printf("offsetof(stu1, a): \t%d\n", offsetof(stu1, a)-4);
printf("offsetof(stu1, b): \t%d\n", offsetof(stu1, b)-4);
return;
}
```

运行结果：

```
offsetof(stu1, a): 0
offsetof(stu1, b): 4
```

为了加深读者的印象，再来看图6-1，在上面的代码中没有使用0，而是用4，因为a是结构体中的第一个成员变量，其地址就是结构体的起始地址，其偏移量为0。所以在计算结构体成员变量偏移地址时需要减

去4。当然，在实际使用中通常都用0作为起始地址。通过分析可以看出，该offset宏可以求出结构体中每个成员的偏移地址。

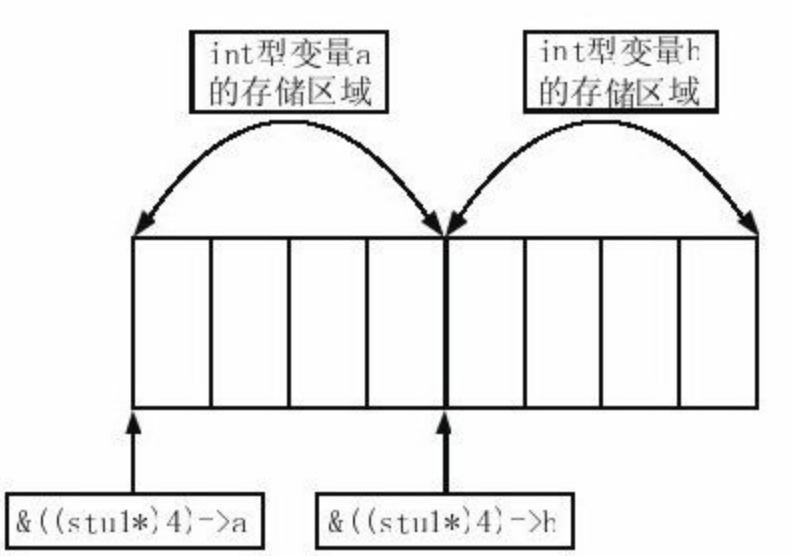

图　6-1　结构体stu1的内存结构

3.结构体字节对齐

接下来就可以借助宏offsetof来查看结构体中各个成员的偏移地址和结构体所占内存的大小，进而理解结构体是如何实现字节对齐操作的，看下面的一段代码。

```
#include<stdio.h>
#define offsetof(TYPE,MEMBER) ((size_t)&((TYPE*)0)->MEMBER)
typedef struct stu1
{
int a;
char b;
int c;
}stu1;
void main()
{
printf("offsetof(stu1, a): \t%d\n", offsetof(stu1, a));
```

```
printf ("offsetof (stu1, b) : \t%d\n", offsetof (stu1, b) ) ;
printf ("offsetof (stu1, c) : \t%d\n", offsetof (stu1, c) ) ;
printf ("sizeof (stu1) : \t%d\n", sizeof (stu1) ) ;
return;
}
```

运行结果：

```
offsetof (stu1, a) : 0
offsetof (stu1, b) : 4
offsetof (stu1, c) : 8
sizeof (stu1) : 12
```

对于上面的运行结果，对字节对齐不了解的读者可能会疑惑，c的偏移量怎么会是8呢？结构体的大小怎么会是12呢？读者可能会这样理解：c的偏移量是sizeof（int）+sizeof（char）=5，结构体stu1占用的内存大小应该是sizeof（int）+sizeof（char）+sizeof（int）=9。通过如图6-2所示的stu1的内存结构可以知道，编译器对变量存储进行了一个特殊处理。为了提高CPU的存储速度，编译器对一些变量的起始地址做了对齐处理。在默认情况下，编译器规定各成员变量存放的起始地址相对于结构的起始地址的偏移量，必须为该变量的类型所占用的字节数和编译器编译过程中采用的字节对齐数两者中最小值的整数倍。

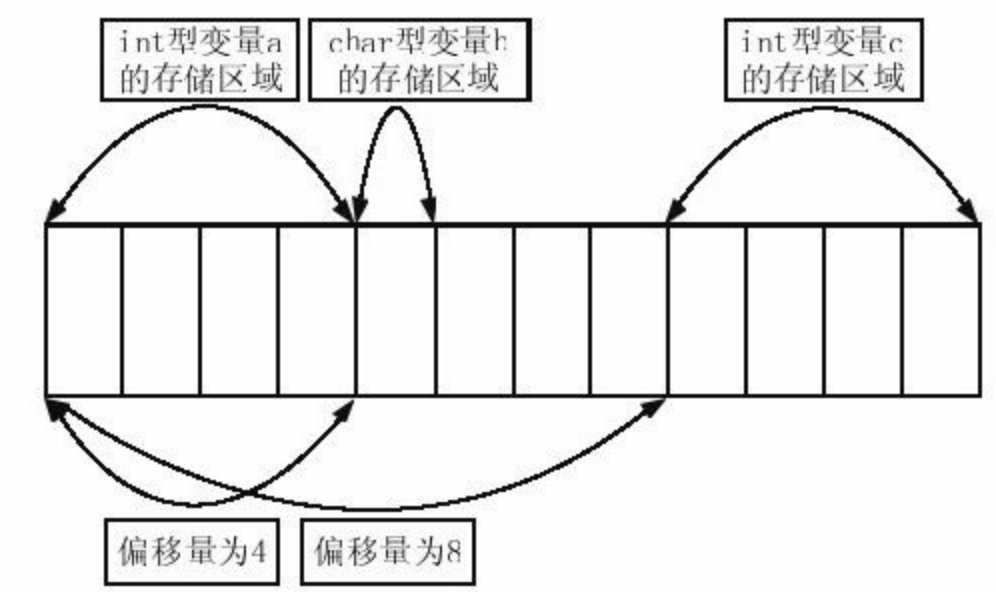

图　6-2　结构体stu1的内存结构

现在来分析前面的代码，假定a的起始地址为0，它占用了4字节，接下来的空闲地址就是4，是1的倍数，满足要求，所以b存放的起始地址是4，占用一个字节，接下来的空闲地址为5。而c是int变量，占用4字节，同时需要注意的是，编译器默认按照结构体中占有内存最大的类型所占用的字节数进行字节对齐。在此结构体中占用内存最大的为整型，占用4字节，所以在此取两者的最小值4，5不是4的整数倍，所以向后移动，找到离5最近的8作为存放c的起始地址，c也占用4字节，最后结构体的大小为12。需要注意的就是，变量b后面3字节的存储空间是由编译器自动填充的，其中没有存储任何有用的信息。

如果遇到结构体的嵌套，那么又该如何分析结构体的字节对齐呢？再来看下面的代码。

```
#include＜stdio.h＞
struct stu1
{
char array[7];
int a;
}stu1;
struct stu2
{
double a;
char b;
}stu2;
struct stu3
{
stu1 s;
char str;
}stu3;
struct stu4
{
stu2 s;
char str;
}stu4;
void main ()
{
printf ("sizeof (stu1) : \t%d\n", sizeof (stu1) ) ;
printf ("sizeof (stu2) : \t%d\n", sizeof (stu2) ) ;
printf ("sizeof (stu3) : \t%d\n", sizeof (stu3) ) ;
printf ("sizeof (stu4) : \t%d\n", sizeof (stu4) ) ;
return;
}
```

运行结果：＜/p＞

```
sizeof (stu1) : 12
sizeof (stu2) : 16
sizeof (stu3) : 16
sizeof (stu4) : 24
```

在上面的运行结果中，stu1和stu2所占内存的分别为12字节和16字节，对这两者的分析与前面相同，在此重点讲解stu3和stu4。在默认情况下，结构体采用该结构体中占用内存最大的类型所占的字节数作为字

节对齐方式，但是在stu3中定义的stu1结构体类型的变量s占用16字节，而stu3并不是按照16字节进行对齐的，而是采用4字节对齐，这是因为stu1和stu3中占用内存最大的是int型变量，占用4字节。因此在分析结构体字节对齐方式时需要将结构体分解为“原子类型”，如int、double、char、float、short等，而不是自定义的结构体类型。找出分解出来的“原子类型”中占用内存最大的类型，将其占用的内存值作为结构体的默认字节对齐值。

在stu4中定义了stu2类型的结构体变量s，按照上面的方法先对stu2进行分解。分解出来的类型有double、char,stu4中还有char类型，其中占用内存最大的是double类型，占用内存大小为8字节，由此可知，stu4采用8字节对齐。由于stu4中的stu2结构体类型变量s所占用的内存大小为16，而接下来定义了一个char类型的str变量，其偏移地址为16，占用一个字节，此时stu4占用的内存大小为17，不是字节对齐数8的整数倍，所以在stu4占用的内存的最后添加7字节的空间，使其占有内存大小为24。值得注意的是，添加的内存并没有使用，由编译器自动填充，没有存放任何有意义的内容。

在结构体的嵌套中，不管遇到多少层的嵌套，都可以按照这种分解方法，对结构体进行逐层分解，再根据分解出来的“原子类型”分析结构体的字节对齐方式，看下面的代码。

```
#include＜stdio.h＞
```

```
struct stu1
{
double a;
char b;
};
struct stu2
{
struct stu1 s;
int c;
};
struct stu3
{
struct stu2 s;
char str;
};
void main ()
{
printf ("sizeof (stu1) : \t%d\n", sizeof (struct stu1) ) ;
printf ("sizeof (stu2) : \t%d\n", sizeof (struct stu2) ) ;
printf ("sizeof (stu3) : \t%d\n", sizeof (struct stu3) ) ;
return;
}
```

运行结果：

```
sizeof (stu1) : 16
sizeof (stu2) : 24
sizeof (stu3) : 32
```

在上面的代码中并没有人为指定字节对齐数，所以编译器按照默认方式来处理，在stu1中占用内存最大的是double类型，占8字节，所以编译器进行的是8字节对齐。由于编译器最终占用内存的大小必须是8的整数倍，因此stu1在内存中占用16字节；而在stu2中，由于其中有stu1，需要先对stu1进行分解，stu1中的“原子类型”有double、char,stu2中的“原子类型”是int，因此Stu2中占用内存最大的是double类型，因此同样要进行

8字节对齐；stu3中包含stu2，而stu2中又包含stu1，逐一进行分解，stu1和stu2的分解同前，stu3的“原子类型”只有char，所以同样进行8字节对齐，从而得出stu3占用的内存大小为32字节。

在上面的代码中均是使用默认值来进行字节对齐的，当然，也可以使用预编译指令“#pragma pack（value）”来告诉编译器使用指定的字节对齐值来取代默认字节对齐值。接下来看下面的一段代码。

```
#include＜stdio.h＞
#pragma pack（1）/*指定按1字节对齐*/
typedef struct stu1
{
char str[10];
double b;
}stu1;
#pragma pack（）/*取消指定对齐，恢复默认对齐*/
typedef struct stu2
{
char str[10];
double b;
}stu2;
void main（）
{
printf（"sizeof（stu1）：\t%d\n"，sizeof（stu1））；
printf（"sizeof（stu2）：\t%d\n"，sizeof（stu2））；
return;
}
```

运行结果：

```
sizeof（stu1）：18
sizeof（stu2）：24
```

在上面的代码中，定义了两个结构体成员完全相同的结构体类型，

但是采用了不同的字节对齐方式，所以最终两个结构体占用的内存大小并不相同。通过图6-3可以看出，由于在结构体stu1中指定采用1字节对齐，使其中的成员b所占用的偏移量为1的整数倍，因此stu1占用内存大小为18字节；在结构体stu2中取消了设定的字节对齐方式，采用默认的按照其中占用内存最大的字节数作为字节对齐方式，在此为double类型，占用8字节，所以在此stu2中b的偏移量是8的整数倍，从而使stu2所占用的内存大小为24。编译器在编译的过程中会在str字符数组的后面添加存储单元，使结构体中成员变量b的偏移量为8的整数倍，即16，而b占有内存大小为8字节，stu2存储b变量之后的内存大小为24，刚好为8的整数倍，无需在后面添加任何存储单元，所以stu2占用的内存大小为24字节。

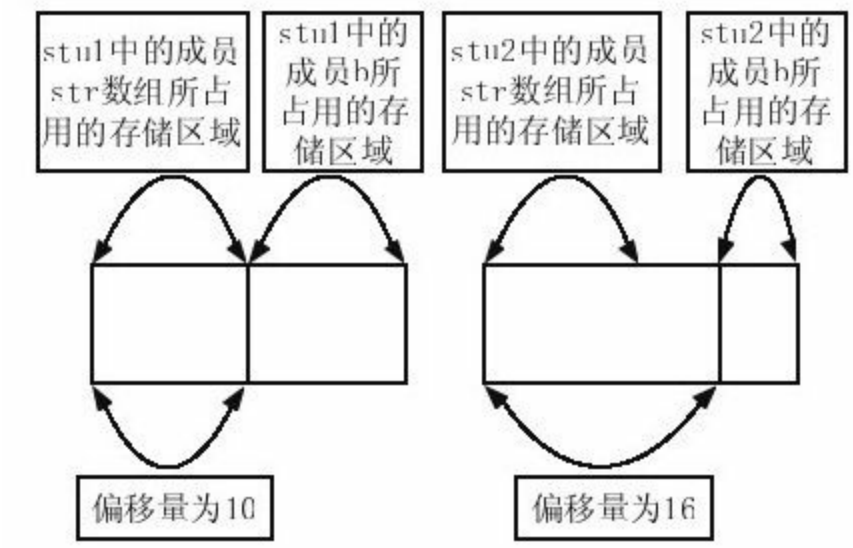

图　6-3　结构体stu1和stu2的内存结构

在上面的代码中指定的字节对齐数小于stu1结构体默认的字节对齐

数8，如果指定的字节对齐数大于结构体默认的字节对齐数，会怎么样呢？看下面的代码。

```
#include＜stdio.h＞
#pragma pack（1）/*指定按1字节对齐*/
struct stu1
{
char str[10];
double b;
};
#pragma pack（）/*取消指定对齐，恢复默认对齐*/
struct stu2
{
char str[10];
double b;
};
#pragma pack（16）/*指定按16字节对齐*/
struct stu3
{
char str[10];
double b;
};
void main（）
{
printf（"sizeof（stu1）：\t%d\n"，sizeof（struct stu1））；
printf（"sizeof（stu2）：\t%d\n"，sizeof（struct stu2））；
printf（"sizeof（stu3）：\t%d\n"，sizeof（struct stu3））；
return;
}
```

运行结果：

```
sizeof（stu1）：18
sizeof（stu2）：24
sizeof（stu3）：24
```

在上面的代码中，定义了3个内部成员完全相同的结构体类型，在stu1中指定采用1字节对齐，得到的stu1占用内存大小为18字节；而在

stu2中按照默认方式进行字节对齐，得到的stu2占用内存大小为24字节；在stu3中指定采用16字节对齐，得到的运行结果与stu2完全相同，这是因为编译器在处理的过程中所取得的字节对齐方式是指定的字节对齐数和默认字节对齐数两者中的最小值。

6.4 共用体变量的初始化方法及成员的引用

前面特地强调过，共用体和结构体的最大区别在于共用体中的成员共用同一片内存区域，所有成员变量均从同一起始地址开始，仅仅是占用的内存大小区域不一致，其占用内存大小由成员变量占用内存最大的类型决定。如果共用体中的成员含有结构体类型的变量，就不存在对结构体的“原子分解”了，而是要把结构体所占用的内存大小视为一个整体，如：

```
#include<stdio.h>
struct fe
{
double a;
char b;
};
union st
{
struct fe f;
double a;
int b;
char c;
};
void main ()
{
union st d;
printf ("sizeof (d) : %d\n", sizeof (d) ) ;
return;
}
```

运行结果：

```
sizeof (d) : 16
```

按照前面对结构体的分析可知，结构体fe占用的内存大小为16字节，共用体类型的成员类型分别为struct fe、double、int、char，其中占用内存最大的为struct fe，占用16字节，所以共用体在内存中占用的内存大小为16字节。

如何对共用体变量进行初始化和引用呢？接下来通过下面的一段代码来展示对共用体变量的初始化和引用。

```
#include＜stdio.h＞
union st
{
double a;
int b;
char c;
};
void main ()
{
union st d;
d.a=1.23;
printf ("d.a=%.2lf\n", d.a) ;
d.b=98;
printf ("d.b=%.d\n", d.b) ;
d.c='a';
printf ("d.c=%c\n", d.c) ;
return;
}
```

运行结果：

```
d.a=1.23
d.b=98
d.c=a
```

从上面的代码中可以看出，对于共用体类型的一般变量，其成员的

引用方式为：

```
共用体类型名.共用体成员名
```

从其引用方式可以看出，这与前面讲解的结构体很类似，所以在此不再赘述。需要注意的一点是，共用体成员共享同一片内存单元，可以通过下面这段代码证实这一点。

```
#include＜stdio.h＞
union st
{
double a;
int b;
char c;
};
void main ()
{
union st d;
printf ("&d=%d\n", &d) ;
printf ("&d.c=%d\n", &d.c) ;
printf ("&d.a=%d\n", &d.a) ;
printf ("&d.b=%d\n", &d.b) ;
printf ("sizeof (d.a) =%d\n", sizeof (d.a) ) ;
printf ("sizeof (d.b) =%d\n", sizeof (d.b) ) ;
printf ("sizeof (d.c) =%d\n", sizeof (d.c) ) ;
return;
}
```

运行结果：

```
&d=1245048
&d.c=1245048
&d.a=1245048
&d.b=1245048
sizeof (d.a) =8
sizeof (d.b) =4
sizeof (d.c) =1
```

在上面的运行结果中，共用体变量的起始地址为1245048，而共用体中的所有成员的起始地址均为1245048，共用体成员所占用的内存大小并不相同，正是由于共用体的这种特殊性，在初始化时不能像对结构体那样对其成员一次性初始化，但是可以对共用体中的第一个成员进行初始化，如：

```
#include<stdio.h>
union st
{
double a;
int b;
char c;
};
void main ()
{
union st d={3};
printf ("d.a=%.2lf\n", d.a) ;
printf ("d.b=%d\n", d.b) ;
return;
}
```

运行结果：

```
d.a=3.00
d.b=0
```

上面代码中的初始化方式是对共用体中的第一个成员进行初始化，所以通过对第一个成员的引用能够得到初始化的值，而对第二成员变量的引用得到的不是所需的值。

接下来看指向共用体的指针是如何对其成员变量进行引用的，代码如下：

```
#include<stdio.h>
union st
{
double a;
int b;
char c;
};
void main ()
{
union st d, *per;
d.a=1.23;
per=&d;
printf ("d.a=%.2lf\tper->a=%.2lf\t (*per) .a=%.2lf\n", d.a,per->
a, (*per) .a) ;
d.b=99;
printf ("d.b=%d\tper->b=%d\t (*per) .b=%d\n", d.b,per->b,
(*per) .b) ;
d.c='g';
printf ("d.c=%c\tper->c=%c\t (*per) .c=%c\n", d.c,per->c,
(*per) .c) ;
return;
}
```

运行结果：

```
d.a=1.23 per->a=1.23 (*per) .a=1.23
d.b=99 per->b=99 (*per) .b=99
d.c=g per->c=g (*per) .c=g
```

在上面的代码中用了3种方法引用共用体中的成员变量，一种是前面讲解的以一般共用体类型的形式进行引用，另两种是采用指针的方式进行引用。

指针引用方法一：

```
（*共用体指针名）.共用体成员名
```

与前面的结构体类型一样，在使用这种方法引用共用体成员的时候须考虑优先级的关系，其中的括号一定不能少。

指针引用方法二：

```
共用体指针名->共用体成员名
```

这种方法简单地采用共用体指针名和“->”结合的方式来引用共用体成员。

6.5　传统链表的实现方法及注意事项

可能有不少人会疑惑链表还分什么传统链表和非传统链表吗？在此简单解释一下，这里所说的传统链表，是相对下一节所要讲解的Linux内核中的双向循环链表而言的。由于Linux内核中双向循环链表的特殊性，在此把以往在C语言中所学习的链表统称为传统链表。

我们知道，数组是由数组元素所组成的，链表同样是由链表元素所组成的，但是链表元素的特殊性在于，它是由不连续的内存单元块按照一种逻辑顺序所组成的数据结构。链表中的每个元素被称为结点，所以结点是构成链表的最小单元，结点中包含两部分信息，一部分是在结点中存储的数据信息，另一部分是相邻结点的地址信息。

如图6-4所示为单向链表，在单向链表中，结点之间的指向是单向的，其中的结点由两部分组成，一部分是数据域，用来存储数据信息；另一部分是指针域，用来存储下一个结点的地址。需要注意的是，在单向链表中有一个头结点，它仅含有指针域，没有数据域；而在链表的尾结点中将指针域赋值为NULL。

图　6-4　单向链表

如果是单向循环链表，那么尾结点的指针域就不再为NULL，而是

与头结点的指向相同，指向链表的第一个结点，如图6-5所示。

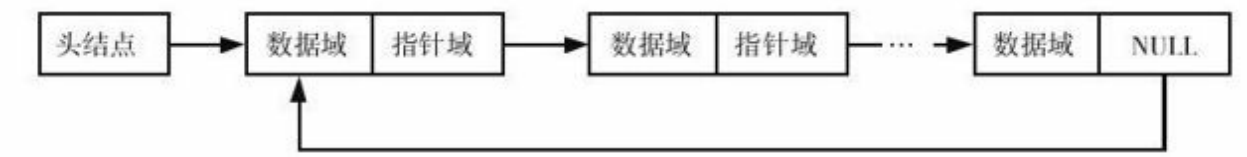

图　6-5　单向循环链表

如图6-6所示为双向链表，双向链表与单向链表的不同之处在于结点多了一个指针域，在单向链表中只有一个指针域用来存储后一个结点的地址信息，而双向链表中多了一个指针域用来存储前一个结点的地址。在此，将保存前一个结点地址信息的指针域称为前指针域，将保存后一个结点地址信息的指针域称为后指针域。这两个指针域使得通过当前结点不仅可以访问后一个结点，还可以访问前一个结点。注意，双向链表的尾结点的后指针域为NULL。

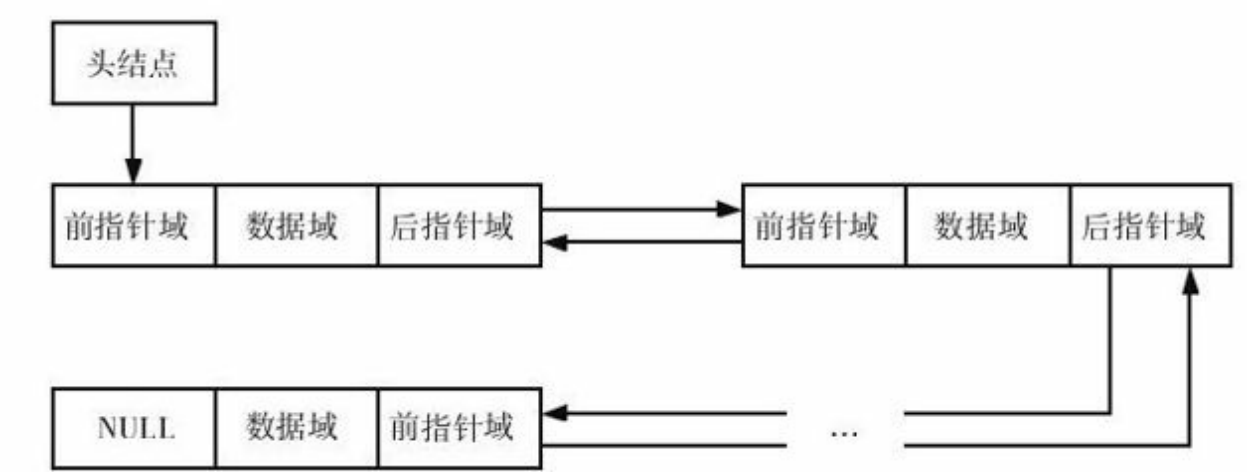

图　6-6　双向链表

如果是双向循环链表，那么尾结点的后指针域不再为NULL，如图

6-7所示，此时尾结点的后指针域指向的是头结点所指向的结点。

链表的创建大致可以分为以下几步：

1）创建链表头。

2）创建结点。

3）将链表头指向链表的第一个结点。

4）继续创建结点，同时将结点加入到链表中适当的位置。

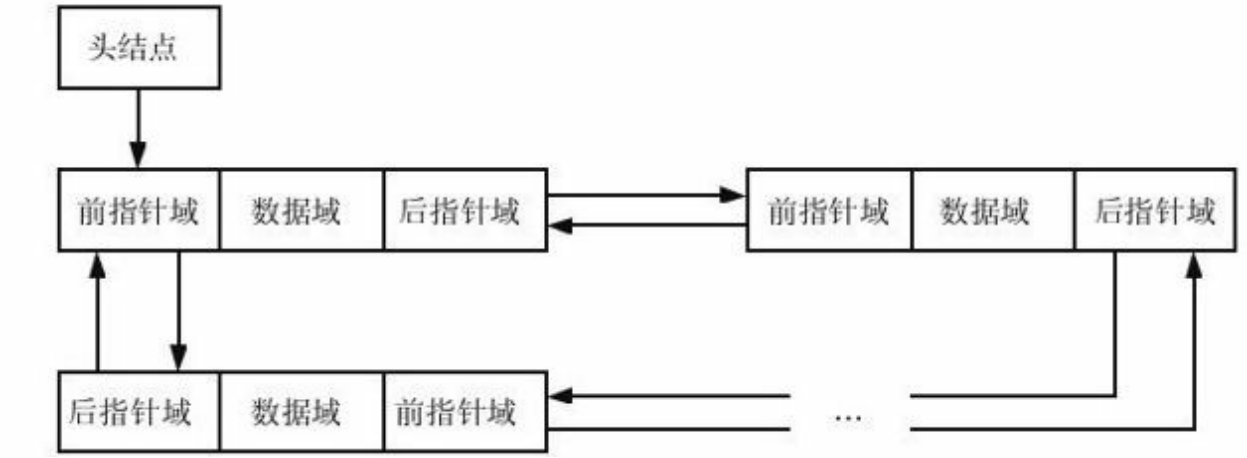

图　6-7　双向循环链表

接下来通过上面的步骤创建一个单链表。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define N 3
struct Stu
{
```

```
    struct Stu*next;
    char name[10];
    int score;
    };
    struct Stu_head
    {
    struct Stu*head;
    };
    struct Stu_head*head_create ()
    {
    struct Stu_head*single_link;
    if ( (single_link= (struct Stu_head*) malloc (sizeof (struct
Stu_head) ) ) ==NULL)
    {
    printf ("分配空间失败! ") ;
    exit (0) ;
    }
    if (single_link! =NULL)
    {
    single_link->head=NULL;
    }
    return single_link;
    }
    struct Stu*node_create ()
    {
    struct Stu*node;
    if ( (node= (struct Stu*) malloc (sizeof (struct Stu) ) ) ==NULL)
    {
    printf ("分配空间失败! ") ;
    exit (0) ;
    }
    if (node! =NULL)
    {
    node->next=NULL;
    printf ("请输入学生的姓名、成绩: ") ;
    scanf ("%s%d", &node->name, &node->score) ;
    }
    return node;
    }
    void node_append (struct Stu_head*single_link)
    {
    struct Stu*node=NULL;
    struct Stu*cursor=NULL;
    node=node_create () ;
    if (single_link->head==NULL)
    {
    single_link->head=node;
    }
```

```
else
{
cursor=single_link->head;
while (cursor! =NULL&&cursor->next! =NULL)
{
cursor=cursor->next;
}
cursor->next=node;
}
return;
}
void node_print (struct Stu_head*single_link)
{
struct Stu*iter=single_link->head;
while (iter! =NULL)
{
printf ("%s的成绩为: %d\n", &iter->name,iter->score) ;
iter=iter->next;
}
return;
}
void node_destroy (struct Stu*node)
{
if (node! =NULL)
{
node->next=NULL;
free (node) ;
}
return;
}
void single_link_destroy (struct Stu_head*single_link)
{
struct Stu*iter=single_link->head;
struct Stu*next=NULL;
while (iter! =NULL)
{
next=iter->next;
node_destroy (iter) ;
iter=next;
}
single_link->head=NULL;
free (single_link) ;
return;
}
int main ()
{
int i;
struct Stu_head*single_head=head_create () ;
```

```
for（i=0；i＜N；i++）node_append（single_head）；
node_print（single_head）；
single_link_destroy（single_head）；
return 0；
}
```

运行结果：

```
请输入第1个人的姓名、成绩：张晓燕236
请输入第2个人的姓名、成绩：王大鹏369
请输入第3个人的姓名、成绩：方小军458
张晓燕的成绩为236
王大鹏的成绩为369
方小军的成绩为458
```

在上面的代码中，把结点的信息分为两部分，一部分是数据域，即学生姓名name和成绩score，另一部分是指针域next，其中保存的是下一个结点的地址。按照上面创建链表的操作步骤，先通过head_create（）函数创建一个链表头，接下来通过node_create（）函数创建结点，创建结点的操作在node_append（）函数中进行，该函数的功能是将创建好的结点插入到链表的末端，作为链表的末端结点。注意，这里需要将链表头指向第一个创建的结点。

为了验证是否成功地实现链表的创建，最后通过一个node_print（）函数打印出每个结点所携带的信息，输出结果表明已经成功实现了链表的创建。

由于结点都是动态申请的，因此最后还要对每个结点申请的内存空间进行释放。释放动态分配的结点空间分为两部分，一部分是含信息结

点的释放，另一部分是链表头结点的释放。先调用single_link_destroy（）函数，在该函数中通过一个while循环来判断当前结点是否为有效的链表结点，如果是，先将链表的指针域赋值为NULL，然后释放该链表结点。释放完所有的链表结点之后退出循环体，要注意链表头同样是动态分配的，所以最后也需要通过free（）函数来释放为其分配的内存空间。

上面代码是在单向链表的末端插入结点，在头结点插入的方法与尾结点类似，在此介绍在两个普通结点之间插入新结点的方法，如图6-8所示。

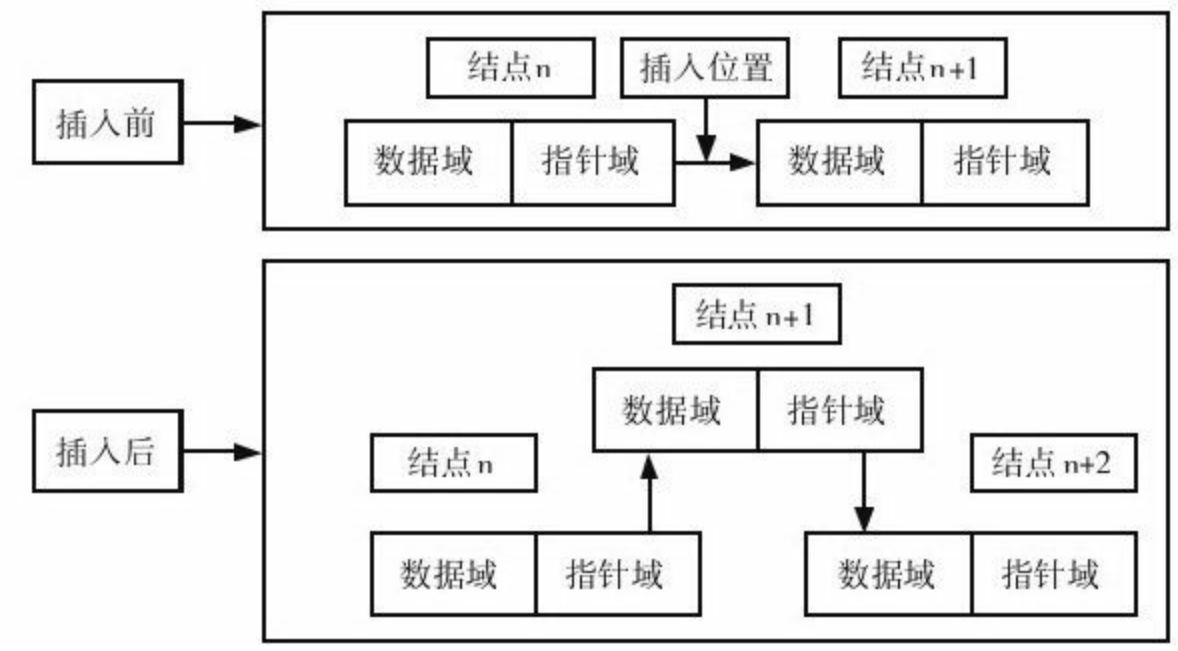

图　6-8　单向链表中结点的插入

注意插入结点前后的变化，插入前，结点n的指针域指向结点n+1，

插入之后，结点n的指针域指向的是新插入结点，而原来的结点n+1变为结点n+2，新加入的结点的指针域指向加入新结点前的结点n+1，这样就实现了在单向链表中插入结点。

接下来通过图6-9来说明在单向链表中两个普通结点之间删除结点的方法。

从图6-9中可以看到删除前后链表的变化，删除结点n+1后，原来链表中的结点n+2变为结点n+1，结点n的指针域指向了原结点n+2。需要注意的是，要将删除的结点的指针域赋值为NULL，同时，如果这个结点是动态分配的结点，那么还应该采用free（）函数将其分配的空间释放掉。下面修改上面的代码来实现插入结点和删除结点的操作。

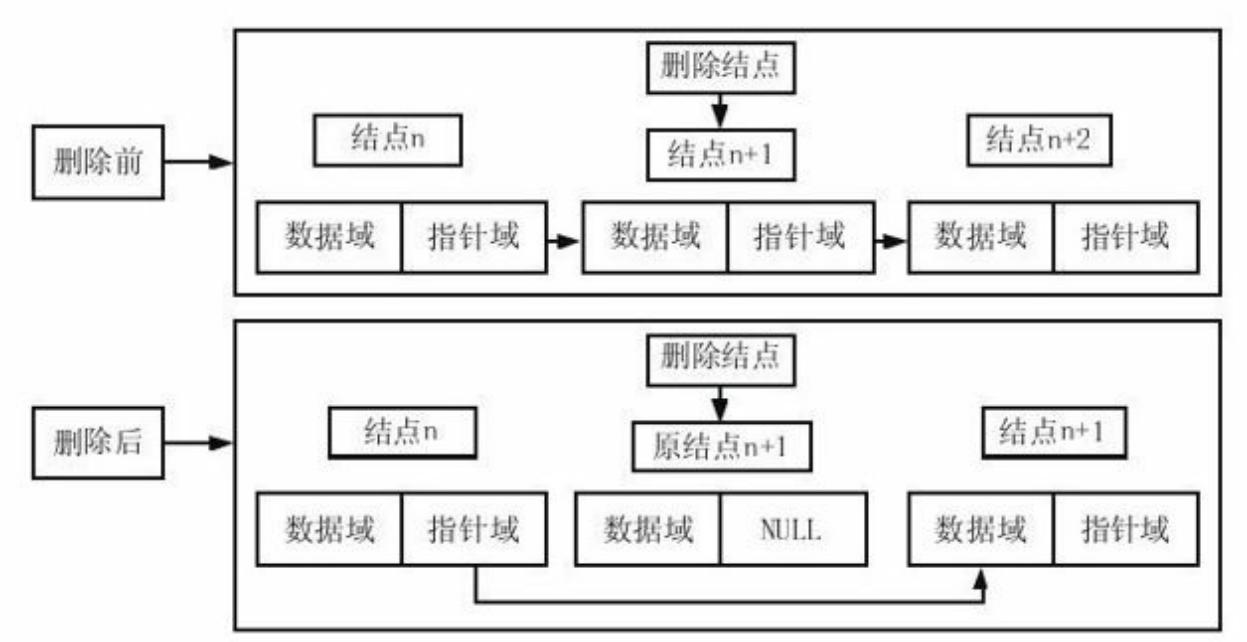

图　6-9　单向链表中结点的删除

```
#include＜stdio.h＞
```

```
#include<stdlib.h>
#include<string.h>
#define N 3
struct Stu
{
struct Stu*next;
char name[10];
int score;
};
struct Stu_head
{
struct Stu*head;
};
struct Stu_head*head_create()
{
struct Stu_head*single_link;
if((single_link=(struct Stu_head*)malloc(sizeof(struct
Stu_head)))==NULL)
{
printf("分配空间失败!");
exit(0);
}
if(single_link!=NULL)
{
single_link->head=NULL;
}
return single_link;
}
struct Stu*node_create()
{
struct Stu*node;
if((node=(struct Stu*)malloc(sizeof(struct Stu)))==NULL)
{
printf("分配空间失败!");
exit(0);
}
if(node!=NULL)
{
node->next=NULL;
printf("请输入学生的姓名、成绩:");
scanf("%s%d",&node->name,&node->score);
}
return node;
}
void node_append(struct Stu_head*single_link)
{
struct Stu*node=NULL;
struct Stu*cursor=NULL;
```

```
node=node_create（）；
if（single_link->head==NULL）
{
single_link->head=node;
}
else
{
cursor=single_link->head;
while（cursor！=NULL&&cursor->next！=NULL）
{
cursor=cursor->next;
}
cursor->next=node;
}
return;
}
struct Stu*node_search（struct Stu_head*single_link,char*name）
{
struct Stu*pos=NULL;
pos=single_link->head;
while（pos！=NULL）
{
if（strcmp（pos->name,name）==0）
return pos;
else
pos=pos->next;
}
if（pos==NULL）
{
printf（"没有查找到该数据！"）；
exit（0）；
}
return NULL;
}
void node_insert（struct Stu*pos）
{
struct Stu*new_node;
printf（"插入信息："）；
new_node=node_create（）；
new_node->next=pos->next;
pos->next=new_node;
return;
}
void node_print（struct Stu_head*single_link）
{
struct Stu*iter=single_link->head;
while（iter！=NULL）
{
```

```
printf（"%s的成绩为：%d\n"，＆iter->name,iter->score）；
iter=iter->next;
}
return;
}
void delete_node（struct Stu_head*single_link,struct Stu*node）
{
struct Stu*node_prev，*tmp;
tmp=single_link->head;
if（tmp==node）
single_link->head=tmp->next;
else
{
while（tmp! =node）
{
node_prev=tmp;
tmp=tmp->next;
}
node_prev->next=tmp->next;
}
tmp->next=NULL;
free（tmp）；
return;
}
void node_destroy（struct Stu*node）
{
if（node! =NULL）
{
node->next=NULL;
free（node）；
}
return;
}
void single_link_destroy（struct Stu_head*single_link）
{
struct Stu*iter=single_link->head;
struct Stu*next=NULL;
while（iter! =NULL）
{
next=iter->next;
node_destroy（iter）；
iter=next;
}
single_link->head=NULL;
free（single_link）；
return;
}
int main（）
```

```
{
int i;
char name[10];
struct Stu*search_node;
struct Stu_head*single_head=head_create();
for(i=0; i<N; i++)
node_append(single_head);
node_print(single_head);
printf("请输入你要查找结点中的人名：");
scanf("%s", name);
search_node=node_search(single_head,name);
node_insert(search_node);
printf("\n插入结点后\n");
node_print(single_head);
delete_node(single_head,search_node);
printf("\n删除结点后\n");
node_print(single_head);
single_link_destroy(single_head);
return 0;
}
```

运行结果：

```
请输入学生的姓名、成绩：王小欢78
请输入学生的姓名、成绩：张晓笑88
请输入学生的姓名、成绩：王大鹏99
王小欢的成绩为：78
张晓笑的成绩为：88
王大鹏的成绩为：99
请输入你要查找结点中的人名：王小欢
插入信息：请输入学生的姓名、成绩：唐雪梅89
插入结点后
王小欢的成绩为：78
唐雪梅的成绩为：89
张晓笑的成绩为：88
王大鹏的成绩为：99
删除结点后
唐雪梅的成绩为：89
张晓笑的成绩为：88
王大鹏的成绩为：99
```

上面的代码成功地实现了单向链表中结点的插入和删除操作，先来

看结点的查找和插入。先通过结点信息中的人名来查找结点，查找成功就返回结点地址，查找失败就退出。如果查找成功，那么接下来就调用node_insert（）函数将创建的新结点插入到查找到的结点后，插入方法参见图6-8。插入结束之后，就可以根据要求删除查找到的结点。需要注意的是，在删除结点时需要判断删除的结点是否为头结点所指向的结点，如果是，那么要移动头结点，否则就按如图6-9所示的方法进行删除。

对于单向循环链表，它的特殊之处就在于链表尾结点的指针域指向链表的头一个结点。单向循环链表的一般操作方法与单向链表类似，在此就不一一讲解，接下来介绍双向链表。

双向链表比单向链表多了一个指向前结点的前指针域，其实现步骤与前面讲解的单向链表完全相同。下面的代码实现的是双向链表的插入。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#include＜string.h＞
#define N 3
struct Stu
{
struct Stu*next，*prev;
char name[10];
int score;
};
struct Stu_head
{
struct Stu*head;
};
struct Stu_head*head_create（）
```

```
    {
    struct Stu_head*single_link;
    if ( (single_link= (struct Stu_head*) malloc (sizeof (struct
Stu_head) ) ) ==NULL)
    {
    printf ("分配空间失败! ") ;
    exit (0) ;
    }
    if (single_link! =NULL)
    {
    single_link->head=NULL;
    }
    return single_link;
    }
    struct Stu*node_create ()
    {
    struct Stu*node;
    if ( (node= (struct Stu*) malloc (sizeof (struct Stu) ) ) ==NULL)
    {
    printf ("分配空间失败! ") ;
    exit (0) ;
    }
    if (node! =NULL)
    {
    node->next=NULL;
    node->prev=NULL;
    printf ("请输入学生的姓名、成绩: ") ;
    scanf ("%s%d", &node->name, &node->score) ;
    }
    return node;
    }
    void node_append (struct Stu_head*single_link)
    {
    struct Stu*node=NULL;
    struct Stu*cursor=NULL;
    node=node_create () ;
    if (single_link->head==NULL)
    {
    single_link->head=node;
    }
    else
    {
    cursor=single_link->head;
    while (cursor! =NULL&&cursor->next! =NULL)
    {
    cursor=cursor->next;
    }
    cursor->next=node;
```

```
node->prev=cursor;
}
return;
}
void node_print (struct Stu_head*single_link)
{
struct Stu*iter=single_link->head;
while (iter! =NULL)
{
printf ("%s的成绩为: %d\n", &iter->name,iter->score) ;
iter=iter->next;
}
return;
}
void node_destroy (struct Stu*node)
{
if (node! =NULL)
{
node->next=NULL;
free (node) ;
}
return;
}
void single_link_destroy (struct Stu_head*single_link)
{
struct Stu*iter=single_link->head;
struct Stu*next=NULL;
while (iter! =NULL)
{
next=iter->next;
node_destroy (iter) ;
iter=next;
}
single_link->head=NULL;
free (single_link) ;
return;
}
int main ()
{
int i;
struct Stu_head*single_head=head_create () ;
for (i=0; i<N; i++)
node_append (single_head) ;
node_print (single_head) ;
single_link_destroy (single_head) ;
return 0;
}
```

运行结果：

```
请输入学生的姓名、成绩：王美清32
请输入学生的姓名、成绩：张梅梅45
请输入学生的姓名、成绩：彭晓倩78
王美清的成绩为：32
张梅梅的成绩为：45
彭晓倩的成绩为：78
```

双向链表的实现与前面单向链表的实现几乎完全相同，不同之处在于双向链表的结点多了一个前指针域，所以在向链表中添加结点时多了一个对前指针域的操作，这个指针域指向的是前一个结点。在上面的代码中，node_append（）函数中的“node-＞prev=cursor；”语句使前指针域指向当前加入结点的前一个结点。

在上面的代码中采用的是在双向链表的末端插入结点，在末端插入结点的方法与在头结点处插入结点的方法类似，接下来通过图6-10来了解如何在双向链表的两个普通结点之间插入一个结点。

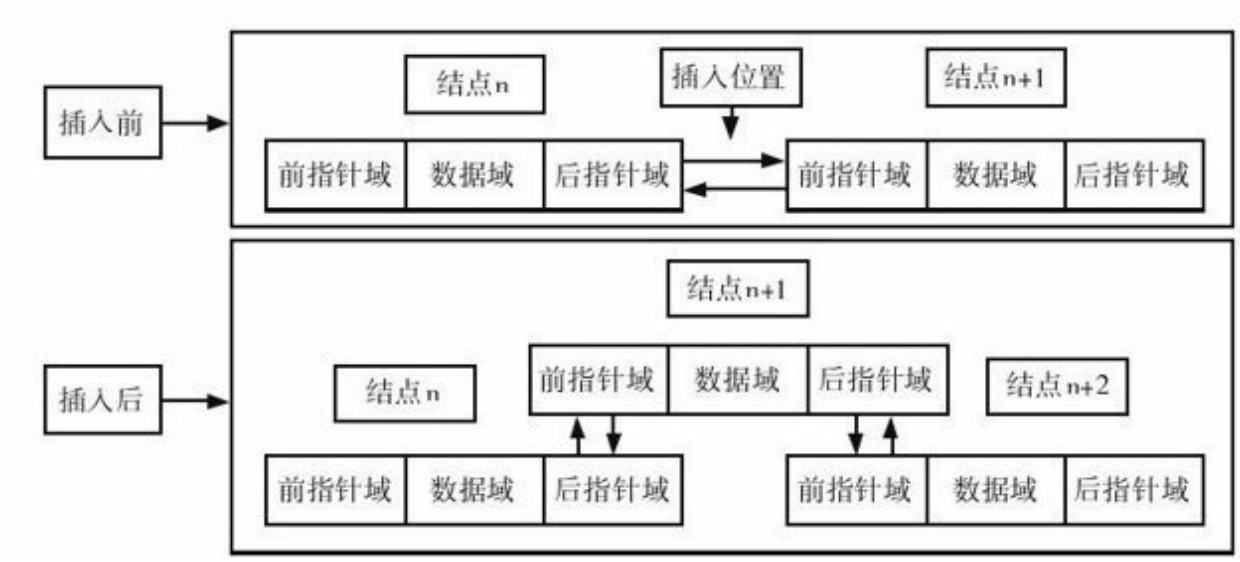

图　6-10　双向链表中结点的插入

在图6-10中，向双向链表中插入结点与前面介绍的向单向链表中插入结点的不同之处在于，多了一个前指针域的操作。接下来再通过图6-11来了解如何在双向链表中删除一个结点。

在图6-11中，注意删除结点前后的变化，在删除结点前，将其前指针域和后指针域赋值为NULL，对于动态分配的结点，需要将其申请的内存空间释放掉；删除结点后，原来的结点n+2就变为了当前链表的结点n+1。

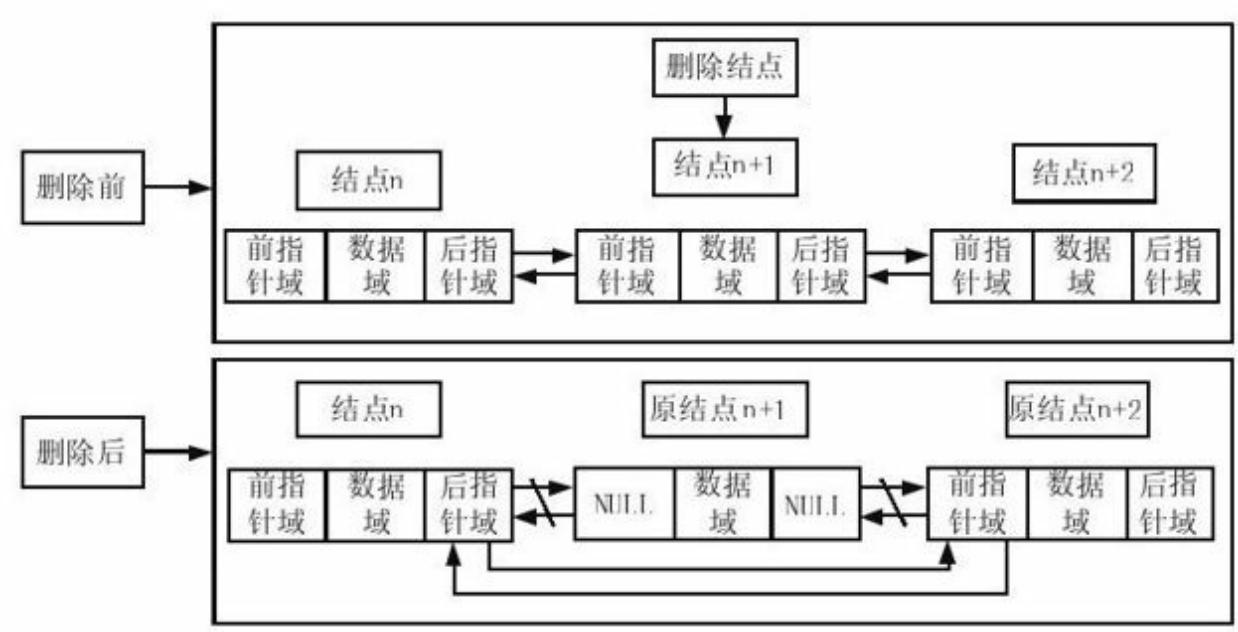

图　6-11　双向链表中结点的删除

对于双向链表中的任何一个结点，可以通过前后指针域对整个链表进行遍历，如：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define N 5
struct Stu
{
struct Stu*next, *prev;
char name[10];
int score;
};
struct Stu_head
{
struct Stu*head;
};
struct Stu_head*head_create ()
{
struct Stu_head*single_link;
if ( (single_link= (struct Stu_head*) malloc (sizeof (struct
Stu_head) ) ) ==NULL)
{
printf ("分配空间失败! ") ;
exit (0) ;
}
if (single_link! =NULL)
{
single_link->head=NULL;
}
return single_link;
}
struct Stu*node_create ()
{
struct Stu*node;
if ( (node= (struct Stu*) malloc (sizeof (struct Stu) ) ) ==NULL)
{
printf ("分配空间失败! ") ;
exit (0) ;
}
if (node! =NULL)
{
node->next=NULL;
node->prev=NULL;
printf ("请输入学生的姓名、成绩: ") ;
scanf ("%s%d", &node->name, &node->score) ;
}
return node;
}
void node_append (struct Stu_head*single_link)
```

```
{
struct Stu*node=NULL;
struct Stu*cursor=NULL;
node=node_create ();
if (single_link->head==NULL)
{
single_link->head=node;
}
else
{
cursor=single_link->head;
while (cursor! =NULL&&cursor->next! =NULL)
{
cursor=cursor->next;
}
cursor->next=node;
node->prev=cursor;
}
return;
}
struct Stu*node_search (struct Stu_head*single_link,char*name)
{
struct Stu*pos=NULL;
pos=single_link->head;
while (pos! =NULL)
{
if (strcmp (pos->name,name) ==0)
return pos;
else
pos=pos->next;
}
if (pos==NULL)
{
printf ("没有查找到该数据！");
exit (0);
}
return NULL;
}
void node_print (struct Stu*cursor,int mark)
{
struct Stu*iter=cursor;
while (iter! =NULL)
{
if (0==mark)
iter=iter->next;
else
iter=iter->prev;
if (NULL! =iter)
```

```
printf（"%s的成绩为：%d\n"，＆iter->name,iter->score）；
}
return;
}
void node_destroy（struct Stu*node）
{
if（node! =NULL）
{
node->next=NULL;
free（node）；
}
return;
}
void single_link_destroy（struct Stu_head*single_link）
{
struct Stu*iter=single_link->head;
struct Stu*next=NULL;
while（iter! =NULL）
{
next=iter->next;
node_destroy（iter）；
iter=next;
}
single_link->head=NULL;
free（single_link）；
return;
}
int main（）
{
int i;
char name[10];
struct Stu*search_node;
struct Stu_head*single_head=head_create（）；
for（i=0; i<N; i++）
node_append（single_head）；
printf（"请输入所有查找结点的人名："）；
scanf（"%s"，name）；
search_node=node_search（single_head,name）；
printf（"\n查找结点后面的结点信息\n"）；
node_print（search_node，0）；
printf（"\n查找结点前面的结点信息\n"）；
node_print（search_node，1）；
single_link_destroy（single_head）；
return 0;
}
```

运行结果：

```
请输入学生的姓名、成绩：张晓敏34
请输入学生的姓名、成绩：萧大鹏67
请输入学生的姓名、成绩：范德萨89
请输入学生的姓名、成绩：田晓梅99
请输入学生的姓名、成绩：庞大天23
请输入所有查找结点的人名：范德萨
查找结点后面的结点信息
田晓梅的成绩为：99
庞大天的成绩为：23
查找结点前面的结点信息
萧大鹏的成绩为：67
张晓敏的成绩为：34
```

上面的运行结果表明成功地实现了通过双向链表中的任何一个结点遍历整个链表。

对双向循环链表的操作，与前面所讲的双向链表的操作类似，最大的区别在于双向循环链表是首尾相连的一个“闭环”，如图6-12所示，所以从其中任何一个结点开始可以单方向遍历整个链表。在构成双向循环链表的过程中需要注意的一个重点就是首尾相连，即尾结点的后指针域指向首结点，而首结点的前指针域指向末结点。

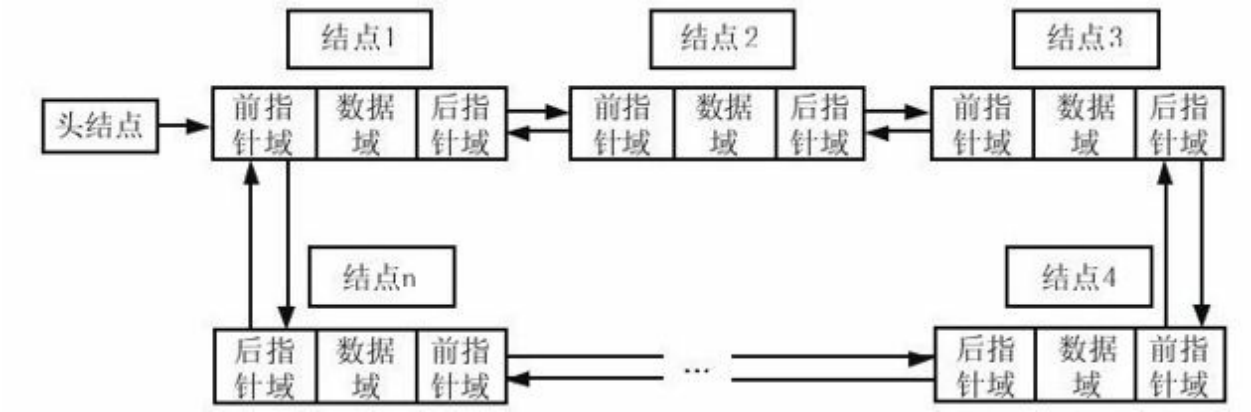

图　6-12　双向循环链表

6.6 颠覆传统链表的实现方法

由于Linux内核中有大量的数据结构都需要用到双向循环链表，若还采用以往那种传统双向循环链表的实现方式，就不得不为这些数据结构维护各自的链表，并且必须为每个链表设计插入、查找、删除等操作函数，这是因为常规链表中的前向指针prev和后向指针next都是指向对应类型的对象，一种数据结构的链表操作函数不能操作其他数据结构的链表。

为了解决传统链表中存在的上述问题，在Linux内核中将结构体中的前向指针prev和后向指针next从具体的数据结构中提取出来，构成一个通用的双向循环链表数据结构list_head。如果需要构造某类对象的特定链表，那么只需要在其结构体中定义一个类型为list_head类型的成员，通过这个list_head类型的成员将这类对象连接起来，形成所需的双向循环链表，进而通过通用链表函数对其进行操作。

显而易见，这种颠覆传统双向循环链表的实现方法，使我们无需为每个创建的双向循环链表编写专用函数，从而大大提高了代码的重用性。

由于VC编译器对部分操作符并不支持，所以本节的代码均在Linux环境下采用gcc编译运行。读者在开始学习本节之前，可以从网上先下

载一个Linux内核双向循环链表的头文件list.h，同时鉴于内核版本的不同，下载的头文件可能有些细小的差别，但是这并不影响我们对双向循环链表的学习。对双向循环链表的讲解大致按照其头文件中代码的先后顺序进行。

6.6.1 头结点的创建

首先来看list_head结构的实现，代码如下：

```
struct list_head{
struct list_head*next, *prev;
};
```

在Linux内核双向循环链表中，用list_head类型定义一个变量，将该变量作为一个成员嵌入到宿主结构中。所谓的宿主结构体，就是所创建的双向循环链表中每个结点的结构体类型。可以将链表结构放在宿主结构内的任何地方，当然也可以为链表结构取任何可用的变量名。注意，list_head结构体类型中同样有在建立双向循环链表时所用到的前指针域和后指针域。我们可以用list_head中的成员和相对应的处理函数来对链表进行遍历操作，要想得到宿主结构的指针，可以使用list_entry计算出来，先别急着知道list_entry是什么，我们会在下面讲解，接着往下看。

在宿主结构体中定义了list_head之后，接下来当然要对所有创建的双向循环链表头结点进行初始化工作。

方法一：

```
#define LIST_HEAD_INIT(name){&(name),&(name)}
#define LIST_HEAD(name)\
struct list_head name=LIST_HEAD_INIT(name)
```

方法二：

```
#define INIT_LIST_HEAD(ptr)do{\
(ptr)->next=(ptr);(ptr)->prev=(ptr);\
}while(0)
```

对头结点进行初始化有两种方法：一种是使用INIT_LIST_HEAD（ptr）宏进行初始化，使用这种方法时需要先定义一个头结点；另外一种是使用LIST_HEAD（name）宏进行初始化，在使用该方法进行初始化的时候，无需定义头结点，只需给出宏中的参数，即给出需要进行初始化的头结点名。

6.6.2 结点的添加

完成链表头的创建之后，接下来当然是向创建的链表头添加结点，可以通过__list_add（）函数来实现，其实现代码如下：

```
    static inline void__list_add(struct list_head*new,
    struct list_head*prev,
    struct list_head*next)
    {
    next->prev=new;
    new->next=next;
    new->prev=prev;
    prev->next=new;
    }
```

__list_add（）函数的功能是在两个非空结点中插入一个结点。值得注意的是，new、prev、next均不能为空值。当然，prev可以等于next，这表示在只含头结点的链表中插入新结点。知道了__list_add（）函数的实现后，接下来再看函数list_add（）和list_add_tail（）的实现，代码如下：

```
    static inline void list_add(struct list_head*new,struct
list_head*head)
    {
    __list_add(new,head,head->next);
    }
    static inline void list_add_tail(struct list_head*new,struct
list_head*head)
    {
    __list_add(new,head->prev,head);
    }
```

由上面的实现方式可知，list_add（）和list_add_tail（）都是调用底层的__list_add（）来实现的。通过在__list_add（）函数中传递不同的参数就能实现不同的添加结点的方法。__list_add（）函数前面的“__”通常表示这是一个底层函数，供其他的模块调用。

list_add（）函数传递的参数实现的是在head和head-＞next两个指针所指向的结点之间插入new所指向的结点，即在head指针的后面插入一个new所指向的结点。head并不一定为头结点。如果链表只含有一个头结点，那么上面的__list_add（new,head,head-＞next）；仍然成立。

list_add_tail（）函数的功能是在结点指针head所指向结点的前面插入new所指向的结点。如果head指向的是头结点，那么就相当于在尾结点后增加一个new所指向的结点。注意，这个函数中的head-＞prev不能为空，如果head为头结点，那么head-＞prev要指向一个数值，一般指向尾结点，从而构成循环链表。

为了帮助读者掌握list.h头文件中以上函数的使用，接下来就利用Linux内核中的list.h在应用层中创建链表。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#include"list.h"
typedef struct_stu
{
char name[20];
int num;
struct list_head list;
}stu;
```

```
    int main ()
    {
    stu*pstu;
    stu*tmp_stu;
    struct list_head stu_list; struct list_head*pos;
    int i=0;
    INIT_LIST_HEAD (&stu_list) ;
    pstu=malloc (sizeof (stu) *5) ;
    for (i=0; i<5; i++)
    {
    sprintf (pstu[i].name, "Stu%d", i+1) ;
    pstu[i].num=i+1;
    list_add (& (pstu[i].list) , &stu_list) ;
    }
    list_for_each (pos, &stu_list)
    {
    tmp_stu=list_entry (pos,stu,list) ;
    printf ("student num: %d\tstudent name: %s\n", tmp_stu->
num,tmp_stu->name) ;
    }
    free (pstu) ;
    return 0;
    }
```

运行结果：

```
    root@ubuntu:/home/paixu/dlist_node#./a
    student num: 5 student name: Stu5
    student num: 4 student name: Stu4
    student num: 3 student name: Stu3
    student num: 2 student name: Stu2
    student num: 1 student name: Stu1
```

上面的代码主要完成了以下几项基本工作：

定义了一个宿主结构体stu，并且在宿主结构体中定义了一个struct list_head类型的list变量。

定义一个头结点并对其进行初始化。

对定义的一个宿主结构体变量申请内存空间。

对申请的宿主结构体变量初始化和并将其添加到以stu_list为头结点的链表中。

在上面的代码中，值得注意的是函数list_for_each（）和list_entry（），这两个函数会在后面进行讲解，在这里，读者只需知道它们两个合在一起的作用是打印出宿主结构stu中的每个数据。list_add_tail（）函数的使用和list_add（）函数类似，读者可以自己修改代码实现。

6.6.3 结点的删除

接下来看删除结点的函数__list_del（），其实现代码如下：

```
    static inline void__list_del（struct list_head*prev,struct
list_head*next)
    {
    next->prev=prev;
    prev->next=next;
    }
```

对于prev和next指针所指向的结点，两者互相所指，也就是说，prev为待删除结点的前一个结点，next为待删除结点的后一个结点。在上面的代码中，仅仅将所要删除的结点从链表中“拿掉”了，并没有对其前指针域和后指针域进行相应的处理，这对于结点来说无疑是危险的，所以在list.h中通常会采用以下这个删除结点的函数。

```
    static inline void list_del（struct list_head*entry)
    {
    __list_del（entry->prev,entry->next）;
    entry->next=LIST_POISON1;
    entry->prev=LIST_POISON2;
    }
```

上面的代码实现了删除entry所指的结点，同时将entry所指向的结点指针域“封死”。而实现“封死”操作的两个值分别是LIST_POISON1、LIST_POISON2。它们的值到底是什么呢？它们在list.h中是如下定义的：

```
#define LIST_POISON1 ( (void*) 0x00100100)
#define LIST_POISON2 ( (void*) 0x00200200)
```

对LIST_POISON1、LIST_POISON2的说明，在Linux内核中有这么一句话：“These are non-NULL pointers that will result in page faults under normal circumstances,used to verify that nobody uses non-initialized list entries.”，也就是说，这两者并不是空指针，但是访问这样的指针在正常情况下会出错。按照一般的思路，把entry-＞next和entry-＞prev赋值为NULL，不可以通过该结点对链表再次进行访问，但是在这里使用了一种特殊的方法。注意，在Linux环境下，以上宏LIST_POISON1和LIST_POISON2的值不进行修改是不会出错的，但是在VC下就会出错，要将（（void*）0x00100100）和（（void*）0x00200200）均修改为NULL。

如果要将删除的结点作为链表头另外创建一个链表，那么可以采用下面的函数来实现，代码如下：

```
static inline void list_del_init (struct list_head*entry)
{
__list_del (entry->prev,entry->next) ;
INIT_LIST_HEAD (entry) ;
}
```

以上函数的功能为删除entry所指向的结点，同时调用INIT_LIST_HEAD（）把被删除结点作为链表头构建一个新的空双循环链表。

通过讲解了如何通过哪些函数来实现结点的删除操作，接下来通过代码来看看具体的实现方法。

```
    #include＜stdio.h＞
    #include＜stdlib.h＞
    #include"list.h"
    typedef struct_stu
    {
    char name[20];
    int num;
    struct list_head list;
    }stu;
    int main ()
    {
    stu*pstu;
    stu*tmp_stu;
    struct list_head stu_list;
    struct list_head*pos;
    int i=0;
    INIT_LIST_HEAD（&stu_list）;
    pstu=malloc（sizeof（stu）*5）;
    for（i=0; i＜5; i++）
    {
    sprintf（pstu[i].name, "Stu%d", i+1）;
    pstu[i].num=i+1;
    list_add（&（pstu[i].list）, &stu_list）;
    }
    list_del（&（pstu[3].list））;
    printf（"使用list_del（）删除pstu[3]\n"）;
    list_for_each（pos, &stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）;
    printf（"student num: %d\tstudent name: %s\n", tmp_stu-＞
num,tmp_stu-＞name）;
    }
    list_del_init（&（pstu[2].list））;
    printf（"使用list_del_init（）删除pstu[2]\n"）;
    list_for_each（pos, &stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）;
    printf（"student num: %d\tstudent name: %s\n", tmp_stu-＞
num,tmp_stu-＞name）;
    }
    free（pstu）;
```

```
return 0;
}
```

运行结果：

```
root@ubuntu:/home/paixu/dlist_node#./a
使用list_del（）删除pstu[3]
student num: 5 student name: Stu5
student num: 3 student name: Stu3
student num: 2 student name: Stu2
student num: 1 student name: Stu1
使用list_del_init（）删除pstu[2]
student num: 5 student name: Stu5
student num: 2 student name: Stu2
student num: 1 student name: Stu1
```

在以上的代码中创建了5个结点，然后调用list_del（）函数删除了pstu[3]，调用list_del_init（）函数删除了pstu[2]，从打印结果可以看出成功地调用了这两个删除函数实现了链表中结点的删除操作。

6.6.4 结点位置的调整

我们经常会根据需要对链表中结点的位置进行相应的调整，那么在list.h中又是如何来实现的呢？下面的代码是实现结点位置调整的函数list_move（）和list_move_tail（）。

```
    static inline void list_move (struct list_head*list,struct
list_head*head)
    {
    __list_del (list->prev,list->next) ;
    list_add (list,head) ;
    }
    static inline void list_move_tail (struct list_head*list,
    struct list_head*head)
    {
    __list_del (list->prev,list->next) ;
    list_add_tail (list,head) ;
    }
```

在上面的代码中，list_move（）函数的功能是把list移至head和head-＞next两个指针所指向的结点之间，而list_move_tail（）函数的功能是把list移至head和head-＞prev两个指针所指向的结点之间。接下来还是通过一段代码来具体看看如何使用以上两个函数来实现结点位置的调整。

```
    #include＜stdio.h＞
    #include＜stdlib.h＞
    #include"list.h"
    typedef struct_stu
    {
    char name[20];
    int num;
```

```
    struct list_head list;
    }stu;
    int main ()
    {
    stu*pstu;
    stu*tmp_stu;
    struct list_head stu_list;
    struct list_head*pos;
    int i=0;
    INIT_LIST_HEAD (&stu_list) ;
    pstu=malloc (sizeof (stu) *5) ;
    for (i=0; i<5; i++)
    {
    sprintf (pstu[i].name, "Stu%d", i+1) ;
    pstu[i].num=i+1;
    list_add (& (pstu[i].list) , &stu_list) ;
    }
    list_move (& (pstu[3].list) , &stu_list) ;
    printf ("把pstu[3]移至head和head->next两个指针所指向的结点之间\n") ;
    list_for_each (pos, &stu_list)
    {
    tmp_stu=list_entry (pos,stu,list) ;
    printf ("student num: %d\tstudent name: %s\n", tmp_stu->
num,tmp_stu->name) ;
    }
    list_move_tail (& (pstu[2].list) , &stu_list) ;
    printf ("把pstu[2]移至head和head->prev两个指针所指向的结点之间\n") ;
    list_for_each (pos, &stu_list)
    {
    tmp_stu=list_entry (pos,stu,list) ;
    printf ("student num: %d\tstudent name: %s\n", tmp_stu->
num,tmp_stu->name) ;
    }
    free (pstu) ;
    return 0;
    }
```

运行结果：

```
    root@ubuntu:/home/paixu/dlist_node#./a
    把pstu[3]移至head和head->next两个指针所指向的结点之间
    student num: 4 student name: Stu4
    student num: 5 student name: Stu5
    student num: 3 student name: Stu3
    student num: 2 student name: Stu2
```

```
student num: 1 student name: Stu1
把pstu[2]移至head和head->prev两个指针所指向的结点之间
student num: 4 student name: Stu4
student num: 5 student name: Stu5
student num: 2 student name: Stu2
student num: 1 student name: Stu1
student num: 3 student name: Stu3
```

这里需要提醒读者注意的是，pstu[]的下标是从0开始的，所以pstu[3]对应的是stu4。从运行结果可以看出，上面的代码成功地实现了对结点在链表中位置的调整。

6.6.5 检测链表是否为空

在实际编程中，我们经常要判断一个链表是否为空，list.h中有相应的函数list_empty（）和list_empty_careful（），通过调用这两个函数就可以实现相应的判断操作，这两个函数的实现如下：

```
    static inline int list_empty（const struct list_head*head）
    {
    return head->next==head;
    }
    static inline int list_empty_careful（const struct
list_head*head）
    {
    struct list_head*next=head->next;
    return（next==head）&&（next==head->prev）;
    }
```

list_empty（）函数和list_empty_careful（）函数都是用来检测链表是否为空的，它们之间仅有的区别是第一个函数使用的检测方法是判断表头结点的下一个结点是否是其本身，如果是，则返回true，否则返回false；第二个函数使用的检测方法是判断表头的前一个结点和后一个结点是否为其本身，如果同时满足，则返回false，否则返回值为true。接下来通过一段具体的代码来看如何使用以上函数来判断链表是否为空。

```
    #include<stdio.h>
    #include<stdlib.h>
    #include"list.h"
    typedef struct_stu
    {
    char name[20];
    int num;
```

```
    struct list_head list;
    }stu;
    int main ()
    {
    stu*pstu;
    stu*tmp_stu;
    struct list_head stu_list;
    struct list_head*pos;
    int i=0;
    INIT_LIST_HEAD (&stu_list) ;
    pstu=malloc (sizeof (stu) *5) ;
    for (i=0; i<5; i++)
    {
    sprintf (pstu[i].name, "Stu%d", i+1) ;
    pstu[i].num=i+1;
    list_add (& (pstu[i].list) , &stu_list) ;
    }
    list_for_each (pos, &stu_list)
    {
    tmp_stu=list_entry (pos,stu,list) ;
    printf ("student num: %d\tstudent name: %s\n", tmp_stu->
num,tmp_stu->name) ;
    }
    if (list_empty (&stu_list) )
    printf ("使用list_empty () 检测，链表为空\n") ;
    else
    printf ("使用list_empty () 检测，链表非空\n") ;
    if (list_empty_careful (&stu_list) )
    printf ("使用list_empty_careful () 检测，链表为空\n") ;
    else
    printf ("使用list_empty_careful () 检测，链表非空\n") ;
    free (pstu) ;
    return 0;
    }
```

运行结果：

```
    root@ubuntu:/home/paixu/dlist_node#./a
    student num: 5 student name: Stu5
    student num: 4 student name: Stu4
    student num: 3 student name: Stu3
    student num: 2 student name: Stu2
    student num: 1 student name: Stu1
    使用list_empty () 检测，链表非空
    使用list_empty_careful () 检测，链表非空
```

上面的代码通过list_empty（）函数和list_empty_careful（）函数实现了对链表是否为空的检测。

6.6.6 链表的合成

如果需要对链表进行合成操作，那么在list.h中又是如何实现的呢？接下来看看用于链表合成操作的函数__list_splice（）的代码。

```
static inline void__list_splice(struct list_head*list,
struct list_head*head)
{
struct list_head*first=list->next;
struct list_head*last=list->prev;
struct list_head*at=head->next;
first->prev=head;
head->next=first;
last->next=at;
at->prev=last;
}
```

该函数的功能是将一个链表插入到另外一个链表中，不做链表是否为空的检查。调用该函数进行合成操作的时候，对于链表是否为空的检测需要在调用该函数之前进行。因为每个链表只有一个头结点，将空链表插入到另外一个链表中是没有意义的，而被插入的链表可以为空。当然，也可以采用下面的函数先对链表进行是否为空的检测，再对其进行合成操作。

```
static inline void list_splice(struct list_head*list,struct
list_head*head)
{
if(!list_empty(list))
__list_splice(list,head);
}
```

在以上链表的合成操作中会丢弃list所指向的头结点，因为两个链表各有一个头结点，所以必须删除其中一个头结点。只要list是非空链，head无任何限制，list_splice（）就能实现链表的合并。如果需要使用丢弃的链表头来创建另外一个链表，那么可以使用下面的函数来实现。

```
static inline void list_splice_init(struct list_head*list,
struct list_head*head)
{
if(!list_empty(list)){
__list_splice(list,head);
INIT_LIST_HEAD(list);
}
}
```

这个函数的功能是将一个链表list的有效信息合并到另外一个链表head之后，重新初始化被丢弃的空链表头。对于以上实现链表合成操作函数，同样通过下面的一段代码来看具体的使用方法。

```
#include<stdio.h>
#include<stdlib.h>
#include"list.h"
typedef struct_stu
{
char name[20];
int num;
struct list_head list;
}stu;
int main()
{
stu*pstu,*pstu2;
stu*tmp_stu;
struct list_head stu_list,stu_list2;
struct list_head*pos;
int i=0;
```

```
    INIT_LIST_HEAD（&stu_list）；
    INIT_LIST_HEAD（&stu_list2）；
    pstu=malloc（sizeof（stu）*3）；
    pstu2=malloc（sizeof（stu）*3）；
    for（i=0；i＜3；i++）
    {
    sprintf（pstu[i].name，"Stu%d"，i+1）；
    sprintf（pstu2[i].name，"Stu%d"，i+4）；
    pstu[i].num=i+1;
    pstu2[i].num=i+4;
    list_add（&（pstu[i].list），&stu_list）；
    list_add（&（pstu2[i].list），&stu_list2）；
    }
    printf（"stu_list链表\n"）；
    list_for_each（pos，&stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）；
    printf（"student num：%d\tstudent name：%s\n"，tmp_stu->
num,tmp_stu->name）；
    }
    printf（"stu_list2链表\n"）；
    list_for_each（pos，&stu_list2）
    {
    tmp_stu=list_entry（pos,stu,list）；
    printf（"student num：%d\tstudent name：%s\n"，tmp_stu->
num,tmp_stu->name）；
    }
    printf（"stu_list链表和stu_list2链表合并以后\n"）；
    list_splice（&stu_list2，&stu_list）；
    list_for_each（pos，&stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）；
    printf（"student num：%d\tstudent name：%s\n"，tmp_stu->
num,tmp_stu->name）；
    }
    free（pstu）；
    return 0;
    }
```

运行结果：

```
    root@ubuntu:/home/paixu/dlist_node#./a
    stu_list链表
    student num：3 student name：Stu3
    student num：2 student name：Stu2
```

```
student num: 1 student name: Stu1
stu_list2链表
student num: 6 student name: Stu6
student num: 5 student name: Stu5
student num: 4 student name: Stu4
stu_list链表和stu_list2链表合并以后
student num: 6 student name: Stu6
student num: 5 student name: Stu5
student num: 4 student name: Stu4
student num: 3 student name: Stu3
student num: 2 student name: Stu2
student num: 1 student name: Stu1
```

6.6.7 宿主结构指针

本书到目前为止一直没有讲读者可能最关心的宿主结构。如何取出宿主结构的指针，这是笔者认为Linux内核双向循环链表实现最为巧妙的地方，接下来就来看如何取出宿主结构的指针。

```
    #define list_entry（ptr,type,member）\
    （（type*）（（char*）（ptr）-（unsigned long）（&（（type*）0）->
member））
```

在上面的代码中可以发现一个很熟悉的身影“（unsigned long）（&（（type*）0）->member））”，这个在前面已经讲解过了，在此就不再过多讲解，有不明白的读者可以回过头去阅读。通过“（unsigned long）（&（（type*）0）->member））”可以得出成员变量member的偏移量，而ptr为指向member的指针，因为指针类型不同，所以要先进行（char*）转换后再进行计算。用ptr减去member的偏移量就得到了宿主结构体的指针，这就是一个非常巧妙的地方，也是使Linux内核双向循环链表能够区别于传统链表的关键所在。看到这儿，读者可能已经感觉非常枯燥了，但是别放弃，坚持看完，因为后面介绍的内容是非常有用的，再坚持一下吧！

6.6.8 链表的遍历

接下来看实现遍历双向循环链表基本操作的宏。

```
#define list_for_each (pos,head) \
for (pos= (head) ->next; prefetch (pos->next) , pos! = (head) ; \
pos=pos->next)
#define__list_for_each (pos,head) \
for (pos= (head) ->next; pos! = (head) ; pos=pos->next)
#define list_for_each_prev (pos,head) \
for (pos= (head) ->prev; prefetch (pos->prev) , pos! = (head) ; \
pos=pos->prev)
```

遍历是双向循环链表的基本操作，head为头结点，遍历过程先从（head）->next开始，当pos==head时退出，故head结点并没有被访问。这和链表的结构设计有关，通常，头结点都不含有其他有效信息，因此可以把头结点作为双向链表遍历一遍的检测标志来使用。在list_for_each（）宏中，读者可能会发现一个比较陌生的面孔——prefetch，有兴趣的读者可以自己查看下它的实现。其功能是预取内存的内容，也就是告诉CPU哪些内容可能马上用到，CPU预先取出内存操作数，然后将其送入高速缓存，用于优化，使执行速度更快。__list_for_each（）宏和list_for_each_prev（）宏与list_for_each（）宏类似，区别分别在于遍历的方向有所改变和不使用预取。

通过上面的代码和讲解，相信读者对于list_for_each（）宏已经不再感到陌生了。上面三个遍历链表的宏都类似，接下来看另外一种遍历

宏的实现方法。

```
#define list_for_each_safe（pos,n，head）\
for（pos=（head）->next,n=pos->next；pos!=（head）；\
pos=n,n=pos->next）
```

这个list_for_each_safe（）宏同样是用于遍历的，不同的是，其中多了一个指针n用于暂存pos的下一个结点的地址，避免了因为pos结点被释放而造成的断链，这也就体现了该宏的安全机制。也就是说，可以遍历完当前结点后将其删除，同时接着访问下一个结点，遍历完毕后，就只剩下一个头结点。一个最典型的应用是，在多进程编程时候，多个进程在同一个等待队列中，若事件发生时唤醒所有进程，则可以在唤醒后将其依次从等待队列中删除。下面通过一段代码来看这种安全机制的使用。

```
#include＜stdio.h＞
#include＜stdlib.h＞
#include"list.h"
typedef struct_stu
{
char name[20];
int num;
struct list_head list;
}stu;
int main（）
{
stu*pstu;
stu*tmp_stu;
struct list_head stu_list;
struct list_head*pos，*n;
int i=0;
INIT_LIST_HEAD（＆stu_list）；
pstu=malloc（sizeof（stu）*3）；
for（i=0；i＜3；i++）
```

```
    {
    sprintf（pstu[i].name，"Stu%d"，i+1）；
    pstu[i].num=i+1;
    list_add（&（pstu[i].list），&stu_list）；
    }
    printf（"通过list_for_each_safe（）遍历使用list_del（pos）删除结点前
\n"）；
    list_for_each_safe（pos,n，&stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）；
    printf（"student num：%d\tstudent name：%s\n"，tmp_stu->
num,tmp_stu->name）；
    list_del（pos）；
    }
    printf（"通过list_for_each（）遍历使用list_del（pos）删除结点后\n"）；
    list_for_each（pos，&stu_list）
    {
    tmp_stu=list_entry（pos,stu,list）；
    printf（"student num：%d\tstudent name：%s\n"，tmp_stu->
num,tmp_stu->name）；
    }
    free（pstu）；
    return 0;
    }
```

运行结果：

```
    root@ubuntu:/home/paixu/dlist_node#./a
    通过list_for_each_safe（）遍历使用list_del（pos）删除结点前
    student num：3 student name：Stu3
    student num：2 student name：Stu2
    student num：1 student name：Stu1
    通过list_for_each（）遍历使用list_del（pos）删除结点后
```

只提供对list_head结构的遍历操作是远远不够的，我们希望实现对宿主结构的遍历，即在遍历时直接获得当前链表结点所在的宿主结构项，而不是每次都要同时调用list_for_each（）和list_entry（），为此，Linux专门提供了list_for_each_entry（）宏，该宏的代码如下：

```
#define list_for_each_entry(pos,head,member) \
for(pos=list_entry((head)->next,typeof(*pos),member); \
prefetch(pos->member.next), &pos->member!=(head); \
pos=list_entry(pos->member.next,typeof(*pos),member))
```

第一个参数为传入的遍历指针，指向宿主数据结构；第二个参数为链表头，为list_head结构；第三个参数为list_head结构在宿主结构中的成员名。接下来通过下面的一段代码来看该宏的具体使用。

```
#include<stdio.h>
#include<stdlib.h>
#include"list.h"
typedef struct_stu
{
char name[20];
int num;
struct list_head list;
}stu;
int main()
{
stu*pstu;
stu*tmp_stu;
struct list_head stu_list;
struct list_head*pos, *n;
int i=0;
INIT_LIST_HEAD(&stu_list);
pstu=malloc(sizeof(stu)*3);
for(i=0; i<3; i++)
{
sprintf(pstu[i].name, "Stu%d", i+1);
pstu[i].num=i+1;
list_add(&(pstu[i].list), &stu_list);
}
list_for_each_entry(tmp_stu, &stu_list,list)
printf("student num: %d\tstudent name: %s\n", tmp_stu->
num,tmp_stu->name);
free(pstu);
return 0;
}
```

运行结果：

```
root@ubuntu:/home/paixu/dlist_node#./a
student num: 3 student name: Stu3
student num: 2 student name: Stu2
student num: 1 student name: Stu1
```

接下来看list.h中与双向循环链表操作有关的最后几个宏的实现。

```
#define list_for_each_entry_reverse(pos,head,member) \
for(pos=list_entry((head)->prev,typeof(*pos),member); \
prefetch(pos->member.prev), &pos->member!=(head); \
pos=list_entry(pos->member.prev,typeof(*pos),member))
#define list_prepare_entry(pos,head,member) \
((pos)?: list_entry(head,typeof(*pos),member))
#define list_for_each_entry_continue(pos,head,member) \
for(pos=list_entry(pos->member.next,typeof(*pos),member); \
prefetch(pos->member.next), &pos->member!=(head); \
pos=list_entry(pos->member.next,typeof(*pos),member))
#define list_for_each_entry_safe(pos,n,head,member) \
for(pos=list_entry((head)->next,typeof(*pos),member), \
n=list_entry(pos->member.next,typeof(*pos),member); \
&pos->member!=(head); \
pos=n,n=list_entry(n->member.next,typeof(*n),member))
```

以上几个函数与list_for_each_entry（）类似，只是略有差别。list_prepare_entry（）中含有prefetch（），它的作用在上面已经讲解过，在此不再做过多讲解。list_for_each_entry_continue（）和list_for_each_entry（）的区别主要是list_for_each_entry_continue（）可以不从链表头开始遍历，而是从已知的某个pos结点的下一个结点开始遍历。在某些时候，如果不从头结点开始遍历，那么为了保证pos的初始值有效，必须使用list_prepare_entry（）。如果pos为非空，那么pos的

值就为其本身；如果pos为空，那么就从链表头强制扩展一个虚pos指针，读者可以自己分析list_prepare_entry（）的实现。

list_for_each_entry_safe（）要求调用者另外提供一个与pos同类型的指针n，用于在for循环中暂存pos下一个结点的宿主结构体的地址，以免因pos结点被释放而造成断链。

第7章　函数

每位C语言学习者都知道，C语言是通过函数来实现模块化程序设计的，所以较大的C语言程序通常由多个函数组成，而每个函数都对应一种特定的功能。正是这种模块化的程序设计使程序的结构更加清晰，更加易于后期的维护，因此掌握好函数的使用对学习C语言尤为重要。本章有针对性地对C语言初学者在函数使用中的一些易错点进行讲解，同时深入分析函数间的调用关系，使读者对于函数之间的调用关系更加清晰。

7.1 函数参数

在讲解函数参数之前，先来看函数定义的一般形式：

```
类型说明符 函数名（参数）
{
函数体;
}
```

函数由以上几个部分所构成，如果函数在定义时不带有任何的参数，即参数为空，那么函数在调用时同样不用带参数，看下面的代码。

```
#include<stdio.h>
void print ()
{
printf ("Hello Wrold! \n") ;
return;
}
int main ()
{
print () ;
return 0;
}
```

运行结果：

```
Hello Wrold!
```

上面定义的print（）函数没有任何参数，所以在调用的时候只需要使用函数名加括号的简单形式就可以实现函数调用。但是更多的时候使用的是带参数的函数，在使用带参数的函数时，需要注意的是传值和传

址的区别：传值是一种简单的对于实参的复制，形参有自己的存储空间，所以在函数中对于形参的操作不会影响实参；而传址传递的是地址，实参和形参之间共享同一存储空间，在函数中对于形参的操作同样影响到实参。因此在写函数的时候，要清楚地知道需要采用传值还是传址。接下来通过下面的一段代码来了解数组作为函数参数的使用。

```
#include＜stdio.h＞
void sort（int a[]，int n）
{
int i,j，k,temp;
for（i=0；i＜n-1；i++）
{
k=i;
for（j=i+1；j＜n；j++）
{
if（a[j]＜a[k]）
k=j;
}
if（k! =i）
{
temp=a[i];
a[i]=a[k];
a[k]=temp;
}
}
return;
}
int main（）
{
int i;
int arr[8]={99，65，45，15，89，57，28，23};
sort（arr，8）；
for（i=0；i＜8；i++）
printf（"%d\t"，arr[i]）；
printf（"\n"）；
return 0;
}
```

运行结果：

```
15 23 28 45 57 65 89 99
```

通过上面的代码可知，在sort（）函数中对形参数组a进行排序，但是在main（）函数中打印输出的arr数组是进行排序后的结果，这是什么原因呢？如图7-1所示为参数传递的示意图。

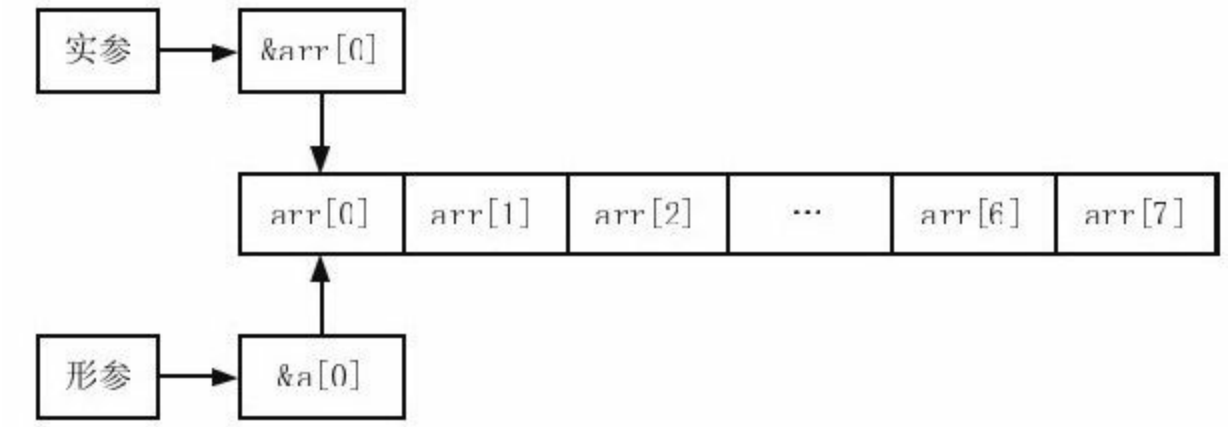

图　7-1　参数的传递

从图7-1中可以看出，形参和实参都指向同一片存储空间，所以会在函数中对形参进行排序，从而实现对实参的排序操作。但是需要注意的是，在采用一维数组作为参数的时候，在形参中并没有直接指定数组的大小，而是通过第二个参数来告知函数一维数组的长度。当然也可以直接指定数组的大小，如：

```
#include<stdio.h>
void print(int a[8])
{
int i;
```

```
for (i=0; i<8; i++)
printf ("%d\t", a[i]) ;
printf ("\n") ;
return;
}
int main ()
{
int arr[8]={99, 65, 45, 15, 89, 57, 28, 23};
print (arr) ;
return 0;
}
```

运行结果：

```
99 65 45 15 89 57 28 23
```

从上面的代码可知，在函数的形参中指定了数组的大小，这样在调用函数时就需要注意，如果调用函数的数组长度小于形参中指定的数组长度，那么编译就会出现“error C2078：too many initializers”错误，这是因为在函数中形参的数组长度大于调用时的实参数组长度，使得形参中多余的数组元素没有确定值而出错，如果调用的数组长度大于形参中给定的数组长度，那么调用不会出现任何问题。

在前面已经多次提及，数组作为函数参数时会退化为指针，已经不再是数组了，通过下面一段代码来加深一下印象。

```
#include<stdio.h>
void print (int a[8])
{
printf ("sizeof (a) =%d\n", sizeof (a) ) ;
return;
}
int main ()
```

```
{
int arr[8]={99，65，45，15，89，57，28，23};
printf（"sizeof（arr）=%d\n"，sizeof（arr））；
print（arr）；
return 0;
}
```

运行结果：

```
sizeof（arr）=32
sizeof（a）=4
```

由上面的运行结果可知，形参中的数组所占用的内存大小为4字节，而main（）函数中的实参数组占用的内存大小为32字节，刚好为数组长度和每个数组元素所占用内存的乘积。而形参中的数组a[]实际已经不是数组，而是一个int型指针a，这也就是为什么在以数组作为参数时是传址的原因了。因为数组名就是数组的首地址，所以函数调用过程中传递的也就是数组的首地址。

接下来看多维数组作为函数参数的使用情况。

```
#include＜stdio.h＞
void print（int row,int a[][3]）
{
int i,j;
for（i=0; i＜row; i++）
{
for（j=0; j＜3; j++）
printf（"%d\t"，a[i][j]）；
printf（"\n"）；
}
return;
}
int main（）
```

```
{
int arr[3][3]={99，65，45，15，89，57，28，23，45};
print（3，arr）;
return 0;
}
```

运行结果：

```
99 65 45
15 89 57
28 23 45
```

在使用二维数组作为函数参数的时候需要注意的是，必须执行第二维的大小，否则会出错，前面介绍过多维数组在内存中是按照一维数组的方式来存储的，所以在使用多维数组作为函数参数的时候必须执行每行有多少个元素，否则无法区分每行有多少个元素，进而导致编译出错。当然，可以同时指定数组中每一维的长度，所以在使用所谓数组作为函数参数的时候，要尤其注意其使用格式。接下来再看看三维数组作为函数参数的使用。

```
#include＜stdio.h＞
void print（int row,int a[][2][2]）
{
int i,j，k;
for（i=0; i＜row; i++）
{
for（j=0; j＜2; j++）
{
for（k=0; k＜2; k++）
printf（"%d\t"，a[i][j][k]）;
printf（"\n"）;
}
printf（"\n"）;
}
```

```
return;
}
int main ()
{
int arr[2][2][2]={99, 65, 45, 15, 89, 57, 28, 23};
print (2, arr) ;
return 0;
}
```

运行结果：

```
99 65
45 15
89 57
28 23
```

从上面的代码中可以看到，对形参中的三维数组同样指定了后两维的长度，所以对于多维数组作为函数参数的情况，我们必须清楚地知道，除了第一维可以省略不写以外，其余每一维都必须给出其数组长度。但是同样需要注意的是，多维数组作为函数参数时，它也是指针，而且这时函数调用的实参和形参除了第一维大小可以不同以外，其余各维的大小必须一致，否则编译出错，同时第一维中形参大于实参的部分，其结果都是随机的。

7.2 变参函数的实现方法

此前介绍的自定义函数都是一些确定参数的函数，但是我们却使用了变参函数。例如，常用的printf（）、scanf（）函数等就是典型的变参函数。printf（）函数和scanf（）函数的声明形式如下：

```
printf（const char*，……）；
scanf（const char*，……）；
```

读者可以发现上面两个函数的声明中有一个共同点，那就是都含有一个占位符“……”。占位符在这里并不是参数，只是告诉编译器，该函数是变参函数，不管该函数使用时的参数有多少，都对其一一做压栈处理，这就实现了变参函数。

说到压栈，读者应该并不陌生，在第1章讲解堆栈的概念时特地分析了函数的压栈操作，但是并没有提及函数参数的压栈是如何进行的，接下来就介绍函数参数的压栈。其实，函数参数的压栈和函数中数组的压栈操作类似。在第1章讲解函数中数组的压栈操作时讲到，数组的压栈是从最后一个元素开始的，从高地址到低地址，数组中的第一个元素最后被压栈，而函数参数的压栈操作也是从右向左进行的，从最后一个参数开始压栈，直到将第一个参数压栈为止。为了加深读者的印象，下面通过一段代码来看具体的压栈操作。

```
#include＜stdio.h＞
```

```
void print（int n，……）
{
int*p,i;
p=&n+1;
for（i=0; i＜n; i++）
printf（"%d\t", p[i]）;
printf（"\n"）;
return;
}
int main（）
{
print（4，12，34，56，78）;
return 0;
}
```

运行结果：

```
12 34 56 78
```

从上面的代码可以看到，首先在print（）函数中使用了占位符“……”，因此该函数在编译的时候被当成变参函数来处理，对该函数调用中的参数一一进行压栈处理。如图7-2所示为函数参数在栈中的存储结构。上述代码定义了一个int型指针变量p，由于函数参数的压栈顺序是从右向左，由高地址到低地址，所以在函数中通过“p=＆n+1；”得到的是第一个可变参数的地址，接下来通过一个for循环一一取出函数中的参数。在此使用第一个函数参数表示传递可变参数的个数。在使用变参函数的时候，必须知道参数什么时候结束。如果没有给出变参函数的个数，直接给出第一个参数，那么必须约定一个参数作为结束标志。

当然，在C语言中，系统也提供了实现变参函数的宏，接下来就通

过下面一段代码来了解如何采用系统提供的宏来实现变参函数。

```
#include<stdio.h>
#include<stdarg.h>
void print(int n, ……)
{
int arg,i;
va_list p;
va_start(p,n);
for(i=0; i<n; i++)
{
arg=va_arg(p,int);
printf("%d\t", arg);
}
printf("\n");
va_end(p);
return;
}
int main()
{
print(3, 21, 32, 54);
return 0;
}
```

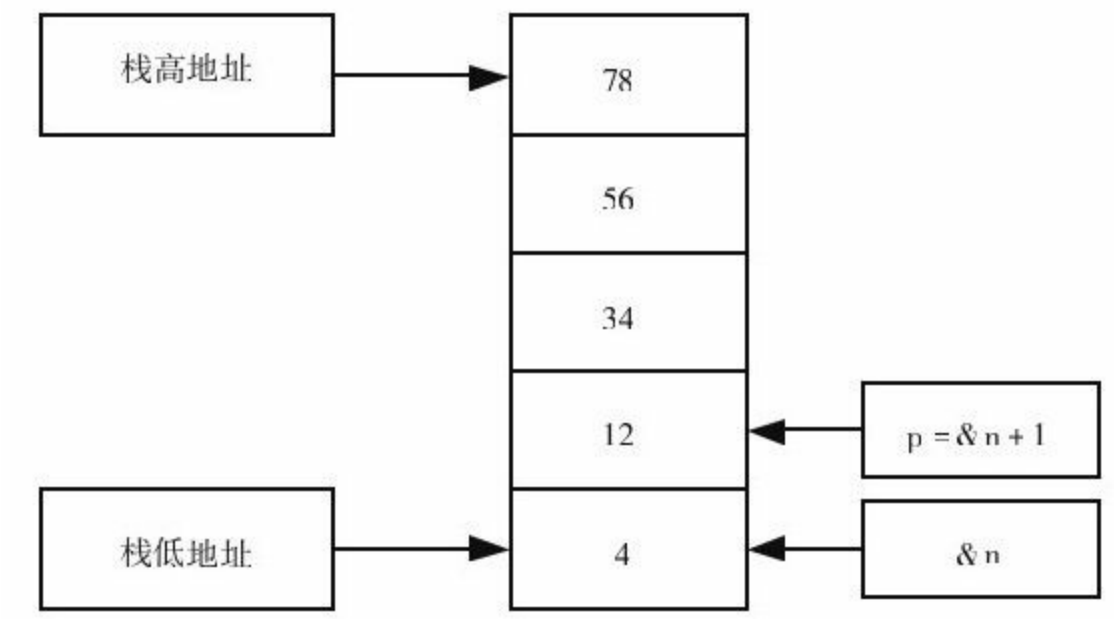

图　7-2　函数参数在栈中的存储结构

运行结果：

```
   21 32 54
```

在讲解变参函数的宏的实现方法之前，先来看系统实现变参函数的代码。

```
   typedef char*va_list;
   #define_INTSIZEOF（n）（（sizeof（n）+sizeof（int）-1）＆～
（sizeof（int）-1））
   #define va_start（ap,v）（ap=（va_list）＆v+_INTSIZEOF（v））
   #define va_arg（ap,t）（*（t*）（（ap+=_INTSIZEOF（t））-
_INTSIZEOF（t））)
   #define va_end（ap）（ap=（va_list）0）
```

由于va_list等价于char*，因此在print（）函数中定义的p就是一个字符型指针变量。接下来看宏_INTSIZEOF（n）的定义，这主要是为了实现内存中的字节对齐操作。宏va_start（ap,v）的作用是先得到变量v的地址，然后将其转换为char型指针，再加上变量v所占用的内存大小，使指针ap指向下一个参数，注意此时的指针为char类型的指针，所以接下来在使用宏va_arg（ap,t）的时候要将其强制转换为此时参数的类型t的指针。对于宏va_arg（ap,t）要注意的是，“ap+=_INTSIZEOF（t）”得到的是下一个参数的地址，再减去_INTSIZEOF（t）得到的当前参数的地址。通过一个for循环就可以一一取出其中压栈的所有参数，最后一个宏“#define va_end（ap）”的作用是清除ap指针，表明在接下来的部分不再使用该指针变量。

了解了系统实现变参函数的方法，就可以通过宏定义来实现变参函数了，看下面的代码。

```
#include<stdio.h>
typedef char*va_list;
#define va_start(ap,v) (ap=(va_list)&v+sizeof(v))
#define va_arg(ap,t) (*(t*)((ap+=sizeof(t))-sizeof(t)))
#define va_end(ap) (ap=(va_list)0)
void print(int n,……)
{
int arg,i;
va_list p;
va_start(p,n);
for(i=0;i<n;i++)
{
arg=va_arg(p,int);
printf("%d\t",arg);
}
printf("\n");
va_end(p);
return;
}
int main()
{
print(4,12,34,56,78);
return 0;
}
```

运行结果：

```
12 34 56 78
```

上面的实现方式不再调用系统提供的头文件，而是将其实现的宏提取出来，同时去掉了字节对齐操作，同样成功地实现了变参函数。但是这样会存在一个问题，因为函数的压栈操作是4字节对齐，所以如果这

里的参数不是int型，而是char型，那么就会出现错误，如：

```
#include＜stdio.h＞
typedef char*va_list;
#define va_start（ap,v）（ap=（va_list）&v+sizeof（v））
#define va_arg（ap,t）（*（t*）（（ap+=sizeof（t））-sizeof（t））
#define va_end（ap）（ap=（va_list）0）
void print（int n，……）
{
int arg,i;
va_list p;
va_start（p,n）;
for（i=0; i＜n; i++）
{
arg=va_arg（p,char）;
printf（"%c\t", arg）;
}
printf（"\n"）;
va_end（p）;
return;
}
int main（）
{
print（4, 'A', 'B', 'C', 'D'）;
return 0;
}
```

运行结果：

```
A
```

这时打印出来的只有字符“A”，后面的字符没有能够成功地打印出来，我们可以通过图7-3来加以分析。看看参数的压栈操作，先通过“va_start（p,n）；”使p指向函数的第一个参数，其中，p的值为&n+4，但是接下来使用“arg=va_arg（p,char）；”取参数时就出现了问

题，由于char型变量占用内存大小为1字节，而压栈操作采用的是4字节对齐，因此通过for循环就不可能取出后面的字符。

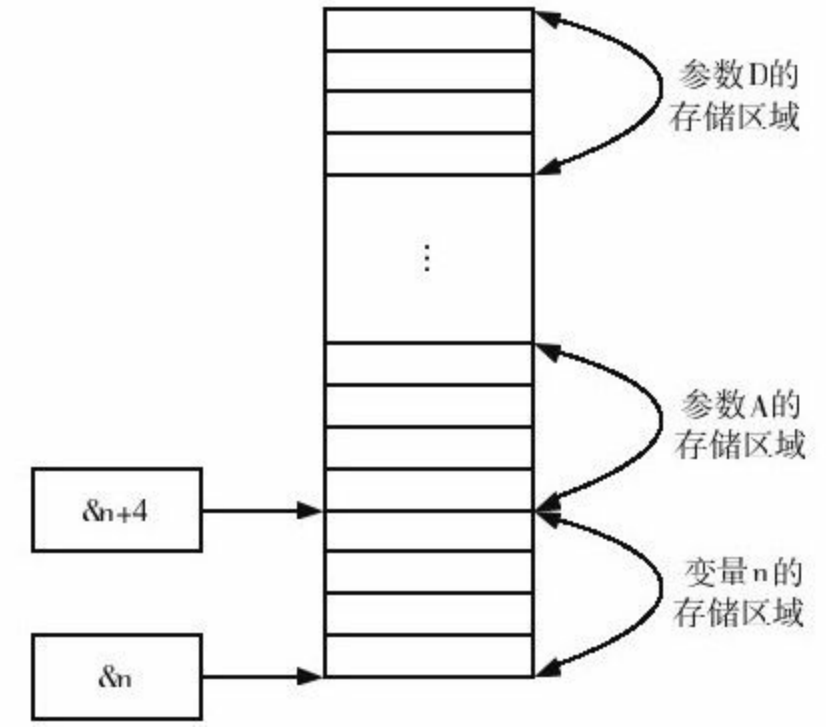

图　7-3　函数参数采用4字节对齐在栈中的存储结构

为了成功实现所有字符的打印操作，应该在代码中加上字节对齐的操作。下面的代码用于实现一个类似于printf（）的函数。

```
   #include<stdio.h>
   #include<stdlib.h>
   typedef char*va_list;
   #define_INTSIZEOF(n) ((sizeof(n)+sizeof(int)-1)&~
(sizeof(int)-1))
   #define va_start(ap,v) (ap=(va_list)&v+_INTSIZEOF(v))
   #define va_arg(ap,t) (*(t*)((ap+=_INTSIZEOF(t))-
_INTSIZEOF(t)))
   #define va_end(ap) (ap=(va_list)0)
   void myprintf(char*fmt,……)
   {
```

```
va_list p;
char c;
va_start (p,fmt) ;
do
{
c=*fmt;
if (c! ='%')
{
putchar (c) ;
}
else
{
switch (*++fmt)
{
case'd':
printf ("%d", * ( (int*) p) ) ;
break;
case'c':
printf ("%c", * ( (int*) p) ) ;
break;
case'f':
printf ("%3.2f", * ( (double*) p) ) ;
va_arg (p,int) ;
default:
break;
}
va_arg (p,int) ;
}
++fmt;
}while (*fmt! ='\0') ;
va_end (p) ;
return;
}
void main ()
{
int a=12;
short b=56;
char c='A';
double f=123.2;
myprintf ("a=%d\t b=%d\t c=%c\t f=%f\n", a,b, c,f) ;
return;
}
```

运行结果：

```
a=12 b=56 c=A f=123.20
```

从上面的代码可以看出，实现了字节对齐操作后，不管参数为何种类型，均可以成功地打印出来。对于变参函数的实现需要注意的就是参数的压栈采用的是4字节对齐。对于以上代码中的实现方法，有几个地方需要注意：如果参数是double类型，那么就应该多执行一次“va_arg（p,int）；”，因为double类型在内存中占用的内存大小为8字节；如果参数是float类型，压栈的时候采用的也是8字节对齐，如果采用它本身所占用的内存大小进行操作，打印出来的就是错误的结果。

7.3 函数指针的使用方法

在第1章讲解函数指针的时候，我们简要介绍了函数指针和指针函数的区别，同时也列举了几个简单的实例代码，接下来介绍函数指针在代码中的两种使用方法，一种是作为函数参数，而另外一种是用其调用函数。先来回顾一下函数指针的声明方法。

```
类型标识符（*指针变量名）（形参列表）；
```

“类型标识符”是函数的返回类型，由于“（）”的优先级高于“*”，因此指针变量名外的括号必不可少，后面的“形参列表”表示指针变量指向的函数所带的参数列表，例如：

```
int function（int x,int y）；/*声明一个函数*/
int（*f）（int x,int y）；/*声明一个函数指针*/
f=function；/*将function函数的首地址赋给指针f*/
```

赋值时，函数function不带括号，也不带参数，由于function代表函数的首地址，因此经过赋值以后，指针f就指向函数“function（int x,int y）；”代码的首地址。

下面的程序说明了函数指针调用函数的方法。

```
#include＜stdio.h＞
int max（int x,int y）
{
return x＞y?x：y；
```

```
}
int min（int x,int y）
{
return x＜y?x: y;
}
void main（）
{
int（*f）（int x,int y）=max;
printf（"max（2，6）=%d\tf（5，4）=%d\n"，max（2，6），f（5，4））；
f=min;
printf（"min（2，6）=%d\tf（5，4）=%d\n"，min（2，6），f（5，4））；
return;
}
```

运行结果：

```
max（2，6）=6（f）（5，4）=5
min（2，6）=2（f）（5，4）=4
```

f是指向函数的指针变量，所以可把函数max赋给f作为f的值，即把max（）的入口地址赋给f，以后就可以通过f来调用该函数。实际上，f和max都指向同一个入口地址，不同的是，f是一个指针变量，不像函数名称那样是固定的，它可以指向任何与它类型相同的函数。在程序中，把哪个函数的地址赋给f,f就指向哪个函数，而后用指针变量调用它，因此可以先后指向不同的函数。不过要注意，指向函数的指针变量没有++和--运算，使用时要小心。

接下来介绍另一种函数指针的声明方式，它可以分为以下两步。

定义函数指针类型：

```
typedef int（*fun_ptr）（int,int）；
```

声明变量并赋值：

```
fun_ptr max_func=max;
```

需要注意的是，有的编译器并不支持省略参数，这要视情况而定。通过这种声明方式定义一个函数指针，其指向的是返回值为int型，同时带有两个int型参数的函数。使用该方式定义的函数指针名可以作为一种定义类型，通过它来定义一个变量，该变量就是声明的函数指针类型，但是要注意赋给函数指针的函数应该和函数指针所指的函数原型是一致的。具体的运用通过下面的一段代码来了解。

```
#include＜stdio.h＞
void fun1 ()
{
printf ("This is fun1\n") ;
return;
}
void fun2 ()
{
printf ("This is fun2\n") ;
return;
}
void main ()
{
typedef void (*funp) () ;
funp fun=fun1;
fun () ;
fun=fun2;
fun () ;
return;
}
```

运行结果：

```
This is fun1
This is fun2
```

从上面的代码可以看到，先采用“typedef void（*funp）（）；”定义一个函数指针，接下来用funp定义一个该函数指针类型的变量fun，因为函数指针不同于函数名，函数指针可以灵活地指向与它具有相同类型的函数，因此对于在上面的代码中定义的两个具有相同类型的函数fun1和fun2，都采用函数指针的方式来调用，得到了正确的结果。需要注意的是，在使用函数指针调用函数的时候，唯一发生变化的是函数名。

讲解完通过函数指针调用函数的使用，接下来介绍函数指针作为函数参数的使用。

```
#include＜stdio.h＞
typedef int（*print）（int）；
int fun1（int i）
{
return（int）i;
}
void fun2（int n,print prt）
{
for（int k=0; k＜n; k++）
{
printf（"%d\t", prt（k））；
if（0==（k+1）%3）
printf（"\n"）；
}
return;
}
void main（）
{
int n=9;
```

```
fun2（n,fun1）;
return;
}
```

运行结果：

```
0 1 2
3 4 5
6 7 8
```

从上面的代码可以看到，定义了一个函数指针，在fun2函数中定义了一个函数指针类型的参数，所以在接下来调用函数指针的时候，对相应的函数指针参数调用一个与其类型一致的函数，成功地实现了通过函数指针作为参数对函数进行调用的操作。

最后看函数名，为什么通过函数名对函数指针进行赋值操作就可以用函数指针成功地实现函数的调用呢？首先需要明白的一个概念是，函数名本身就是一个地址，它的特殊之处就在于它是一个函数的入口地址，并且该入口地址有其特殊的地方，看下面的代码。

```
#include＜stdio.h＞
void print（）
{
printf（"Hello World\n"）;
return;
}
void main（）
{
printf（"&print=%d\n", &print）;
printf（"print=%d\n", print）;
printf（"**print=%d\n", **print）;
void（*pfun）（）;
pfun=print;
```

```
printf ("*pfun=%d\n", *pfun);
printf ("**pfun=%d\n", **pfun);
printf ("pfun=%d\n", pfun);
printf ("&pfun=%d\n", &pfun);
pfun ();
return;
}
```

运行结果：

```
&print=4198420
print=4198420
**print=4198420
*pfun=4198420
**pfun=4198420
pfun=4198420
&pfun=1245052
Hello World
```

从上面的运行结果可以发现，用函数名取地址得到的同样是函数的入口地址，而且在通过*号与函数名结合试图取出该地址的内容时发现，得到的仍然是该函数的入口地址，从而可以看出，函数名本身就是一个入口地址。接下来定义了一个函数指针，函数指针的特殊之处在于它有一个存储单元，用来存储函数的入口地址。但是当将函数指针指向一个函数的入口地址时，接下来对函数指针取值的操作得到的结果跟前面对函数名的操作所得结果完全一致。

7.4 函数之间的调用关系

关于函数之间的调用关系，不少初学者可能并不清楚，对函数调用的认识也只是表面上的。本节先根据系统提供的函数实现函数间的调用关系，明白了其实现原理之后，再实现函数间的调用关系。

在讲解之前，先来看以下几个函数。

backtrace（）函数

```
int backtrace(void**buffer,int size)
```

该函数用来获取当前线程的调用堆栈。获取的信息将会被存放在buffer中。buffer是一个指针列表，参数size用来指定buffer中可以保存多少个void*元素。函数返回值是实际获取的指针个数，最大不超过size的大小，在buffer中的指针实际是从堆栈中获取的返回地址，每一个堆栈框架有一个返回地址。

backtrace_symbols（）函数

```
char**backtrace_symbols(void*const*buffer,int size)
```

该函数将backtrace（）函数获取的信息转化为一个字符串数组，其中，参数buffer就是在backtrace（）函数中获取的buffer参数，size是数

组中的元素个数，也就是backtrace（）函数的返回值。

backtrace_symbols（）函数的返回值是一个指向字符数组的指针，其大小同buffer相同，数组中的每个元素都包含函数名、函数偏移地址、实际的返回地址。需要注意的是，该函数的返回指针是通过malloc函数申请的空间，因此在调用该函数后还要使用free（）函数来释放该函数返回的指针。

backtrace_symbols_fd（）函数

```
void backtrace_symbols_fd(void*const*buffer,int size,int fd)
```

该函数与上面的backtrace_symbols（）函数的功能一样，区别在于该函数没有返回值，不必采用malloc函数分配内存空间，而是将结果写入文件描述符fd的文件中，每个函数对应文件中的一行。

接下来通过前两个函数来查看函数间的调用关系，代码如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<execinfo.h>
#define MAX_LEVEL 10
void call_2()
{
int i=0;
void*buffer[MAX_LEVEL]={0};
int size=backtrace(buffer,MAX_LEVEL);
char**strings=backtrace_symbols(buffer,size);
printf("Obtained%zd stack frames.\n", size);
for(i=0; i<size; i++)
printf("%s\n", strings[i]);
free(strings);
```

```
return;
}
void call_1 ()
{
call_2 () ;
return;
}
void call ()
{
call_1 () ;
return;
}
int main (int argc,char*argv[])
{
call () ;
return 0;
}
```

运行结果：

```
root@ubuntu:/home#gcc backtr.c-rdynamic-o backtr
root@ubuntu:/home#./backtr
Obtained 6 stack frames.
./backtr (call_2+0x35) [0x80486b9]
./backtr (call_1+0xb) [0x804872a]
./backtr (call+0xb) [0x8048737]
./backtr (main+0xb) [0x8048744]
/lib/i386-linux-gnu/libc.so.6 (__libc_start_main+0xe7) [0x737e37]
./backtr () [0x80485f1]
```

分析上面的代码可知，这里采用backtrace_symbols（）函数来实现打印，当然也可以采用backtrace_symbols_fd（）函数实现将结果保存到一个文件中。在编译的时候指定了一个-rdynamic选项，该选项的功能是支持函数的功能名。从运行结果可以看出，得到的是一些不直观的偏移地址和实际的返回地址，可以将它们转换为代码中的相对位置，这可以通过addr2line工具来实现，接下来简单地讲解一下该工具的使用方法。

要通过实际地址得到其在源代码中的相对位置，可以使用如下指令：

```
addr2line[-f]-e可执行文件名 实际地址
```

需要注意的是，在使用addr2line工具的时候，需要在编译可执行文件的时候带有-g选项，其中[-f]为可选项，其功能为打印出实际地址所在的函数名称。例如选用上面的第一个实际地址，执行的指令及得到的结果如下：

```
root@ubuntu:/home#addr2line-f-e backtr 0x80486b9
call_2
/home/backtr.c: 11
```

从运行结果可以看出，该地址位于call_2函数中，位于代码的第11行。

在讲解backtrace（）函数的实现原理之前，先来看看函数调用过程中的压栈操作。第1章讲解堆栈时，提到了临时变量的压栈，接下来通过图7-4来分析函数调用过程中的压栈操作。

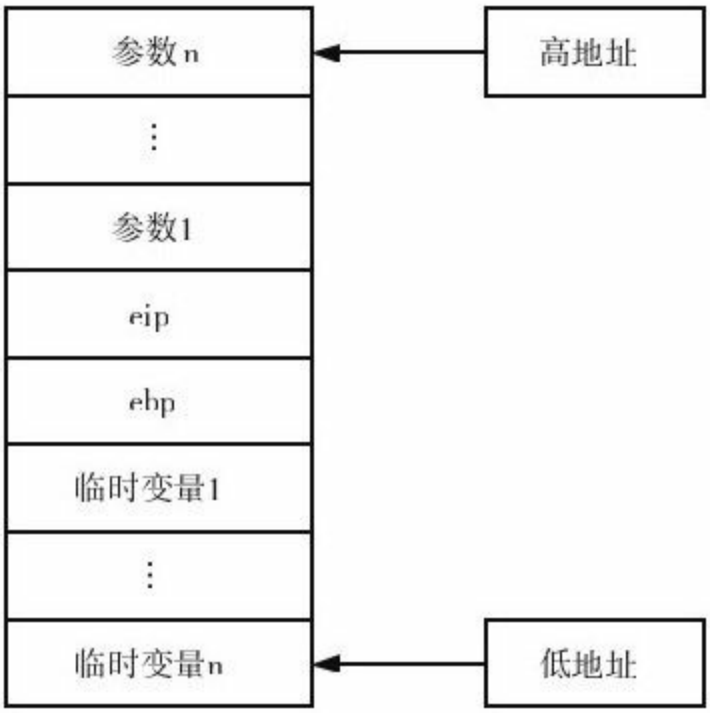

图　7-4　函数调用过程中的压栈操作

从图7-4可以看出，函数参数的压栈是从右向左进行的，而临时变量的压栈是从上到下进行的，对临时变量中的数组压栈同样是从右向左进行的，即首先压栈的是下标最大的元素，最后压栈的是第一个数组元素。在函数参数和临时变量的压栈之间还有两个值，一个是ebp寄存器的值，另外一个是eip寄存器的值。ebp寄存器保存的是上一个函数的ebp的压栈地址，通过该ebp就能够取出上一个调用函数的ebp值。eip寄存器保存的是CPU下次所要执行的指令地址。ebp寄存器存储的是栈底地址，而这个地址是由esp在函数调用前传递给ebp的。等到调用结束，ebp会把其地址再次回传给esp，这样esp又一次指向了函数调用结束后，即栈顶的地址，通过这样的方法就实现了函数之间的调用。为了加深对

栈结构的印象，先来看下面的代码。

```
    #include＜stdio.h＞
    void test（int a,int b,int c）
    {
    int arr[3]={1，2，3};
    printf（"函数参数地址：&a=%d\t&b=%d\t&c=%d\n"，&a，&b，&c）；
    printf（"临时变量地址：&arr[2]=%d\t&arr[1]=%d\t&arr[0]=%d\n"，&
arr[2]，&arr[1]，&arr[0]）；
    return;
    }
    int main（）
    {
    test（1，2，3）；
    return 0;
    }
```

运行结果：

```
    函数参数地址：&a=1244968&b=1244972&c=1244976
    临时变量地址：&arr[2]=1244956&arr[1]=1244952&arr[0]=1244948
```

对于上面的运行结果，可以通过图7-5来演示函数的压栈操作，函数参数的压栈和临时变量中数组元素的压栈都符合上面的分析。第一个函数参数地址和数组最后一个元素a[2]地址之差为12，因为这12字节的存储空间中有4字节是数组元素arr[2]的，剩下的8字节存储空间是ebp和eip的。

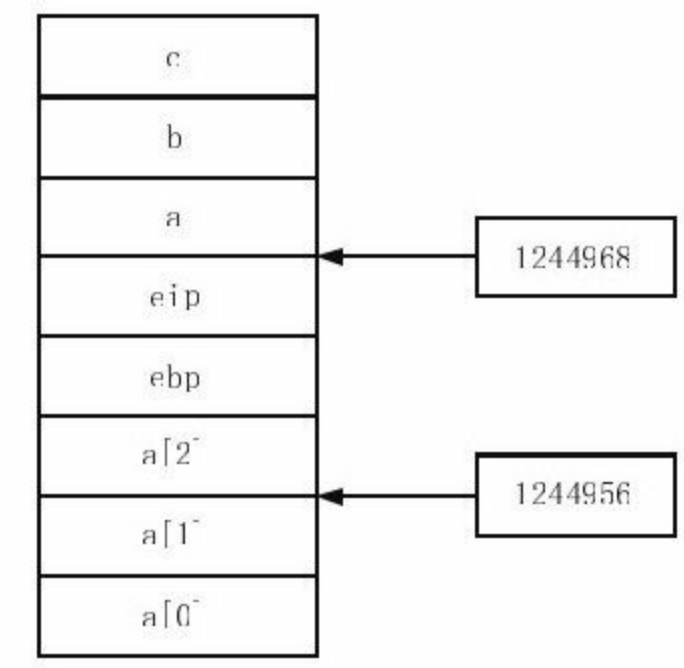

图　7-5　函数调用时压栈的存储结构

分析完参数的压栈操作，再来验证当前函数中压栈的ebp寄存器的值是否为上一个函数压栈的ebp寄存器的地址，看下面的代码。

```
#include＜stdio.h＞
void test1 ()
{
int a;
printf ("test1 () 函数中ebp的值为：%d\n", * (&a+1) );
return;
}
void test ()
{
int n;
printf ("test () 函数中存放ebp寄存器的内存单元地址为：%d\n", &n+1) ;
test1 () ;
return;
}
int main (int argc,char*argv[])
{
test () ;
return 0;
}
```

运行结果：

```
test（）函数中存放ebp寄存器的内存单元地址为：1244972
test1（）函数中ebp的值为：1244972
```

从上面的运行结果可以看出，每个函数中压栈的ebp寄存器的值都是上一个函数中ebp值所在内存单元的地址。了解了函数的压栈操作，接下来实现函数间的调用关系，看下面的代码。

```
#include＜stdio.h＞
#define MAX_LEVEL 4
int backtrace（void**buffer,int size）
{
int n;
int*p=&n;
int i=0;
int ebp=*（p+5）;
int eip=*（p+6）;
for（i=0; i＜size; i++）
{
buffer[i]=（void*）eip;
p=（int*）ebp;
ebp=*p;
eip=*（p+1）;
}
return size;
}
void call2（）
{
int i=0;
void*buffer[MAX_LEVEL]={0};
int size=backtrace（buffer,MAX_LEVEL）;
for（i=0; i＜size; i++）
{
printf（"called by%p\n", buffer[i]）;
}
return;
}
void call1（）
```

```
{
call2 () ;
return;
}
void call ()
{
call1 () ;
return;
}
int main ()
{
call () ;
return 0;
}
```

运行结果：

```
root@ubuntu:/home#gcc-g backtr.c-o backtr
root@ubuntu:/home#./backtr
called by 0x804848a
called by 0x80484c7
called by 0x80484d4
called by 0x80484e1
```

分析上面的代码，首先需要注意的是早期gcc版本中的压栈操作与新版本稍有不同，主要体现在对ebp和eip的压栈处理上。早期的压栈与VC的压栈方法相同，就是在函数参数和临时变量之间压栈，而新版本则把ebp和eip的压栈操作放到了最后一个临时变量的后面。上面的代码是采用新版本的gcc编译的，由于backtrace（）函数中有5个临时变量，因此先通过整型指针p取出第一个临时变量的地址，对其加5，指向ebp所在的内存单元地址，然后将其值放到ebp变量中，最后通过一个for循环来逐一取出每个函数中eip的值。

可以对上面结果中的实际地址使用addr2line工具获得其在文件中的相对位置，如：

```
root@ubuntu:/home#addr2line-f-e backtr 0x804848a
call2
/home/backtr.c: 30
```

7.5 函数的调用方式及返回值

关于函数的调用类型，在此之前都已经接触过，但是并没有对其进行归类，接下来看函数的几种调用方式。

1.表达式的调用方式

对于函数在表达式中的调用，可以通过下面的一段代码来学习。

```
#include<stdio.h>
int max(int a,int b)
{
return a>b?a: b;
}
int min(int a,int b)
{
return a>b?b: a;
}
int main()
{
int a,b;
int abs;
printf("请输入a的值: ");
scanf("%d", &a);
printf("请输入b的值: ");
scanf("%d", &b);
abs=max(a,b)-min(a,b);
printf("输入的两数之差的绝对值为: %d\n", abs);
return 0;
}
```

运行结果：

```
请输入a的值：-9
请输入b的值：96
输入的两数之差的绝对值为：105
```

上面的函数调用采用的是通过函数表达式的方法。由于函数的返回值为整数，因此可以将其返回值视为一个整型数据，使其参与表达式的运算。在使用表达式方法调用函数的时候，需要注意函数的返回值，这时会将其函数作为一个返回值类型的变量来参与运算，当表达式执行函数的时候，函数会被调用，执行函数体。

2.参数的调用方式

除了通过函数表达式调用函数外，还可以通过函数参数的方法来调用函数，但是此时函数是作为函数的实参来调用的，看下面的代码。

```
#include＜stdio.h＞
#define N 10
char*str（）
{
char*string="Hello World! ";
return string;
}
int sum（int n）
{
int i;
int total;
total=0;
for（i=1; i＜=n; i++）
total+=i;
return total;
}
int main（）
{
printf（"函数的返回值为字符串\n"）;
printf（"%s\n", str（））;
printf（"函数的返回值为整数\n"）;
printf（"%d\n", sum（N））;
return 0;
}
```

运行结果：

```
函数的返回值为字符串
Hello World!
函数的返回值为整数
55
```

在上面的代码中，通过把函数作为形参的方法来对函数进行调用，同样需要注意的是，在使用函数作为实参的时候，将函数作为一个返回值类型的变量来进行处理。从这里可以看出，函数的返回值决定了函数的调用方式。

3.函数语句的调用方式

如果函数的返回值为空，那么就不可以采用上面的两种方法对其进行调用，这时只能通过函数语句的方法来对其进行调用，如：

```
#include＜stdio.h＞
#define N 10
int factorial（int n）
{
int i;
int total;
total=1;
for（i=1; i＜=n; i++）
total*=i;
return total;
}
void print（char*str）
{
printf（"%s\n", str）;
return;
}
int main（）
```

```
{
factorial（N）;
print（"Hello World! "）;
return 0;
}
```

运行结果：

```
Hello World!
```

分析上面的运行结果可知，对函数的调用采用的是函数语句的调用方式。需要注意的是，对于那些返回值为非空的函数，虽然也可以采用这种方式，但是没有实际意义，所以一般不要对返回值为非空的函数采用这种方法。如果返回值为空，那么应该采用这种方法。

从以上三种方式的函数调用中发现，函数的返回值决定了函数调用方式。接下来分析函数的返回值，正如前面所讲解的，函数可以返回空值和非空值，对于返回值为空的函数，在此就不再讲解了，重点看看返回值为非空的函数，其返回类型可以是int、float、double等一般数据类型，也可以是指针、自定义结构体等复杂类型。

如果函数的返回类型和函数指定的返回类型不一致，会出现什么情况呢？看下面的代码。

```
#include＜stdio.h＞
#define N 10
double power（int a,int n）
{
int i;
```

```
int total;
total=1;
for（i=1; i＜=n; i++）
total*=a;
return total;
}
char print（）
{
int a;
a=0x12345865;
return a;
}
int main（）
{
printf（"%2.2lf\n", power（2, N））;
printf（"%c\n", print（））;
return 0;
}
```

运行结果：

```
1024.00
e
```

以上代码在power函数中，定义的函数返回类型是double类型，但是在函数体中，采用的却是返回int型的total，虽然得到的是正确的返回值，但是类型已经是double了，所以如果在函数体中返回的不是定义的函数返回类型，那么编译器在编译的时候将会对其进行强制转换处理。在下面的print函数中定义的返回类型是char型，但是在函数体中返回的却是int型。分析运行结果可知，编译器在编译的过程中对返回值进行了截取低字节的处理，定义的int型变量a的值为0x12345865，截取的低字节为0x65，转换为十进制是101，刚好是字符e的ASCII码。因此在使用函数返回值的时候，需要留意这些隐含转换处理，虽然没有明确指出处

理方法，但是编译器在编译的过程中会自动对其进行转换。

第8章　文件

本章主要针对C语言初学者对文件操作中一些模糊不清的概念和文件读写操作中的易错点进行相应的讲解，如文件指针到底意味着什么，如何区分文件的读操作是否结束，如何选择读写函数，以及如何在文件的操作中进行出错检测等，读者可以带着以上这些问题开始本章的阅读。

8.1 文件及文件指针

所谓文件，通常指的是存储在外表介质上的一组相关信息数据的集合。可以从以下两个方面对文件进行分类。

（1）根据文件的读写方式可分为随机文件和顺序文件

随机文件：对这类文件中数据的读写是随机的，只要按照相关的函数对所要读写的文件进行定位，就可以对其进行读写操作，即允许跳跃式地对所需要的文件位置的数据进行处理。

顺序文件：对这类文件的读写操作是顺序进行的，如果要对文件中某个位置的数据进行读写操作，那么必须也对它前面的数据进行相应的操作，不允许跳跃式地对数据进行处理。

（2）根据文件中数据存放的格式可分为文本文件和二进制文件

文本文件：这类文件中数据的存储方式是将数据转换为相应的ASCII码来存放。例如，对于整数5236，按照文本文件的存储方式需要占用4字节。如图8-1所示为文本文件对5236的存储方式，每个字符都用相应的ASCII码来存储，并占用1字节，所以5236总共占用4字节的大小。

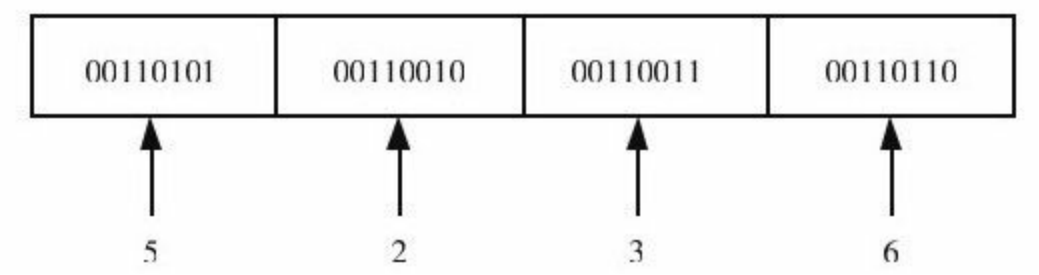

图　8-1　文本文件中数据的存储方式

二进制文件：这类文件中数据的存放都是按照二进制进行的。还是以5236为例，按照二进制来存放仅仅占用2字节。如图8-2所示为二进制文件对5236的存储方式。

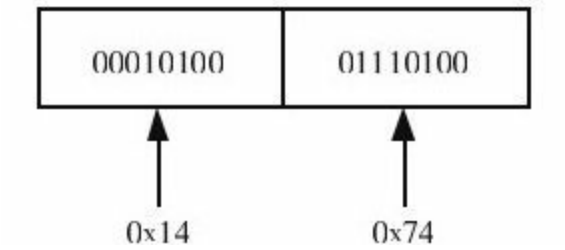

图　8-2　二进制文件中数据的存储方式

对比文本文件和二进制文件的存储方式，可以发现，在采用文本文件时，存储的每个字符都要转换为ASCII码，所以比二进制文件花费了更多的转换时间，同时，所占用的存储空间也比二进制文件大，但是采用文本文件便于对单个字符进行处理，也便于输出。

文件指针，指的是一种特殊类型的指针，该指针的特殊之处在于它指向的是文件，而不是之前讲解的一般数据类型。通过该指针可以对文件进行读写等相关操作。定义文件指针的一般形式如下：

```
FILE*文件指针变量名;
```

FILE类型是在stdio.h中声明的结构体类型，因为本书的编译环境是VC++6.0，所以介绍一下VC++6.0下的FILE结构体类型，其形式如下：

```
struct_iobuf{
char*_ptr;
int_cnt;
char*_base;
int_flag;
int_file;
int_charbuf;
int_bufsiz;
char*_tmpfname;
};
typedef struct_iobuf FILE;
```

上述FIEE结构体中每个成员的含义如下：_ptr字符指针表示文件输入的下一个位置；_cnt表示当前缓冲区的相对位置；_base字符指针表示文件的起始位置；_flag表示文件标志；_file表示文件的有效性验证；_charbuf表示缓冲区的检查，若无此成员，则不读取；_bufsize表示文件大小；_tmpfname表示临时文件名。利用上面的FILE结构体类型定义的文件指针是一个FILE结构体类型的指针变量，通过该指针变量，可以访问它所执行的文件，进而对其进行读写操作。

对于文件的操作，一般有以下4个步骤：

1）定义文件指针；

2）打开文件；

3）读写文件；

4）关闭文件。

打开文件时使用fopen函数，其函数原型为：

```
FILE*fopen（const char*fname,const char*mode）；
```

注意fopen函数的返回值是一个文件类型的指针，其中，参数fname是要打开的文件名，该参数可以带存储路径。参数mode指示文件的打开模式，指示文件打开的基本操作。mode参数的取值主要由表8-1中的6个字符组合而成。

将以上字符进行组合可以得到多个mode值，但是有些字符是不能组合的，如b和t,a和r等。以下是一些mode值的说明。

表 8-1　mode 参数的取值

字　符	含　义
r(read)	具有文件的读权限
w(write)	具有文件的写权限
a(append)	在文件的末端添加数据
b(binary)	文件类型为文本文件
t(text)	文件的类型为二进制文件
+	具有文件的读写权限

r：以只读的方式打开文件，如果不存在，就返回NULL；

r+：以读写方式打开文件，如果不存在，就返回NULL；

a：追加文件并进行写操作，如果不存在，就创建文件，否则新增的内容被添加到文件的末端；

rt：以只读方式打开一个文本文件；

wt：以写方式打开一个文本文件，如果不存在，则新建该文件；

rt+：以读写方式打开一个文件；

wt+：以读写方式打开一个文件，如果不存在，则新建该文件。

对打开的文件进行读写操作之后，接下来对文件进行关闭操作。关闭文件使用的是fclose函数，其一般格式如下：

```
int fclose（FILE*fp）；
```

如果文件关闭成功，则返回0，否则返回-1。使用fclose（）函数可以把缓冲区内最后剩余的数据输出到磁盘文件中，并释放文件指针和有关的缓冲区。如果没有使用fclose函数关闭文件，那么可能因为缓冲区中的数据没有满而导致数据丢失，从而造成错误，所以对于那些操作完的文件一定要使用fclose函数进行关闭操作。接下来看下面的代码：

```
#include＜stdio.h＞
int main（）
{
```

```
FILE*fp;
fp=fopen（"edit.txt", "r"）;
if（NULL==fp）
{
printf（"文件edit.txt打开失败! \n"）;
fp=fopen（"edit.txt", "wt"）;
if（NULL==fp）
{
printf（"文件edit.txt打开失败! \n"）;
return 0;
}
}
printf（"文件edit.txt打开成功! \n"）;
fclose（fp）;
return 0;
}
```

运行结果：

```
文件edit.txt打开失败!
文件edit.txt打开成功!
```

从上面的运行结果可知，一开始没有在当前文件目录下建立文件edit.txt，所以最初采用rt模式打开文本文件时，fopen函数的返回值为NULL。但是，接下来采用wt模式打开edit.txt的时候，如果不存在edit.txt，那么将会自动创建该文件，因此显示成功打开了edit.txt文件。在使用mode模式获取对于文件的相关操作权限的时候，需要注意模式本身所代表的含义。

8.2 EOF和FEOF的区别

通过上一节的讲解，读者对文件应该有了大概的认识，接下来介绍在文件操作中经常使用到的EOF和FEOF究竟有什么样的区别。

首先，读者需要明白EOF是C语言中的一个宏，相应的实现代码如下：

```
#define EOF（-1）
```

EOF是end of file的缩写，其值为-1。从字面上也可以知道它以类作为判断文件结束的标志，但是需要注意的是，它只能作为文本文件的结束标志，因为文本文件的存储方式是将字符转换为ASCII码，而ASCII码中不会出现-1。EOF不仅可以用来检测文件是否结束，还可以用来判断函数是否调用成功，下面通过一段代码来看其具体的使用。

```
#include＜stdio.h＞
void copy（FILE*fpin,FILE*fpout）
{
char ch;
ch=getc（fpin）;
while（! feof（fpin））
{
putc（ch,fpout）;
ch=getc（fpin）;
}
return;
}
int main（）
{
FILE*fpin，*fpout;
```

```
fpin=fopen（"text1.txt", "r"）;
fpout=fopen（"text2.txt", "w"）;
int c;
while（（c=getchar（））！=EOF）
{
putchar（c）;
c=getchar（）;
}
copy（fpin,fpout）;
fclose（fpin）;
fclose（fpout）;
return 0;
}
```

先在当前目录下建立一个text1.txt，在其中输入“hello world！”，由于要测试在按下Ctrl+C组合键之后代码是退出循环执行接下来的程序还是直接退出程序，因此在while循环的后面调用了一个copy函数，其功能是将text1.txt中的内容写入text2.txt中。运行代码，由于程序中有一个while循环，因此可以进行输入，只要输入的不是EOF，就会不停地执行while循环体，但是当按下Ctrl+C组合键时窗口就消失了。为了验证是否执行接下来的语句，打开当前目录后发现，目录下多了一个text2.txt，其中的内容和text 1.txt中的完全相同。所以，EOF不仅可以用来判断文本文件是否结束，还可以用来判断函数是否得到成功调用。

但是在二进制文件中，通常不采用EOF来判断是否到达文件的末端，因为二进制文件中会出现-1值，如果没有到达末端就出现了-1值，那么就会出现误判的情况。先来看下面的代码。

```
#include＜stdio.h＞
void copy（FILE*fpin,FILE*fpout）
{
```

```
char ch;
while ( (ch=getc (fpin) ) ! =EOF)
{
putc (ch,fpout) ;
}
return;
}
int main ()
{
FILE*fpin, *fpout;
char b=0x34;
char a=0xff;
fpin=fopen ("dat1.dat", "wb") ;
fpout=fopen ("dat2.dat", "wb") ;
fputc (b,fpin) ;
fputc (a,fpin) ;
fputc (b,fpin) ;
fclose (fpin) ;
fpin=fopen ("dat1.dat", "rb") ;
copy (fpin,fpout) ;
fclose (fpin) ;
fclose (fpout) ;
return 0;
}
```

在上面的代码中，先以二进制文件的格式打开dat1.dat，对其进行写操作，写入的3个字符的ASCII码分别是0x34、0xff、0x34。然后再次打开该文件，对其进行读操作，将其中的字符读取到dat2.dat中。运行完程序后打开dat1.dat和dat2.dat，发现它们的内容并不一样，在dat1.dat中是“44”，而在dat2.dat中的是“4”，由此发现0xff被视为结束符，从而判定文件已到末端，不再对后面的数据进行读写操作，所以后面的数据丢失了。但如果适当修改上面的代码，那么同样可以用EOF作为二进制文件结束符的判断标准，修改方法是将函数copy中的“char ch”改为“short ch”，即：

```
void copy（FILE*fpin,FILE*fpout）
{
short ch;
while（（ch=getc（fpin））！=EOF）
{
putc（ch,fpout）；
}
return;
}
```

修改为以上代码之后再来看运行结果，这时发现能够完整地将dat1.dat中的数据写入到dat2.dat中去了。下面来分析其实现的原因，由于定义的是short型，占用2字节，读取的0xff实际变为0x00ff，这个值不再是-1（0xffff），所以可以得到完整的数据信息。

通常，对于二进制文件的操作都是采用feof宏来判断的。feof宏是一个带参数的宏，接下来介绍feof宏的实现。

```
#define_IOEOF 0x0010
#define feof（_stream）（（_stream）->_flag&_IOEOF）
```

需要注意的是，文件指针的位置是由其中的结构体成员变量_ptr来标识的，只有文件指针已经到达文件的末端再进行读写操作时，其中的文件状态标识符才会被置为_IOEOF，这个时候再调用FEOF宏才会得到其返回值16，表示到了文件的末端。看下面的代码。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin，*fpout;
char b=0x34;
fpin=fopen（"dat1.dat"，"wb"）；
```

```
fputc（b,fpin）；
fclose（fpin）；
fpin=fopen（"dat1.dat", "rb"）；
char c;
while（! feof（fpin））
{
c=getc（fpin）；
printf（"%X\n", c）；
}
fclose（fpin）；
return 0;
}
```

运行结果：

```
34
FFFFFFFF
```

可以发现上面的运行结果中多了一个FFFFFFFF，正如前面介绍的，由于在读到最后一个字符时再次对文件进行读写操作，其中的文件状态标识符才会被置为_IOEOF，因此多输出了一个FFFFFFFF。适当修改上面的代码来去除FFFFFFFF，如下：

```
#include＜stdio.h＞
int main（）
{
FILE*fpin, *fpout;
char b=0x34;
fpin=fopen（"dat1.dat", "wb"）；
fputc（b,fpin）；
fclose（fpin）；
fpin=fopen（"dat1.dat", "rb"）；
char c;
c=getc（fpin）；
while（! feof（fpin））
{
printf（"%X\n", c）；
c=getc（fpin）；
```

```
}
fclose（fpin）;
return 0;
}
```

运行结果：

```
34
```

所以，在使用FEOF的时候要尤其注意这点，否则就会在对文件进行读写操作时得到一些不需要的数据。

需要注意的是，FEOF不仅可以用来判断二进制文件是否结束，还可以用来判断文本文件是否结束。如果遇到文件结束符，那么FEOF的返回值为16，否则为0。可以通过下面的代码来测试。

```
#include＜stdio.h＞
void main（）
{
FILE*fp;
fp=fopen（"text1.txt", "r"）;
if（NULL==fp）
{
printf（"打开失败！\n"）;
return;
}
char ch;
printf（"feof作为文本文件结束的判断标志\n"）;
ch=getc（fp）;
while（！feof（fp））
{
ch=getc（fp）;
printf（"%d\n", feof（fp））;
}
fclose（fp）;
fp=fopen（"text2.txt", "rb"）;
if（NULL==fp）
```

```
{
printf（"打开失败！\n"）；
return;
}
printf（"feof作为二进制文件结束的判断标志\n"）；
ch=getc（fp）；
while（！feof（fp））
{
ch=getc（fp）；
printf（"%d\n", feof（fp））；
}
fclose（fp）；
return;
}
```

运行结果：

```
feof作为文本文件结束的判断标志
0
0
16
feof作为二进制文件结束的判断标志
0
0
16
```

在上面的代码中，将FEOF作为文本文件和二进制文件结束的判断标志。首先在当前的路径下建立两个文件text1.txt和text.txt，在两个文件中随机输入3个字符。分析运行结果，在没有到达文件末端的时候，文本文件和二进制文件都打印出0，在到达文件末端时打印出16，由此可知，使用FEOF宏可以作为二进制文件和文本文件结束的判断标志，没有结束时返回值为0，结束后返回值为16。

8.3 读写函数的选用原则

本章第8.1节提过，对文件的操作主要分为4个步骤，前面讲解了打开和关闭操作，接下来介绍有关文件的读写操作。对文件的读写操作可以按以下4条原则来选取函数。

1.按照字符进行读写

先来看按照字符进行读写操作的两个函数：fputc和fgetc。

（1）fputc函数

```
int fputc（int ch,FILE*fp）;
```

功能：送一个字符到指定的文件中。

返回值：写入成功则返回写入的字符，失败则返回EOF。

（2）fgetc函数

```
int fgetc（FILE*fp）;
```

功能：从文件指针fp指向的文件中读取一个字符。

返回值：返回读取的一个字节。如果读到文件末尾就返回EOF。

接下来通过代码来看上述两个函数的使用。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin;
fpin=fopen（"text.txt", "w"）;
if（NULL==fpin）
{
printf（"打开失败！\n"）;
return 0;
}
int i;
char a=0x41;
char b;
for（i=0; i＜26; i++）
{
b=a+i;
fputc（b,fpin）;
}
fclose（fpin）;
return 0;
}
```

在上面的代码中，通过fput函数向文件text.txt中写入26个大写的英文字母，运行程序后，打开当天目录下的文件text.txt发现成功实现了字符的写入。接下来通过fgetc函数将上面写入文件的字符读出来并保存到一个数组中，最后将其打印输出，代码如下：

```
#include＜stdio.h＞
int main（）
{
FILE*fpout;
fpout=fopen（"text.txt", "r"）;
if（NULL==fpout）
{
printf（"打开失败！\n"）;
return 0;
}
```

```
int i=0;
char a[26];
char c;
while ( (c=fgetc (fpout) ) ! =EOF)
{
a[i]=c;
printf ("%c", a[i]) ;
i++;
}
fclose (fpout) ;
return 0;
}
```

运行结果：

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

从上面的运行结果可以看出，成功地实现了将text.txt文件中的数据读到数组a中。在上面的代码中，采用的是EOF来判断当前对打开的文本文件的读操作是否到达文件的末端。

2.按照字符串进行读写

按照字符串进行读写操作的两个函数为：fgets和fputs。

（1）fgets函数

```
char*fgets (char*s,int size,FILE*fp) ;
```

功能：该函数从参数fp所指的文件内读入字符并存到参数s所指的内存空间，直到出现换行字符、读到文件尾或已读了size-1个字符为

止，最后加上NULL标示字符串结束。

返回值：读写成功则返回s指针，否则返回NULL。

（2）fputs函数

```
int fputs（char*string,FILE*fp）；
```

功能：将string所指向的字符串写入fp所指定的文件中去。

返回值：读写成功返回为0，否则返回非0值。

接下来通过代码介绍上述两个函数的使用。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin;
char*string[4]={"Hello world! ", "Hello
Bigloomy! ", "Happy", "Gloomy"};
fpin=fopen（"text.txt", "w"）；
if（NULL==fpin）
{
printf（"打开失败！\n"）；
return 0;
}
int i=0;
for（i=0; i＜4; i++）
fputs（string[i], fpin）；
fclose（fpin）；
return 0;
}
```

在上面的代码中，通过fputs函数向打开的文件中一次输入一个字符

串。运行代码，打开当前目录下的text.txt，发现其保存的数据信息与所输入的完全一致。接下来再看如何通过fgets函数从上面的文件中获取写入的字符串。

```
#include＜stdio.h＞
int main（）
{
FILE*fpout;
char string[4];
fpout=fopen（"text.txt", "r"）;
if（NULL==fpout）
{
printf（"打开失败！\n"）;
return 0;
}
while（！feof（fpout））
{
fgets（string, 4, fpout）;
printf（"%s", string）;
}
fclose（fpout）;
return 0;
}
```

运行结果：

```
Hello world! Hello Bigloomy! HappyGloomy
```

上面的运行结果与此前写入的数据信息完全一致，但是需要留意fgets函数，其中的参数n并不是每次读取的字符数，每次读取的字符数是n-1，最后再添加一个回车符。

3.按照磁盘文件进行读写

按照磁盘进行读写操作的两个函数为：fprintf和fscanf。

（1）fprintf函数

```
int fprintf（FILE*fp,char*format[, argument]）
```

功能：fprintf（）函数根据指定的format（格式）写入（参数）到由fp（文件指针）指定的文件中，fprintf（）只能和printf（）一样工作。

返回值：读写成功则返回输出的字符数，错误则返回负数。

（2）fscanf函数

```
int fscanf（FILE*fp,char*format，[argument……]）；
```

功能：向fp文件指针所指向的文件中写入字符。

返回值：读写成功则返回实际读出的数据个数，若文件中没有数据则返回0，失败则返回EOF。

接下来通过代码来看上述两个函数的使用。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin;
fpin=fopen（"text.txt", "w"）；
if（NULL==fpin）
```

```
{
printf（"打开失败！\n"）；
return 0;
}
int i,j;
for（i=1；i＜=9；i++）
{
for（j=1；j＜=i；j++）
{
fprintf（fpin，"%d X%d=%d\t"，j,i，j*i）；
}
fprintf（fpin，"\n"）；
}
fclose（fpin）；
return 0;
}
```

在上面的代码中，采用fprintf函数向打开的text.txt文件中输入九九乘法表，运行程序后打开text.txt，发现已经成功在文件中输入了一个九九乘法表。接下来通过fscanf函数对上面的text.txt文件进行读操作，并且将读取的九九乘法表打印出来。

```
#include＜stdio.h＞
int main（）
{
FILE*fpout;
fpout=fopen（"text.txt"，"r"）；
if（NULL==fpout）
{
printf（"打开失败！\n"）；
return 0;
}
int i,j;
int a,b，c;
for（i=1；i＜=9；i++）
{
for（j=1；j＜=i；j++）
{
fscanf（fpout，"%d×%d=%d\t"，＆a，＆b，＆c）；
printf（"%d×%d=%d"，a,b，c）；
}
```

```
    printf（"\n"）;
    }
    fclose（fpout）;
    return 0;
    }
```

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
```

从上面的运行结果中可以看到，成功地实现了通过fscanf函数将文件text.txt中的数据读取出来，并且采用printf函数进行了打印输出。

4.按组进行读写

按组进行读写操作的两个函数为：fwrite和fread。

fwrite函数如下：

```
    size_t fwrite（const void*buffer,size_t size,size_t
count,FILE*fp）;
```

功能：将一个数据块写入fp文件指针指向的文件中，其中，buffer参数是所要输出数据的地址，size是每次所要写入的字节数，count是所

要写入的次数，fp是所要写入的目标文件指针。

返回值：读写成功则返回写入的字节数。

fread函数如下：

```
size_t fread（void*buffer,size_t size,size_t count,FILE*fp）；
```

功能：从fp文件指针所指向的文件中读取一个数据块。

返回值：buffer是所要读入内存中的地址，size是所要读取的数据块的字节数，而count是所要读取的数据块的个数。

下面看上述两个函数的使用。

```
#include＜stdio.h＞
#define N 4
struct stu
{
char name[10];
char nu[10];
int score;
}st[N]={{"李小鹏", "21009011", 326},
{"张迪蝶", "21009012", 562},
{"彭晓敏", "21009013", 456},
{"王梅梅", "21009014", 258}
};
int main（）
{
FILE*fpin;
fpin=fopen（"bina.txt", "w"）；
if（NULL==fpin）
{
printf（"打开失败！\n"）；
return 0;
}
```

```
int i;
for (i=0; i<N; i++)
{
fwrite (&st[i], sizeof (st[i]) , 1, fpin) ;
}
fclose (fpin) ;
return 0;
}
```

在上面的代码中，通过函数fwrite向指定的文件中输入数据，打开文件后发现使用该函数存储的数据与前面介绍的函数都不同。我们不能主观地判断存储的结果是否正确，为了验证结果的正确与否，接下来用fread函数来读取刚才存储的信息，并将其打印出来看是否成功地实现了fwrite的写入，同时也查看一下fread函数的使用情况，代码如下：

```
#include<stdio.h>
#define N 4
struct stu
{
char name[10];
char nu[10];
int score;
};
int main ()
{
FILE*fpout;
fpout=fopen ("bina", "rb") ;
if (NULL==fpout)
{
printf ("打开失败! \n") ;
return 0;
}
struct stu ss[4];
fread (ss,sizeof (ss[1]) , N,fpout) ;
int i;
for (i=0; i<N; i++)
{
printf ("%s\t%s\t%d\n", ss[i].name,ss[i].nu,ss[i].score) ;
}
fclose (fpout) ;
```

```
return 0;
}
```

运行结果：

```
李小鹏21009011 326
张迪蝶21009012 562
彭晓敏21009013 456
王梅梅21009014 258
```

从上面的运行结果可以发现，输出的就是前面的代码中采用fwrite函数写入的数据，同时通过fread函数一次性从指定的文件中读取出了存储的数据信息。读取的信息存储在ss数组中，接下来通过一个for循环对其进行打印输出。

8.4 位置指针对文件的定位

在讲解位置指针之前，先要明白的是位置指针和前面所讲的文件指针并不相同。文件指针指向所要操作的文件，而位置指针指向读写文件内部存储数据的地址，通过移动位置指针可以在文件的指定位置上进行数据的读写操作。

对位置指针的操作通常用以下3个函数来实现。

1.rewind函数

```
void rewind（FILE*fp）；
```

功能：将位置指针重新定位到文件的开头。

返回值：无。

下面通过一段代码来看看该函数的使用，其功能为对打开的文件进行两次备份操作。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin，*fpout1，*fpout2；
fpin=fopen（"text1.txt"，"r"）；
fpout1=fopen（"text2.txt"，"w"）；
fpout2=fopen（"text3.txt"，"w"）；
if（NULL==fpin||NULL==fpout1||NULL==fpout2）
{
```

```
printf（"打开失败！\n"）；
return 0;
}
while（！feof（fpin））
fputc（fgetc（fpin），fpout1）；
rewind（fpin）；
while（！feof（fpin））
fputc（fgetc（fpin），fpout2）；
fclose（fpin）；
fclose（fpout1）；
fclose（fpout2）；
return 0;
}
```

在上面的代码中，先打开text1.txt，将其中的内容读取出来并写入到text2.txt中，这时文件中的位置指针已经不再指向文件中数据的起始地址，所以再次读取text1.txt之前，要先采用rewind函数将其位置指针重新定位到文件的起始地址，再次将其中的数据备份到text2.txt中。

2.fseek函数

```
int fseek（FILE*fp,long offset,int fromwhere）；
```

功能：按照需要任意移动位置指针，其中，参数fp是所要操作的文件指针，参数offset是以fromwhere参数为起始位置的偏移量，而参数fromwhere的取值如表8-2所示。

表 8-2　fromwhere 参数的取值

取　值	相应的整数值	含　义
SEEK_SET	0	文件开头
SEEK_CUR	1	位置指针的当前位置
SEEK_END	2	文件末尾

所以，在使用该函数进行位置指针定位时需要注意位置指针的起始地址。

返回值：成功则返回0，否则返回非0值。

接下来通过代码了解该函数的使用。以下代码的功能为先通过fwrite函数将信息写入指定的文件中，然后通过fseek函数来进行位置指针的定位，逐一取出其中的数据信息存放到指定的数组中。

```
#include＜stdio.h＞
#define N 4
struct stu
{
char name[10];
char nu[10];
int score;
}st[N]={{"李小鹏", "21009011", 326},
{"张迪蝶", "21009012", 562},
{"彭晓敏", "21009013", 456},
{"王梅梅", "21009014", 258}
};
int main ()
{
FILE*fpin, *fpout;
fpin=fopen ("bina.txt", "w") ;
if (NULL==fpin)
{
printf ("打开失败! \n") ;
return 0;
}
fwrite (＆st,sizeof (struct stu) , 4, fpin) ;
fclose (fpin) ;
fpout=fopen ("bina.txt", "r") ;
if (NULL==fpout)
{
printf ("打开失败! \n") ;
return 0;
}
struct stu sd[N];
```

```
int i;
for (i=0; i<N; i++)
{
fseek (fpout, -sizeof (sd[i]) * (i+1), 2);
fread (&sd[i], sizeof (sd[i]), 1, fpout);
printf ("%s\t%s\t%d\n", sd[i].name,sd[i].nu,sd[i].score);
}
fclose (fpout);
return 0;
}
```

运行结果：

```
王梅梅21009014 258
彭晓敏21009013 456
张迪蝶21009012 562
李小鹏21009011 326
```

在上面的代码中，在使用fseek函数的时候将fromwhere起始地址设置为2，所以位置指针的相对位置是相对于文件的末端开始的，fseek函数中的第二个参数是一个负值，这样就将文件中的数据一一存放到指定的数组中，并且进行打印输出。

3.ftell函数

```
long ftell (FILE*fp);
```

功能：获取位置指针当前位置相对于文件首的偏移字节数。

返回值：调用成功则返回当前文件的读写位置，失败则返回-1。

接下来通过一个简单的代码来了解该函数的使用。以下代码的功能

为通过ftell函数获取每个学生信息存储地址相对于文件首的偏移字节数。

```
#include＜stdio.h＞
#define N 4
struct stu
{
char name[10];
char nu[10];
int score;
}st[N]={{"李小鹏", "21009011", 326},
{"张迪蝶", "21009012", 562},
{"彭晓敏", "21009013", 456},
{"王梅梅", "21009014", 258}
};
int main ()
{
FILE*fpin, *fpout;
fpin=fopen ("bina.txt", "w") ;
if (NULL==fpin)
{
printf ("打开失败! \n") ;
return 0;
}
fwrite (＆st,sizeof (struct stu) , 4, fpin) ;
fclose (fpin) ;
fpout=fopen ("bina.txt", "r") ;
if (NULL==fpout)
{
printf ("打开失败! \n") ;
return 0;
}
struct stu sd[N];
int i;
for (i=0; i＜N; i++)
{
printf ("%d\t", ftell (fpout) ) ;
fread (＆sd[i], sizeof (sd[i]) , 1, fpout) ;
}
return 0;
}
```

运行结果：

```
0 24 48 72
```

从上面的运行结果可以看出，每个学生信息在内存中占用的字节大小为24字节。结合上面讲解的fseek函数，可以取文件的长度，代码如下：

```
#include＜stdio.h＞
#define N 4
struct stu
{
char name[10];
char nu[10];
int score;
}st[N]={{"李小鹏", "21009011", 326},
{"张迪蝶", "21009012", 562},
{"彭晓敏", "21009013", 456},
{"王梅梅", "21009014", 258}
};
int main（）
{
FILE*fpin, *fpout;
fpin=fopen（"bina.txt", "w"）;
if（NULL==fpin）
{
printf（"打开失败！\n"）;
return 0;
}
fwrite（&st,sizeof（struct stu）, 4, fpin）;
fclose（fpin）;
fpout=fopen（"bina.txt", "r"）;
if（NULL==fpout）
{
printf（"打开失败！\n"）;
return 0;
}
fseek（fpout, 0, 2）;
printf（"bian.txt文件占用的内存大小为%d字节\n", ftell（fpout））;
return 0;
```

```
}
```

运行结果：

```
bian.txt文件占用的内存大小为96字节
```

从上面的运行结果可以看出，bian.txt文件占用96字节。由上面的代码可知，每个学生信息在内存中占用24字节的大小，总共有4个学生，即96字节，所以不仅可以通过fseek函数来随机指定位置指针，还可以通过fseek函数和ftell函数来获取文件的长度。

8.5 文件中的出错检测

文件操作中的错误在所难免，那么一般都有哪些错误检测方法呢？接下来就介绍一下文件操作中常见的错误检测方法。

1.ferror函数

```
int ferror（FILE*fp）；
```

功能：检测对文件指针fp所指向的文件读写操作出现的错误。

返回值：没有出错就返回0，出错就返回非0值。

在使用该函数进行错误检测的时候需要注意，由于每次进行读写操作后，再调用ferror函数都会产生一个新的值，因此在调用读写操作函数后要及时地调用ferror函数对其进行检测，否则信息就会丢失。看看下面的代码。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin，*fpout;
fpin=fopen（"bina.txt"，"r"）；
if（ferror（fpin））
{
printf（"打开失败！\n"）；
return 0;
}
fputc（'a'，fpin）；
if（ferror（fpin））
{
```

```
printf（"写入失败！\n"）；
fclose（fpin）；
return 0;
}
fclose（fpin）；
return 0;
}
```

在上面的代码中，对文件bian.txt采用只读方式打开，接下来却通过fputc函数对其进行写入操作，导致出错，所以运行程序后会出现如下信息：

```
写入失败！
```

2.clearerr函数

```
void clearerr（FILE*fp）；
```

功能：复位错误标志。

返回值：无返回值。

如果调用上面的ferror函数进行检测并发现错误，ferror函数就返回一个非0值，那么该值并不会被重新置为0，但是可以人为地将其错误标志清除掉，使其变为0，这需要通过clearerr函数来实现。看看下面的代码。

```
#include＜stdio.h＞
int main（）
{
```

```
FILE*fpin, *fpout;
fpin=fopen ("bina.txt", "r");
if (ferror (fpin) )
{
printf ("打开失败! \n");
return 0;
}
fputc ('a', fpin);
clearerr (fpin);
if (ferror (fpin) )
{
printf ("写入失败! \n");
fclose (fpin);
return 0;
}
printf ("清除成功! \n");
fclose (fpin);
return 0;
}
```

运行结果：

```
清除成功!
```

在上面的代码中，在调用写入函数出现错误后，通过clearerr函数对其进行清零处理，接下来再次调用该函数的时候，就不会显示“写入失败”的信息了，而会打印接下来的“清除成功！”信息。

3.perror函数

```
void perror (const char*s);
```

功能：将上一个函数发生错误的原因输出到标准设备（stderr）。参数s所指的字符串先被打印，后面再加上错误原因字符串。此错误原

因依照全局变量error的值来决定要输出的字符串。

返回值：无返回值。

```
#include＜stdio.h＞
int main（）
{
FILE*fpin，*fpout;
fpin=fopen（"fdsa.txt"，"r"）;
if（NULL==fpin）
{
perror（"E：\\fdsa\\f.txt"）;
return 0;
}
fclose（fpin）;
return 0;
}
```

由于在当前目录下并没有fdsa.txt这个文件，因此在运行上面的代码时打印输出了如下信息：

```
E：\fdsa\f.txt：No such file or directory
```

打印输出的信息中给出了出错的原因。

4.strerror函数

```
char*strerror（int errnum）;
```

功能：通过参数errnum返回对应的错误信息。

返回值：errnum所对应的描述信息。

该函数可以把出错的数值信息转化为易于理解的字符串信息，而不再是那些虚无缥缈的数字，便于编程人员对出错的数值信息进行详细分析。通过下面的代码来看看该函数的使用方法。

```
#include＜stdio.h＞
#include＜errno.h＞#
#include＜string.h＞
int main（）
{
FILE*fpin，*fpout;
fpin=fopen（"fdsa.txt"，"r"）；
if（NULL==fpin）
{
printf（"%s\n"，strerror（errno））；
return 0;
}
fclose（fpin）；
return 0;
}
```

运行结果：

```
出错信息为：No such file or directory
```

由上面的运行结果可知，这次同样得到了出错的原因：由于errno是在errno.h的头文件中声明的，因此需要引入errno.h头文件，而strerror函数是在string.h头文件中声明的，所以在此也要引入string.h头文件。

第9章　调试和异常处理

在编写代码时难免会犯语法和逻辑上的错误，如何有效地进行错误的定位，是调试的一大技巧，而异常处理更多出现在Java、C#等面向对象编程中，在C语言中可能很少被提及，这并不代表C语言不具备异常处理能力。本章就来介绍如何在C语言中实现异常处理和在代码中增加供调试语句的方法，以便在测试中快速进行错误定位。

9.1 assert宏的使用及注意事项

在讲解assert宏之前，先对断言进行基本介绍，让读者对断言有一个大致了解。在使用C语言编程时，我们总会对某种假设条件进行检查，断言就用于在代码中捕捉这些假设，可以将断言看做异常处理的一种高级形式。断言表示为一些布尔表达式，程序员相信在程序中的某个特定点处的表达式值为真。可以在任何时候启用和禁用断言验证，比如可以在测试时启用断言，而在部署时禁用断言。同样，在程序投入运行后，最终用户在遇到问题时可以重新启用断言。通过断言可以快速发现并定位软件的问题，同时对系统错误进行自动报警。对于在系统中隐藏很深，用其他手段极难发现的问题可以用断言来定位，从而缩短问题定位时间，提高系统的可测性。

接下来介绍C语言系统所提供的用于实现断言的宏assert，其一般形式为：

```
assert（expression）；
```

参数expression可以是一般的常量、表达式、函数等。在使用assert宏的时候，先计算expression，如果expression的值为假（即为0），那么它先向stderr打印一条出错信息，然后通过调用abort（）函数来终止程序运行。看看下面的代码。

```
#include＜stdio.h＞
#include＜assert.h＞
int main（void）
{
int i;
i=1;
assert（i++）;
printf（"通过assert宏进行i++运算之后的i值为：%d\n"，i）;
return 0;
}
```

运行结果：

```
通过assert宏进行i++运算之后的i值为：2
```

分析运行结果，因为给定的i初始值为1，所以使用“assert（i++）；”语句时不会出现错误，进而执行了i++，其后的打印语句输出值为2。如果把i的初始值改为0，那么就会出现如下错误。

```
Assertion failed：i++，file E：\fdsa\fdsag.cpp,line 8
```

根据提示，是不是很快就能定位出错点呢？既然assert这么便于定位出错点，看来的确有必要在代码中熟练使用它，但是任何东西的使用都是有规则的，assert也不例外。

断言语句不需要总被执行，可以屏蔽，也可以启用，这就要求assert不管是在屏蔽状态还是启用状态都不能对代码本身的功能有所影响，因此之前在代码中使用了一句“assert（i++）；”是不妥的，一旦禁用了assert,i++的语句就得不到执行，接下来对i值的使用就会出现问

题。对于这样的语句应该分开实现，可以用如下两句来替代。

```
assert（i）；
i++;
```

这就对断言的使用有了相应的要求，那么，一般在什么情况下使用断言呢？

可以在正常情况下程序不会到达的地方放置断言。

使用断言测试方法执行的前置条件和后置条件。

使用断言检查程序中变量的状态，确保变量的状态必须满足。

对于上面提到的前置条件和后置条件，有的读者可能还不是很了解，现解释如下：

前置条件断言：代码执行之前必须具备的特性。

后置条件断言：代码执行之后必须具备的特性。

前后不变断言：代码执行前后不能变化的特性。

当然在使用断言的过程中会有一些应该注意的事项和需要养成的一些良好习惯，如：

每个assert只检验一个条件，如果同时检验多个条件且结果断言失

败，那就无法直观地判断哪个条件导致失败。

不能使用改变环境的语句，比如上面的代码改变了i变量，在实际编写代码的过程中不能这样做。

assert和后面的语句之间应空一行，以形成逻辑和视觉上的一致感，使编写的代码有一种视觉上的美感。

有时，assert不能代替条件过滤。

放在函数参数的入口处检查传入参数的合法性。

断言语句不可以有任何边界效应。

上面那么多文字，似乎很枯燥，但是要先坚持看完文字描述部分，才能在分析代码的过程中很快知道为什么会出现那样的问题，也才能在自己编写代码的时候熟练使用assert，为代码调试带来极大的便利。尤其是在利用C语言开发工程项目的时候，如果能够在代码中合理地使用assert，那么创建出的代码会更稳定，质量更好且不易于出错。但凡优秀的程序员都能够很好地使用assert编写出高质量的代码来。

介绍了assert这么多的优点，当然也要说说它的缺点。使用assert的缺点是，频繁调用会极大地影响程序的性能，增加额外开销。因此在调试结束后，可以通过在包含“#include”的语句之前插入#define NDEBUG来禁用assert调用。接下来看下面的一段代码。

```
#include＜stdio.h＞
//#define NDEBUG
#include＜assert.h＞
int copy_string（char from[], char to[]）
{
int i=0;
while（to[i++]=from[i]）;
return 1;
}
int main（）
{
int ret;
char str[]="this is a string! ";
char dec_str[50];
printf（"main函数中的str字符串为：%s\n"，str）;
ret=copy_string（str,dec_str）;
assert（ret）;
printf（"复杂成功后的dec_str字符串为：%s\n"，dec_str）;
return 0;
}
```

运行结果：

```
main函数中的str字符串为：this is a string!
复杂成功后的dec_str字符串为：this is a string!
```

在以上代码的开头部分，“#define NDEBUG”被注释掉了，所以启用了assert,main函数中使用的“assert（ret）”，而不是“assert（copy_string（str,dec_str））”，这样就避免了由于关闭测试而带来的函数copy_string（str,dec_str）不会被执行的情况。在调试的时候，只需要检测函数的返回值是否满足条件就可以了，而不用将函数作为assert的参数。因此在调试通过的情况下，关闭断言后不会对代码造成任何的影响，最终成功打印了两条输出语句。

需要注意的是，关闭断言的时候，“#define NDEBUG”必须写在“#include＜assert.h＞”的前面，否则将不能关闭断言。看看下面的代码。

```
#include＜stdio.h＞
#include＜assert.h＞
#define NDEBUG
int main（）
{
assert（0）；
printf（"成功关闭断言\n"）；
return 0;
}
```

在上面的代码中，按照前面的方式关闭断言后运行结果是“Assertion failed：0，file E：\fdsa\fdsag.cpp,line 7”，并且程序运行失败，这是因为并没有成功关闭断言，程序还是按照启用assert的方式来进行编译，同时调用abort（）函数终止了程序的运行。所以在关闭断言的时候，需要将“#define NDEBUG”放在加载assert.h头文件语句的前面。

9.2 如何设计一种灵活的断言

在实际应用时，可能需要根据具体情况灵活地设计断言，所以我们对断言的认识就不能仅停留在使用上面，应该学会设计自己的断言代码，以便在代码中灵活地使用断言。接下来就介绍如何自己在代码中编写断言，先来看下面断言的实现代码。

```
#include<stdio.h>
#include<stdlib.h>
//#define NDEBUG
#ifndef NDEBUG
#define assert(p) if(!(p)) \
{\
fprintf(stderr, "Assertion failed: %s,file%s,
line%d\n", #p, __FILE__, __LINE__); \
abort(); \
}
#else
#define assert(p) (void)(0)
#endif
int sum(int a[], int n)
{
assert(n>0);
int i;
int sum=0;
for(i=0; i<n; i++)
{
sum+=a[i];
}
return sum;
}
int main()
{
int i=0;
int total;
int arr[]={2, 3, 4, 7};
total=sum(arr, 4);
printf("数组中的元素之和为: %d\n", total);
total=sum(arr, 0);
```

```
return 0;
}
```

运行结果：

```
数组中的元素之和为：16
Assertion failed: n＞0，file E: \fdsa\fdsag.cpp,line 16
```

以上代码中断言的实现方法是采用条件编译指令，这是为了在程序中可以开启或者关闭断言。下面来看开启断言的实现方法。

```
#define assert（p）if（！（p））\
{\
fprintf（stderr，"Assertion failed: %s,file%s,line%d\n"，#p，__
FILE__，__LINE__）；\
abort（）；\
}
```

以上代码采用fprintf函数实现打印输出，其中，输出设备是stderr，同时在参数中使用了宏__FILE__和__LINE__，用于输出当前出错的文件名及行号。参数中还有一个“#p”，关于形参的使用在第2章“预处理”中已经进行了相应的讲解，读者可以查阅相关内容。再来看关闭断言的实现部分，这里使用的方法之前也介绍过，如果删掉代码中的“||”，关闭断言，那么宏assert（p）可用（void）（0）来替代，这时宏在代码中就像被清除了一样，不会有任何的作用。再来看以上代码中对宏的使用，通过宏来检测函数参数传递是否满足要求。

看上面采用VC++6.0编译实现断言的方法，输出信息中只包含文件

名和行号，如果在Linux环境下采用gcc编译运行，那么可以得到更加具体的信息。例如，可以通过宏__func__获取出错位置的函数名，实现方法如下：

```
    #include＜stdio.h＞
    #include＜stdlib.h＞
    //#define NDEBUG//禁用
    #ifndef NDEBUG//启用断言测试
    void assert_report（const char*file_name,const
char*function_name,unsigned int line_no）
    {
    printf（"file_name：%s,function_name：%s,line%u\n",
    file_name,function_name,line_no）；
    abort（）；
    }
    #define assert（condition）\
    do{\
    if（condition）\
    NULL；\
    else\
    {\
    printf（"Error Report：%s"，#condition）；\
    assert_report（__FILE__，__func__，__LINE__）；\
    }\
    }while（0）
    #else
    #define assert（condition）NULL
    #endif
    int main（void）
    {
    int i；
    i=0；
    assert（i++）；
    printf（"%d\n"，i）；
    return 0；
    }
```

运行结果：

```
    root@ubuntu:/home#gcc assert.c-o assert
```

```
root@ubuntu:/home#./assert
Error Report: i++file_name: assert.c,function_name: main,line 33
Aborted
```

分析上面的运行结果，得到的出错信息包括文件名、函数名、行号，有了这些信息可以对出错位置进行快速定位。在断言的实现中使用了abort（）函数，如果条件为假，那么就打印出错信息，同时运行abort函数终止程序的运行。

仔细阅读代码，会发现在宏定义中使用了一个do{}while（0）。使用它有什么好处呢？或许从以上代码中看不出来，那么看下面的代码就知道了。

```
#include＜stdio.h＞
void print_1（void）
{
printf（"成功地调用print_1函数\n"）；
}
void print_2（void）
{
printf（"成功地调用print_2函数\n"）；
}
#define printf_value（）\
print_1（）；\
print_2（）；
int main（void）
{
int i=0;
if（i==1）
printf_value（）；
return 0;
}
```

运行结果：

```
成功地调用print_2函数
```

看了上面的运行结果，可能有的读者会疑惑为什么会出现这样的错误。如果if语句的条件不满足，那么print_value（）函数应该不会被调用，怎么会出现打印结果呢？如果把上面的“printf_value（）”替换为“print_1（）；print_2（）；”，就会很清楚地发现if语句在此处的作用仅仅是限制不调用“print_1（）；”，而“print_2（）；”在控制之外，所以出现了上面的结果。有的读者可能会马上想到加上括号{}就好了，这里加上括号{}的确就可以了，因为这里是一种特殊情况，即没有else语句。如果在宏定义中使用{}并加入else语句，结果会如何呢？看下面一段代码。

```
#include＜stdio.h＞
void print_1（void）
{
printf（"成功地调用print_1函数\n"）；
return;
}
void print_2（void）
{
printf（"成功地调用print_2函数\n"）；
return;
}
#define printf_value（）\
{\
print_1（）；\
print_2（）；}
int main（void）
{
int i=0;
if（i==1）
printf_value（）；
else
printf（"add else word! "）；
return 0;
```

```
}
```

运行结果：

```
error C2181: illegal else without matching if
```

看似正确的代码，编译的时候就会出现错误。为什么会出现这样的错误呢？因为在编写C语言代码时，在每个语句后面加分号是一种约定俗成的习惯，以上代码中的printf_value（）语句后面有一个分号，这个分号的作用使else没有了与之相对应的if，所以编译时会出错。如果使用do while（0）就不会出现这种问题，因此在编写代码时应该学会在宏定义中使用do{}while（0）。

9.3 如何实现异常处理

很多初学者在此之前可能根本没有使用或听说过C语言的异常处理，印象中只有C++和Java等编程语言才有这些内容，C语言怎么会有异常处理呢？当然，对于一般性的学习或应付考试等，几乎不会提及C语言的异常处理。那么，到底什么是异常处理，C语言又是如何实现异常处理的呢？接下来就讲解C语言异常处理的一种典型的实现方法：以setjmp函数和longjmp函数实现异常处理。

首先来了解什么是异常处理。异常是一个在程序执行期间发生的事件，它中断正在执行的程序的正常指令流，而异常处理提供了处理程序运行时出现的任何意外或异常情况的方法。

下面先看setjmp函数和longjmp函数。

setjmp函数的原型如下：

```
int (jmp_buf env);
```

打开源代码会发现在setjmp函数中涉及很多寄存器的操作，如EBP、EBX、EDI、ESI、ESP、EIP等，在此就不一一例举了，其作用是在调用setjmp函数的过程中保存程序当前运行时的堆栈环境。保存这些堆栈环境有什么用呢？先来看longjmp函数。

longjmp函数的原型如下：

```
void longjmp（jmp_buf env,int value）；
```

setjmp函数的功能是保存程序执行时候的堆栈环境。在longjmp（）函数中也有一个jmp_buf类型的env变量，这是为了保证接下来调用longjmp时根据这个曾经保存的变量来恢复先前的环境，并且当前的程序控制流会因此返回到最初调用setjmp函数时的程序执行点。此时，在接下来的控制流的历程中，所能访问的所有变量都包含了调用longjmp函数时所拥有的变量。仅通过文字讲解可能有点抽象，还是通过一段代码来进行分析。

```
#include＜stdio.h＞
#include＜setjmp.h＞
int main（）
{
double a,b;
printf（"请输入被除数："）；
scanf（"%lf"，&a）；
printf（"请输入除数："）；
if（setjmp（buf）==0）
{
scanf（"%lf"，&b）；
if（0==b）
longjmp（buf，1）；
printf（"相除的结果为：%f\n"，a/b）；
}
else
printf（"出现错误除数为0\n"）；
return 0;
}
```

运行结果：

```
请输入被除数：23
请输入除数：0
出现错误除数为0
```

接着上面的内容继续讲解，在一开始我们并没有具体交代setjmp函数和longjmp函数的返回值和参数的具体含义。两个函数中的env变量保存的是调用setjmp函数时当前运行程序的堆栈信息，而longjmp函数的调用就是根据调用setjmp函数时的堆栈信息返回到最初调用setjmp函数的地方，其中的第二个参数就是此刻setjmp函数的返回值。值得注意的是，在调用longjmp函数之后，setjmp函数返回的值必须是非0值，如果longjmp传送的value参数值为0，那么setjmp的返回值实际上是1。一开始调用setjmp函数的时候，它的返回值为0，之后再调用longjmp函数的时候，通过设置longjmp函数的第二个参数来设定它的返回值。

现在分析上边的代码，在main函数中，最初调用setjmp函数时把当前的环境信息保存在buf中，函数返回0，然后向下运行，输入0。由if语句可知，b的值为0时调用longjmp函数，具体方式为“longjmp（buf，1）；”。通过上面的讲解，我们知道第一个参数的作用是得到最初调用setjmp函数时的环境信息，以便在使用longjmp函数的时候能够正确返回到setjmp函数最初的调用处，而后面的参数表示的是返回到setjmp函数时的返回值。在此返回1，所以执行else部分的语句。

分析完上面的代码，读者应该知道这两个进行异常处理的函数的使用方法。值得注意的是，在函数setjmp与longjmp结合使用时，必须遵循

严格的先后执行顺序，先调用setjmp函数，再调用longjmp函数，以恢复到先前被保存的"程序执行点"。否则，如果在调用setjmp之前执行longjmp函数，将导致程序的执行流变得不确定，可能导致程序崩溃，进而退出执行。

上面实现的异常处理并没有体现出setjmp函数和longjmp函数异常处理的强大之处，这种本地跳转也可以采用前面讲解的goto语句来实现。接下来通过setjmp函数和longjmp函数来实现非本地跳转，看下面的代码。

```
#include＜stdio.h＞
#include＜setjmp.h＞
jmp_buf buf;
void exception（void）
{
longjmp（buf，1）；
}
int main（）
{
double a,b;
printf（"请输入被除数："）；
scanf（"%lf"，&a）；
printf（"请输入除数："）；
if（setjmp（buf）==0）
{
scanf（"%lf"，&b）；
if（0==b）
exception（）；
printf（"相除的结果为：%f\n"，a/b）；
}
else
printf（"出现错误除数为0\n"）；
return 0;
}
```

运行结果：

```
请输入被除数：12
请输入除数：0
出现错误除数为0
```

这里对前面的代码做了适当的修改，修改后代码的功能为通过setjmp函数和longjmp函数实现非本地跳转，再来看下面的一段代码。

```
#include＜stdio.h＞
#include＜setjmp.h＞
#include＜stdlib.h＞
#include＜string.h＞
jmp_buf buf;
int sum（int a[]，int n）
{
int i,total;
if（n＜=0）
longjmp（buf，1）；
total=0;
for（i=0；i＜n；i++）
total+=a[i];
return total;
}
float average（int score[]，int n）
{
int total,i;
float avg;
total=0;
for（i=0；i＜n；i++）
{
if（score[i]＜0）
longjmp（buf，2）；
total+=score[i];
}
avg=total/n;
return avg;
}
int main（）
{
int value,total;
```

```
char str[50];
int a[3]={1，2，3}，score[3]={21，-56，25};
float avg;
value=setjmp（buf）；
if（value==0）
{
total=sum（a，3）；
avg=average（score，3）；
}
switch（value）
{
case 1:
strcpy（str，"error：传递给sum（）函数的参数n为负"）；
break;
case 2:
strcpy（str，"error：传递给average（）函数的学生的成绩为负数"）；
break;
default:
strcpy（str，"error：其他错误值"）；
break;
}
printf（"%s返回值为：%d\n"，str,value）；
return 0;
}
```

运行结果：

```
error：传递给average（）函数的学生的成绩为负数 返回值为：2
```

上面的代码用setjmp函数和longjmp函数对多个函数进行了异常处理，如果在调用函数时出现了错误，那么就返回不同的数值，根据返回的数值打印输出不同的信息。

为了能够对异常处理有更加深入的认识，接下来再看看signal函数的使用。

头文件：#include＜signal.h＞。

功能：设置某一信号的对应动作。

函数原型：

```
void (*signal (int signum,void (*handler) (int) ) ) (int);
```

注意　第一个参数signum指明了所要处理的信号类型，它可以取除了SIGKILL和SIGSTOP外的任何一种信号。

如果读者是第一次接触上面的函数，可能有些不知道如何着手，一时间难以理解这个函数到底有什么作用。在讲解函数signal之前先来看它的另外一种表示方法。

```
typedef void (*sig_t) (int);
sig_t signal (int signum,sig_t handler);
```

将上面的函数原型拆分为如上两行代码。第一行代码定义了一个函数指针，其含有一个int型参数，无返回值；在第二行代码中，signal函数的返回值是一个函数指针，与第一行定义的类型相同，第二个参数也是一个函数指针，signal的返回值就是第二个函数指针指向的函数地址。接下来就用一段代码来模拟该函数的实现方法。

```
#include<stdio.h>
#include<stdlib.h>
typedef void (*pfun) ();
pfun signal_call (int a,pfun fun);
```

```
pfun signal_call（int a,pfun fun）
{
return fun;
}
void func（）
{
printf（"hello world! \n"）；
}
int main（）
{
pfun p=func;
signal_call（1，p）（）；
return 0;
}
```

运行结果：

```
hello world!
```

现在分析上面的代码，先采用前面的定义形式实现如下两行代码：

```
typedef void（*pfun）（）；
pfun signal_call（int a,pfun fun）；
```

在接下来的main函数中定义了一个函数指针p，使其指向func函数，接下来通过一句代码“signal_call（1，p）（）；”实现func函数调用，这到底是怎么实现的呢？这里进行一下分析，前面的signal_call（1，p）返回的是一个函数指针，其实在代码中返回的就是p，所以“signal_call（1，p）（）；”就可以变形为p（）。看到这种形式，我们就可以很清楚地看出，调用的就是代码中的func函数了。到这里，读者就明白了signal函数的实现方法。接下来再看一段MSCN中的

例程——使用signal捕捉除数为0时的异常代码，我们对其进行了适当的修改。

```
#include＜stdio.h＞
#include＜signal.h＞
#include＜setjmp.h＞
#include＜stdlib.h＞
#include＜float.h＞
#include＜string.h＞
jmp_buf buf;
int err;
void handler（int num）
{
err=num;
printf（"发生浮点计算异常\n"）；
longjmp（buf，1）；
}
int main（void）
{
double a,b;
char str[20];
int ret;
_control87（0，_MCW_EM）；
if（signal（SIGFPE,handler）==SIG_ERR）
{
printf（"绑定失败\n"）；
abort（）；
}
ret=setjmp（buf）；
if（0==ret）
{
printf（"请输入被除数："）；
scanf（"%lf"，&a）；
printf（"请输入除数："）；
scanf（"%lf"，&b）；
printf（"a/b=%4.3g\n"，a/b）；
printf（"发生异常时候不会被执行的语句\n"）；
}
return 0;
}
```

没有发生浮点异常的运行结果为：

```
请输入被除数：123
请输入除数：3
a/b=41
发生异常时候不会被执行的语句
```

发生浮点异常的运行结果为：

```
请输入被除数：45
请输入除数：0
发生浮点计算异常
```

现在来分析上面的运行结果，先看“_control87（0，_MCW_EM）；”这句代码，很多读者可能对这句代码比较陌生，它的功能是开启所有的浮点计算异常。在通常情况下，浮点计算异常是被屏蔽掉的，为了能够使接下来的signal能够捕捉到浮点计算异常，要将其开启。接下来通过signal（SIGFPE,handler）来绑定一个浮点计算异常处理函数，当发生异常时，就调用handler（）函数来处理。紧接着通过“ret=setjmp（buf）；”保存程序运行的环境信息，以便接下来调用longjmp函数时能够根据保存的信息返回该程序先前setjmp函数的执行点。对比两次的运行结果可以发现，如果发现异常，接下来的打印语句"printf（"发生异常时候不会被执行的语句\n"）；"是不会被执行的，而且直接跳转到绑定的handler函数开始执行。当然，在此仅列举一些简单的代码教读者学会使用setjmp函数和longjmp函数来实现异常处理，读者完全可以在此基础上编写出复杂的异常处理。

9.4 如何处理段错误

所谓段错误，就是指访问了那些不可用或受保护的内存所导致的错误。那么该如何查找发生段错误的原因呢？其实，前面讲解函数间的调用关系时已经讲到了一种查看的方法，接下来介绍如何通过寄存器来查看段错误的原因。

通过寄存器来查看段错误的思路是在代码中捕捉SIGSEGV信号。该信号是当一个进程执行了一个无效的内存引用或发生段错误时发送给程序的信号。SIGSEGV的符号常量在头文件signal.h中定义。

由于在此不是对捕捉信号进行简单地处理，而是要从中得到一些寄存器方面的信息，因此不采用signal函数来捕捉，而是采用另外一个函数sigaction来捕捉。下面看一下捕捉函数sigaction的定义形式。

```
   int sigaction（int signum,const struct sigaction*act,struct
sigaction*oldact）；
```

在讲解sigaction函数的参数的含义之前，先来看在Linux环境下参数struct sigaction的实现。在sigaction.h的头文件中实现，所在路径为：usr/include/bits。

```
   struct sigaction
   {
   #ifdef __USE_POSIX199309
   union
```

```
{
__sighandler_t sa_handler;
void (*sa_sigaction) (int,siginfo_t*, void*);
}
__sigaction_handler;
#define sa_handler__sigaction_handler.sa_handler
#define sa_sigaction__sigaction_handler.sa_sigaction
#else
__sighandler_t sa_handler;
#endif
__sigset_t sa_mask;
int sa_flags;
void (*sa_restorer) (void);
};
```

在上面的实现代码中，使用了一条条件编译语句来进行选择性编译。如果系统定义了__USE_POSIX199309，那么就编译#ifdef部分的代码，否则编译#else部分的代码。要注意下面两个宏的实现：

```
#define sa_handler__sigaction_handler.sa_handler
#define sa_sigaction__sigaction_handler.sa_sigaction
```

上面的两个宏简化了对结构体中嵌入的共用体内成员变量的引用，其中的共用体数据结构中的两个元素_sa_handler和*_sa_sigaction用于指定信号关联函数，即用户指定的信号处理函数。信号处理函数除了可以是用户自定义的处理函数外，还可以为SIG_DFL（默认的处理方式），也可以为SIG_IGN（忽略信号）。由_sa_handler指定的处理函数只有一个参数，即信号值，因此不能传递除信号值之外的任何信息。由_sa_sigaction指定的信号处理函数带有3个参数，这个函数是为实时信号而设的（当然同样支持非实时信号）。第一个参数为信号值；第三个参

数传递的是一些发生异常时的寄存器信息，在此也是根据这些寄存器所携带的信息进行异常的定位；第二个参数是指向siginfo_t结构的指针，结构中包含信号携带的数据值，参数所指向的结构如下。

```
typedef struct siginfo
{
int si_signo;
int si_errno;
int si_code;
union
{
int_pad[__SI_PAD_SIZE];
struct
{
__pid_t si_pid; __uid_t si_uid;
}_kill;
struct
{
int si_tid; int si_overrun;
sigval_t si_sigval;
}_timer;
struct
{
__pid_t si_pid; __uid_t si_uid; sigval_t si_sigval;
}_rt;
struct
{
__pid_t si_pid; __uid_t si_uid;
int si_status; __clock_t si_utime;
__clock_t si_stime;
}_sigchld;
struct
{
void*si_addr;
}_sigfault;
struct
{
long int si_band;
int si_fd;
}_sigpoll;
}_sifields;
}siginfo_t;
```

分析上面的实现，重点看前三个结构体成员变量，si_signo表示信号值，si_errno是error值，si_code表示信号产生的原因，三个值都对所有信号有意义。其他的结构体成员变量在此并不涉及，所以就不一一讲解了。

接下来分析sigaction结构体成员的含义。

sa_mask，指定在信号处理程序执行过程中哪些信号应当被阻塞。除非指定为SA_NODEFER或者SA_NOMASK标志位，否则默认当前信号本身被阻塞，以防止信号的嵌套发送。

sa_restorer，此参数没有使用。

sa_flags，控制内核对该信号的处理标记，在此的取值为SA_SIGINFO。sa_flags包含了许多标志位，有SA_INTERRUPT和SA_NOCLDWAIT等。当设置了SA_SIGINFO标志位，表示信号附带的参数可以被传递到信号处理函数中，其附加信息为一个指向siginfo结构体的指针和指向进程上下文标识符的指针。因此，应该为sigaction结构中的sa_sigaction指定处理函数，而不应该为sa_handler指定信号处理函数，否则，设置该标志变得毫无意义。即使为sa_sigaction指定了信号处理函数，如果不设置SA_SIGINFO，那么信号处理函数同样不能得到信号传递过来的数据，在信号处理函数中对这些信息的访问都将导致段错误。

接下来介绍sigaction函数中其他参数的含义。

signum：可以取除了SIGKILL和SIGSTOP之外的其他任何信号的编码。

act：如果值非NULL，将采用signum关联信号的新处理方式。

oldact：如果值非NULL，将存储以前对signum关联信号的处理方式。

下面通过代码来了解如何通过捕捉函数实现信号来捕捉和查找段错误。

```
    #include＜memory.h＞
    #include＜stdlib.h＞
    #include＜stdio.h＞
    #include＜signal.h＞
    static void sigsegv_handler（int signum,siginfo_t*info,void*ptr)
    {
    ucontext_t*ucontext=（ucontext_t*）ptr;
    printf（"info.si_signo=%d\n"，signum）；
    printf（"info.si_errno=%d\n"，info-＞si_errno）；
printf（"info.si_code=%d\n"，info-＞si_code）；
    printf（"发送异常的地址为：0x%3x\n"，ucontext-＞
uc_mcontext.gregs[14]）；
    exit（-1）；
    }
    void catch_sig（）
    {
    struct sigaction action;
    memset（＆action，0，sizeof（action））；
    action.sa_sigaction=sigsegv_handler;
    action.sa_flags=SA_SIGINFO;
    if（sigaction（SIGSEGV，＆action,NULL）！=0）
    {
    perror（"sigaction"）；
```

```
}
return;
}
void cause_segv ()
{
int*a;
*a=123;
return;
}
int main ()
{
catch_sig () ;
cause_segv () ;
return 0;
}
```

在分析代码之前，先来看看上面通过捕捉函数实现的处理函数中的一句代码：

```
ucontext_t*ucontext= (ucontext_t*) ptr;
```

这句代码把捕捉函数的第三个参数强制转换为ucontext_t类型的指针。接下来看ucontext_t的实现代码。

```
typedef struct ucontext
{
unsigned long int uc_flags;
struct ucontext*uc_link;
stack_t uc_stack;
mcontext_t uc_mcontext;
__sigset_t uc_sigmask;
struct_libc_fpstate__fpregs_mem;
}ucontext_t;
```

在此重点介绍其结构体中的mcontext_t类型的变量uc_mcontext。mcontext_t的实现代码如下：

```
typedef struct
{
gregset_t gregs;
fpregset_t fpregs;
unsigned long int oldmask;
unsigned long int cr2;
}
```

在代码中又有一个fpregset_t类型的fpregs变量，其实现代码如下：

```
typedef int greg_t;
#define NGREG 19
typedef greg_t gregset_t[NGREG];
```

由此可以看出，fpregs是结构体中的一个数字，其保存的是寄存器中的值，可以通过该数组中的内存进行段错误的查找。知道了其实现方法，接下来看看以上代码的运行结果。

```
root@ubuntu:/home#gcc-g seg.c-o seg
root@ubuntu:/home#./seg
info.si_signo=11
info.si_errno=0
info.si_code=2
发送异常的地址为：0x8048566
```

分析上面的运行结果，在ucontext_t结构体中有很多的寄存器值，在此选用保存发生异常地址数组元素所保存的值，其中保存的是代码运行过程中的出错位置，这样还不能直观地发现出错的具体代码，我们采用前面的addr2line工具将其转换为代码中的相对位置。

```
root@ubuntu:/home#addr2line-f-e seg 0x8048566
cause_segv
```

```
/home/seg.c：36
```

对于addr2line工具的使用，在此就不再讲解了，读者可以查看前面对函数间调用关系的讲解。通过上面定位的行号，我们再来看行号对应的代码。

```
*a=123;
```

分析第36行所对应的代码可以发现，定义的指针没有指向一个具体可用的地址，所以在此导致段错误。分析上面对段错误的查找，再根据代码的运行结果，很快就定位了出错点。因此编程的时候，我们可以采用上面的方法来查找段错误。

第10章　陷阱知识点解剖

在C语言中有不少的陷阱知识点，这对于初学者来说无疑是很头疼的一件事儿，有不少初学者可能会选择逃避的方式来避开这些陷阱知识点，虽然这样可以大大降低编程中的出错率，但是这种选择无疑会使我们对C语言的运用大打折扣，因为这些知识点在C语言中有着很高的使用频率，灵活运用这些知识点能够使我们的程序执行效率更高，更加健壮，可移植性更强。所以本章挑选那些在C语言中使用频繁，但又容易使初学者出错的知识点来进行讲解，让读者能够彻底掌握这些知识点，从而在编程的过程中运用自如。

10.1 strlen和sizeof的区别

前面讲解数据结构时已经对sizeof做了相应的讲解，并且特地强调了sizeof是操作符，而不是函数。需要注意的是，strlen是一个函数。接下来看看它们之间的区别和联系。

sizeof操作符返回的是参数所占的内存数，而strlen函数返回的是参数的字符串长度，不是所占用的内存的大小。需要注意的是，strlen函数的参数是字符串，并且必须以串结束符“\0”结尾。看看下面的代码。

```
#include<stdio.h>
#include<string.h>
int main()
{
char str[8]="fdsa";
printf("字符串的长度为：%d\n", strlen(str));
printf("占用内存的大小为：%d字节\n", sizeof(str));
return 0;
}
```

运行结果：

```
字符串的长度为：4
占用内存的大小为：8字节
```

从上面的运行结果可以发现，strlen和sizeof之间的区别在于，通过sizeof操作符得到的是定义的字符数组str占用的内存大小，而通过strlen函数得到的是对其进行初始化的字符长度。在上面的代码中定义的是数

组，如果定义的是指针，会怎么样呢？看看下面的代码。

```
#include＜stdio.h＞
#include＜string.h＞
int main（）
{
char*str="Hello";
printf（"strlen（str）=%d\n", strlen（str））；
printf（"sizeof（str）=%d\n", sizeof（str））；
printf（"sizeof（*str）=%d\n", sizeof（*str））；
return 0;
}
```

运行结果：

```
strlen（str）=5
sizeof（str）=4
sizeof（*str）=1
```

从上面的运行结果发现，str占用内存的大小变为4，这是因为指针在32位计算机中占用4字节，所以其值为4，而接下来的sizeof（*str）为1，这是因为*str表示字符串首地址的内容，在此就是字符H，占用内存大小为1字节。

10.2 const修饰符

const在C语言中算是一个比较新的描述符，我们称之为常量修饰符，就是说，其所修饰的对象为常量。如果想要设法阻止一个变量被改变，那么可以选择使用const关键字。在为一个变量加上const修饰符的同时，通常需要对它进行初始化，在之后的程序中就不能再去改变它。

可能有的读者会有疑问，在C语言中不是有预处理指令“#define VariableName VariableValue”可以很方便地进行值替代吗，为什么还要引入const修饰符呢？这是因为预处理语句虽然可以很方便地进行值的替代，但是它有个比较致命的缺点，即预处理语句只是进行简单值替代，缺乏类型检测机制，这样，预处理语句就不具备C编译器严格类型检查的优点，因此它的使用存在着一系列的隐患和局限性。

在讲解const修饰符之前，首先说明const修饰符的几个典型作用。

const类型定义：指明变量或对象的值是不能被更新的，引入目的是为了取代预编译指令。

可以保护被修饰的内容，防止其被意外地修改，增强程序的健壮性。

编译器通常不为普通const常量分配存储空间，而是将它保存在符号

表中，这使它成为一个编译期间的常量，没有了存储与读内存的操作，它的效率也很高。

可以节省空间，避免不必要的内存分配。

接下来介绍const修饰符的几种使用方式。

1.const修饰符在函数体内修饰局部变量

```
const int n=5;
```

和

```
int const n=5;
```

是等价的。在编程的过程中一定要清楚地知道const修饰的对象是谁，在这里修饰的是n，和int没有关系。const要求它所修饰的对象为常量，不能被改变，同时也不能够被赋值，所以下面这样的写法是错误的。

```
const int n;
n=0;
```

上面的情况是比较容易理解的，但是当const与指针一起使用时，就容易让人迷惑。例如，下面是关于p和q的声明：

```
const int*p;
int const*q;
```

看了上面的代码，有人可能会觉得“const int*p；”表示的是const int类型的指针（const直接修饰int），而“int const*q；”表示的是int类型的const指针（const直接修饰指针）。实际上，在上面的声明中，p和q都被声明为const int类型的指针。而声明int类型的const指针应该这样：

```
int*const r=&n;
```

以上的p和q都是指向const int类型的指针，也就是说，在以后的程序中不能改变*p的值。而r是一个const指针，在声明的时候将它初始化为指向变量n（即“r=&n；”）之后，r的值将不允许再改变，但*r的值是可以改变的。在此，为了判断const的修饰对象，介绍一种常用的方法：以*为界线，如果const位于*的左侧，那么const就是用来修饰指针所指向的变量的，即指针指向常量；如果const位于*的右侧，那么const就是修饰指针本身的，即指针本身是常量。接下来看下面的代码。

```
#include<stdio.h>
int main(int argc,char*argv[])
{
int ss=9;
int*const r=&ss;
printf("*r=%d\n", *r);
printf("ss=%d\n", ss);
*r=100;
printf("*r=%d\n", *r);
printf("ss=%d\n", ss);
return 0;
}
```

运行结果：

```
*r=9
ss=9
*r=100
ss=100
```

简单分析一下，因为r指向的是ss的地址，所以修改r指向的地址单元的值的同时，ss的值也随之变化。

结合上述两种const修饰的情况，声明一个指向const int类型的const指针，如下：

```
const int*const r=&ss;
```

这时，既不能修改*r的值，也不能修改r的值。

接下来看const用于修饰常量静态字符串的情况，例如：

```
const char*str="fdsafdsa";
```

如果没有const的修饰，我们可以在后面写“str[4]='x'”这样的语句，这样会导致对只读内存区域赋值，然后程序会立刻异常终止。有了const，这个错误就能在程序被编译的时候立即检查出来，这就是const的好处，让逻辑错误在编译期被发现。

2.const在函数声明时修饰参数

```
void*memmove（void*dest,const void*src,size_t count）;
```

这是标准库中的一个函数，在头文件#include＜string.h＞中声明，其功能为由src所指内存区域复制count个字节到dest所指向的内存区域，用于按字节方式复制字符串（内存）。它的第一个参数dest是表明将字符串复制到哪里去，即目的地，这段内存区域必须是可写的。它的第二个参数是要被复制的字符串，我们对这段内存区域仅进行只读操作，不进行写操作。于是，从这个函数自身的角度来看，src指针所指向的内存所存储的数据在整个函数执行的过程中是不变的，因此src所指向的内容是常量，于是就需要用const修饰。另外需要强调的一点就是，src和dest所指内存区域是可以重叠的，但是复制后，dest的内容会更改，函数返回指向dest的指针。看看下面的代码。

```
#include＜stdio.h＞
#include＜string.h＞
int main（int argc,char*argv[]）
{
const char*str="hello";
char buf[10];
memmove（buf,str，6）;
printf（"%s\n"，buf）;
return 0;
}
```

运行结果：

```
hello
```

如果反过来写成“memmove（str,buf，6）；”，那么编译器一定会报错。事实上，我们经常会把各种函数的参数顺序写反，此时编译器就帮了大忙。如果编译器不报错，即在函数声明“void*memmove（void*dest,const void*src,size_t count）；”处去掉const，那么这个程序在运行的时候一定会崩溃。这里还要说明的一点是，在函数参数声明中，const一般用来声明指针而不是变量本身。例如，memmove函数中的参数size_t len在memmove函数实现的时候可以完全不用更改len的值，那么是否应该把len也声明为常量呢？可以这么做。现在来分析这么做有什么优劣。如果加了const，那么对于这个函数的实现者，可以防止他在实现这个函数的时候修改不需要修改的值（len），这样很好。但是对于这个函数的使用者来说，这样做有如下两个缺点：

修饰符号毫无意义，可以传递一个常量整数或者一个非常量整数，反正对方获得的只是传递的一个副本。

如果函数内部需要更改这个值，那么这样的声明在使用中会出现错误，而在实际使用中我们不需要知道实现这个函数时是否修改过len的值。

所以，对于参数的传递，const一般只用来修饰指针。再看一个复杂的例子：

```
int execv（const char*path,char*const argv[]）;
```

着重看argv代表什么。如果去掉const，由“char*argv[]”可以看出，argv是一个数组，它的每个元素都是char*类型的指针。如果加上const，那么const修饰的是谁呢？它修饰的是一个数组，argv[]表明这个数组的元素是只读的。那么数组的元素是什么类型呢？是char*类型的指针，也就是说，指针是常量，它所指向的地址是不能改变的，而地址中的内容是可以改变的，例如：

```
argv[1]=NULL; //非法
argv[0][0]='a'; //合法
```

3.const作为全局变量

在编写程序的过程中，要尽可能少地使用全局变量。因为全局变量的作用域是全局，其值在程序范围内都可以修改，从而导致了全局变量不能保证值的正确性，如果出现错误，会非常难以发现。如果在多线程中使用全局变量，那么程序将会出现很多未知的错误。多线程中一个线程可能会修改另一个线程使用的全局变量的值，如果不注意，那么一旦出错，后果不堪设想。我们要尽可能多地使用const，如果一个全局变量只在本文件中使用，那么其用法和前面所介绍的函数局部变量没有什么区别。如果它要在多个文件间共享，那么就牵扯到一个存储类型的问题，主要有以下两种声明方式。

（1）使用extern修饰

例如：

```
/*pi.h*/
extern const double pi;
/*pi.c*/
const double pi=3.14;
```

编写好上面的源文件和头文件之后，如果在编程中需要使用变量pi，只需要包含头文件pi.h即可。

```
#include"pi.h"
```

或者把头文件中的那句声明在需要使用的源文件中重写一遍。这样做的结果是，整个程序链接完后，所有需要使用变量pi的共享一个存储区域。

（2）使用static修饰（静态外部存储类）

```
/*constant.h*/
static const double pi=3.14;
```

在需要使用这个变量的*.c文件中，必须包含这个头文件。前面的static一定不能少，否则链接的时候会警告该变量被多次定义。这样做的结果是，每个包含了constant.h的*.c文件都有一份该变量的副本，该变量实际上还是被定义了多次，占用了多个存储空间，不过加了关键字

static后，解决了文件间重定义的冲突。使用静态外部存储类的坏处是浪费了存储空间，导致链接完后的可执行文件变大。通常，在存储空间字节的变化不是太大的情况下，不是问题。好处是不用关心这个变量是在哪个文件中被初始化的。看看下面的代码。

```
#include＜stdio.h＞
int main（）
{
const int a=12;
const int*p=&a；//这个是指向常量的指针，指针指向一个常量
p++；//指针可以自加、自减
p--；//合法
int const*q=&a；//这个和上面的"const int*p=&a；"是一个意思
int b=12;
int*const r=&b；//这个就是常量指针（常指针），不能自加、自减，并且要初始化
//r++；//编译出错
const int*const t=&b；//这个就是指向常量的常指针，并且要初始化，用变量初始化
//t++；//编译出错
p=&b；//const指针可以指向const和非const对象
q=&b；//合法
return 0;
}
```

10.3 volatile修饰符

volatile修饰符的重要性对于从事嵌入式编程的程序员来说是不言而喻的，对volatile的了解程度常常被不少公司作为招聘嵌入式编程人员时衡量一个应聘者是否合格的参考标准之一。为什么volatile如此重要呢？因为嵌入式编程人员要经常同中断、底层硬件等打交道，这些都用到volatile，因此嵌入式程序员必须掌握volatile的使用。

其实就像读者所熟悉的const一样，volatile是一个类型修饰符。在开始讲解volatile之前，先讲解一个后面要用到的函数，知道如何使用此函数的读者可以跳过这部分内容。

原型：

```
int gettimeofday（struct timeval*tv,struct timezone*tz）;
```

头文件：#include＜sys/time.h＞。

功能：获取当前时间。

返回值：如果成功获取当前时间，那么返回0，否则返回-1，错误代码存于errno中。

gettimeofday（）会把目前的时间通过tv所指的结构返回，并将当地

时区的信息放到tz所指的结构中。

timeval结构定义为：

```
struct timeval{
long tv_sec;
long tv_usec;
};
```

timezone结构定义为：

```
struct timezone{
int tz_minuteswest;
int tz_dsttime;
};
```

先介绍timeval结构体，其中，tv_sec存放的是秒，tv_usec存放的是微秒。timezone成员变量我们很少使用，在此简单地介绍。timezone结构体在gettimeofday（）函数中的作用是把当地时区的信息存放到tz所指的结构中，其中，tz_minuteswest变量中存放的是与格林威治时间的时差，tz_dsttime存放的则是时间的修正方式。在此主要关注前一个成员变量timeval，后一个timezone在此不使用，因此在使用gettimeofday（）函数的时候把后一个参数设为NULL。下面先来看一段简单的代码。

```
#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;
```

```
    int j;
    for（j=0；j＜1000000；j++）
    ;
    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;
    }
```

运行结果：

```
    root@ubuntu:/home#./p
    运行时间为：0.002736
```

现在简单地分析一下以上代码，通过end.tv_sec-start.tv_sec得到终止时间与起始时间之间以秒为单位的时间间隔，然后通过end.tv_usec-start.tv_usec得到终止时间与起始时间之间以微妙为单位的时间间隔。由于时间单位的原因，在此将由（end.tv_sec-start.tv_sec）得到的结果乘以1 000 000转换为微秒进行计算，之后再使用“timeuse/=1000000；”将其转换为秒。了解了如何通过gettimeofday（）函数来测试start到end代码之间的运行时间，接下来介绍volatile修饰符。

通常，为了防止代码中一个变量在意想不到的情况下被改变，会将变量定义为volatile，从而使编译器不会自作主张地去“动”这个变量的值。准确地说就是，每次用到这个变量时都必须重新从内存中直接读取这个变量的值，而不是使用保存在寄存器中的备份。

在举例之前，先大概介绍一下Debug和Release编译模式的区别。通常，Debug模式被称为调试版本，它包含调试信息，并且不需要做任何优化，便于程序员调试程序。Release模式被称为发布版本，它往往需要进行各种优化，使程序在代码大小和运行速度上都是最优的，以便用户很好地使用。大致地知道了Debug和Release编译模式的区别之后，下面来看一段代码。

```
#include＜stdio.h＞
void main（）
{
int a=12;
printf（"a的值为：%d\n"，a）；
__asm{mov dword ptr[ebp-4]，0h}
int b=a;
printf（"b的值为：%d\n"，b）；
}
```

先分析上面的代码，其中使用了一句__asm{mov dword ptr[ebp-4]，0h}来修改变量a在内存中的值。前面已经讲解了Debug和Release编译模式的区别，现在来对比一下运行结果，在编译的时候别忘了选择编译运行的模式。

使用Debug编译模式的运行结果为：

```
a的值为：12
b的值为：0
```

使用Release编译模式的运行结果为：

```
a的值为：12
b的值为：12
```

从上面的运行结果可以发现，在Release模式下进行优化后b的值变为12，但是使用Debug模式时b的值为0。为什么会出现这样的情况呢？先不说答案，再来看看下面一段代码（注：使用VC++6.0编译运行）。

```
#include＜stdio.h＞
void main（）
{
int volatile a=12;
printf（"a的值为：%d\n"，a）；
__asm{mov dword ptr[ebp-4]，0h}
int b=a;
printf（"b的值为：%d\n"，b）；
}
```

使用Debug编译模式的运行结果为：

```
a的值为：12
b的值为：0
```

使用Release编译模式的运行结果为：

```
a的值为：12
b的值为：0
```

我们发现，在这种情况下不管是使用Debug模式还是使用Release模式都是一样的结果。现在来分析此前介绍的Debug和Release编译模式的区别。

先分析上一段代码，由于在Debug模式下并没有对代码进行优化，因此每次在代码中使用a值的时候都是从它的内存地址直接读取的，在使用“__asm{mov dword ptr[ebp-4]，0h}”语句改变了a值之后，接下来再次使用a值时从内存中直接读取，得到的是更新后的a值；但是在Release模式下运行代码的时候，发现b值为a之前的值，而不是更新后的a值，这是由于编译器在优化的过程中做了优化处理。编译器发现在对a赋值后没有再次改变a值，因此编译器把a值备份到一个寄存器中，在之后的操作中再次使用a值时就直接操作这个寄存器，而不去读取a的内存地址，因为读取寄存器的速度要快于直接读取内存的速度，这就使得读到的a值为之前的12，而不是更新后的0。

在第二段代码中使用了一个volatile修饰符，这样不管在什么模式下得到的都是更新后的a的值，因为volatile修饰符的作用就是告诉编译器不要对它所修饰的变量进行任何优化，每次取值都要直接从内存地址得到。从这里可以看出，对于代码中那些易变量，最好使用volatile进行修饰，以得到每次对其更新后的值。为了加深下读者的印象，再来看下面的一段代码。

```
#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;
int j;
for（j=0; j＜10000000; j++）
```

```
    ;
    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;
    }
```

与之前测试时间的代码一样，这里只是增大了for循环的次数。

先来看不使用优化的结果：

```
    root@ubuntu:/home#gcc time.c-o p
    root@ubuntu:/home#./p
    运行时间为：0.028260
```

而使用了优化的运行结果是：

```
    root@ubuntu:/home#gcc-o p time.c-O2
    root@ubuntu:/home#./p
    运行时间为：0.000001
```

从结果可以明显看出差距如此之大，但是如果在上面的代码中将“int j；”修改为“int volatile j；”，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;
    int volatile j;
    for（j=0; j＜10000000; j++）
    ;
```

```
    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;
    }
```

不使用优化的运行结果为：

```
    root@ubuntu:/home#gcc time.c-o p
    root@ubuntu:/home#./p
    运行时间为：0.027647
```

而使用了优化的运行结果为：

```
    root@ubuntu:/home#gcc-o p time.c-O2
    root@ubuntu:/home#./p
    运行时间为：0.027390
```

可以发现，此时不管是否使用优化语句，运行时间只有微小的差异，这微小的差异是计算机本身所导致的。因此通过对比可知，使用了volatile的变量在使用优化语句时，for循环并没有得到优化，因为for循环执行的是一个空操作，通常，使用了优化语句使这个for循环被优化后，根本就不执行，就好比编译器在编译的过程中将i的值设置为大于或者等于10 000 000的一个数，使for循环语句不会执行。但是由于使用了volatile，使编译器不会自作主张地改变i值，因此循环体得到了执行。举这个例子的目的是让读者牢记，如果定义了volatile变量，那么它就不会被编译器优化。

volatile还有哪些值得注意的地方呢？由于访问寄存器的速度要快过直接访问内存的速度，因此编译器一般都会减少对内存的访问，但是如果对变量加上volatile修饰，则编译器会保证对此变量的读写操作都不被优化。这样说可能有些抽象了，再看下面的代码，在此只简要地给出几步。

```
main（）
{
int i=o;
while（i==0）
{
……
}
}
```

分析以上代码，如果没有在while循环体结构中改变i的值，那么编译器在编译的过程中会将i的值备份到一个寄存器中，每次执行判断语句时就从该寄存器取值，这将是一个死循环，除非做如下的修改：

```
main（）
{
int volatile i=o;
while（i==0）
{
……
}
}
```

我们在i的前面加上了volatile，假设在while循环体内执行的是跟前面一段代码完全一样的操作，这时就不能说此时的while循环是一个死循环了，因为编译器不会再对i值进行“备份”操作了，每次执行判断的时

候都会直接从i的内存地址中读取，一旦其值发生变化，就退出循环体。

最后介绍volatile在实际应用中的适用场景有以下几个。

中断服务程序中修改的供其他程序检测的变量需要加volatile；

多任务环境下各任务间共享的标志应该加volatile；

存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能有不同的意义。

10.4 void和void*的区别

虽然在代码中经常见到的void和void*，但是可能不少初学者对它们并不是很了解，接下来看看它们适用于哪些情况。首先来看void,void使用最多的场景是设置函数的返回值类型，如：

```
#include<stdio.h>
void print ()
{
printf ("Hello World\n") ;
return;
}
int main ()
{
print () ;
return 0;
}
```

运行结果：

```
Hello World
```

分析上面的代码，print函数的返回值类型为void，意味着其没有返回值，需要注意的是，如果没有对函数的返回值声明任何的类型，那么默认返回值为int型，并不是void类型。看看下面的代码。

```
#include<stdio.h>
#include<stdlib.h>
sum (int a[], int n)
{
int i,sm; sm=0;
for (i=0; i<n; i++)
```

```
sm+=a[i];
return sm;
}
int main（void）
{
int a[4]={1，2，3，4};
printf（"%d\n"，sum（a，4））；
return 0;
}
```

运行结果：

```
10
```

void也可以作为函数的参数，如果参数没有返回值，那么可以用void来表示，如：

```
#include＜stdio.h＞
int main（void）
{
printf（"Hello World\n"）；
return 0;
}
```

运行结果：

```
Hello World
```

分析上面的运行结果，因为main函数没有携带任何参数，所以将其参数设置为void，表示其没有传递任何参数信息。

切记不要用void来定义参数，如：

```
void n;
```

以上定义在编译时会出现“error C2182：'n'：illegal use of type'void'”错误，由此可以看出，void的用途也就体现在两个方面：一是限定函数的返回值，二是限定函数的参数。

接下来介绍void*的使用。void*同样可以用来限定函数的返回值和参数，而且可以用来定义变量，但是切记不可将其理解为指向void类型的指针。看看下面的代码。

```
#include＜stdio.h＞
int main（）
{
void*n;
printf（"void*类型的指针定义成功\n"）；
return 0;
}
```

运行结果：

```
void*类型的指针定义成功
```

分析上面的运行结果，我们成功地定义了void*类型的指针，该指针并不是某些初学者所理解的空指针，而是一种特殊类型的指针，与之前定义的确定类型指针的不同之处在于，该指针是一种可以指向任意类型的指针，因此称作万能指针。看看下面的代码。

```
#include＜stdio.h＞
```

```
int main ()
{
void*n;
int a=0;
int*p;
char c='A';
p=&a;
printf ("p=%d, *p=%d\n", p, *p) ;
n=p;
printf ("n=%d, *n=%d\n", n, * (int*) n) ;
char*pc;
pc=&c;
printf ("pc=%d, *pc=%d\n", pc, *pc) ;
n=pc;
printf ("n=%d, *n=%d\n", n, * (char*) n) ;
return 0;
}
```

运行结果：

```
p=1245048, *p=0
n=1245048, *n=0
pc=1245040, *pc=65
n=1245040, *n=65
```

分析上面的代码，我们定义了可指向任何类型的指针n，首先将该指针指向int型变量，然后将该指针指向char类型的变量，由此可以看出，void定义的指针类型的确是一种可指向任何类型的指针，没有固定的指向。正是因为void类型的指针没有固定类型，所以不可以对void类型的指针进行自加或者自减运算，如：

```
#include<stdio.h>
int main ()
{
void*n;
int a=9;
int*pa;
```

```
pa=&a;
printf ("pa=%d\t*pa=%d", pa, *pa);
pa++;
printf ("pa++=%d", pa);
//printf ("%d", n++);
return 0;
}
```

运行结果：

```
pa=1245048*pa=9
pa++=1245052
```

分析上面的代码，我们对整型指针pa做了自加运算，同时在代码中还有一句打印语句，用于实现对void类型的指针进行自加运算。但是打印语句被注释掉了，如果没有注释掉，那么编译的时候就会出现如下错误：

```
error C2036: 'void*': unknown size
```

所以在使用void类型的指针时需要注意，由于其所指向的类型是不确定的，因此不可对其进行自加或者自减运算。

10.5 #define和typedef的本质区别

在讲解#define和typedef的本质区别之前，先来简单认识#define和typedef。#define在前面已经做了相应的介绍，是一个预处理指令。而typedef并不是一个预处理指令，通常用于为C语言中的标识符或关键字取“别名”，相应的过程是在编译的过程中进行的。

接下来通过代码来了解#define和typedef有什么样的本质区别，以及分别如何使用，先来看下面的代码。

```
#include<stdio.h>
#include<stdlib.h>
#define INT int
typedef short SHORT;
int main(void)
{
INT a=2;
SHORT b=9;
printf("a=%d\tb=%d\n", a,b);
return 0;
}
```

运行结果：

```
a=2 b=9
```

分析上面的代码，用#define和typedef分别为int和short取了一个别名，两者在这里有着相同的效果，但是建议采用typedef，因为如果定义稍微复杂点的别名，用#define可能会出现问题，例如：

```
#include＜stdio.h＞
#include＜stdlib.h＞
#define PINT int*
typedef short*PSHORT;
int main（void）
{
int a=4;
short b=9;
PINT pa1，pa2;
PSHORT pb1，pb2;
pa1=&a;
pa2=pa1;
pb1=&b;
pb2=pb1;
printf（"*pa1=%d\t*pa2=%d\n"，*pa1，*pa2）;
printf（"*pb1=%d\t*pb2=%d\n"，*pb1，*pb2）;
return 0;
}
```

先分析一下上面的代码，我们为int指针和short型指针分别取了别名。在代码中定义了int型变量a和short型变量b，看接下来的代码，本意是定义int指针pa1和pa2，short指针pb1和pb2，它们分别指向变量a和变量b，但是在编译时会出现如下错误：

```
error C2440: '=': cannot convert from'int*'to'int'
```

为什么会发生这样的错误呢？我们先进行宏扩展，将代码中的“PINT pa1，pa2；”扩展为“int*pa1，pa2；”，这时就能清楚地发现定义的pa2并不是一个指针，而是一个普通的整型变量，所以当采用整型指针的方式来使用它时就会出现上面的错误了。需要注意的是，通过typedef所取的别名并不是像宏一样只进行简单地替换，这样定义的别名作用于其后出现的所有变量。因此要在代码中取别名，建议采用typedef

来实现，以防出现类似的错误。再来看下面的代码。

```
#include<stdio.h>
#include<stdlib.h>
typedef int arr[4];
int main(void)
{
arr b={1, 2, 3, 4}, c={1, 2, 3, 4};
int i;
for(i=0; i<4; i++)
printf("b[%d]=%d\tc[%d]=%d\n", i,b[i], i,c[i]);
return 0;
}
```

运行结果：

```
b[0]=1 c[0]=1
b[1]=2 c[1]=2
b[2]=3 c[2]=3
b[3]=4 c[3]=4
```

分析上面的代码，用typedef为int arr[4]取了一个别名arr。这时arr就是int[4]类型，因此接下来在代码中定义的arr b,c其实就是int b[4]和int c[4]。由此可以发现，通过typedef可以使代码更加简洁，而且也更加符合我们对类型的理解。接下来再看看typedef在函数指针中的使用，代码如下：

```
#include<stdio.h>
#include<stdlib.h>
typedef int(*pfun)(int n);
int power(int n)
{
int pow,i;
pow=1;
for(i=0; i<n; i++)
```

```
    pow*=2;
    return pow;
    }
    int main（void）
    {
    pfun pwer;
    pwer=power;
    printf（"2的%d次方为：%d\n"，3，pwer（3））；
    return 0;
    }
```

运行结果：

```
    2的3次方为：8
```

分析上面的代码，通过typedef为函数指针int（*pfun）（int n）取了一个别名pfun，所以接下来在main函数中通过pfun定义的变量pwer实为int（*pwer）（int n），由此可见，通过typedef为函数指针取别名可以大大简化和美观代码。接下来看typedef在为结构体取别名时的使用。

```
    #include＜stdio.h＞
    #include＜stdlib.h＞
    typedef struct stu
    {
    char name[10];
    char nu[10];
    int score;
    }stu_inf;
    int main（void）
    {
    stu_inf sta={"杭萌"，"20336545"，256};
    printf（"sta.name：%s\tsta.nu：%s\tsta.score：%d\n",
sta.name,sta.nu,sta.score）；
    struct stu stb={"蒙蒙"，"20336546"，356};
    printf（"stb.name：%s\tstb.nu：%s\tstb.score：%d\n",
stb.name,stb.nu,stb.score）；
    return 0;
    }
```

运行结果：

```
sta.name：杭萌sta.nu：20336545 sta.score：256
stb.name：蒙蒙stb.nu：20336546 stb.score：356
```

分析上面的代码，定义了一个表示学生的简单结构体信息，在定义结构体的时候，采用typedef为结构体取了一个别名stu_inf，所以接下来在需要定义结构体类型的变量时，可以采用结构体的别名来定义，与采用struct stu的方式定义的效果是完全一致的。通过上面的结果可以发现，通过typedef为结构体定义别名之后，在很大程度上简化了代码。

10.6　条件语句的选用

前面讲解选择结构时已经接触了条件语句，读者对如何使用条件语句应该有了一定的认识。好的条件语句对我们查错也是很有帮助的，正如讲解在if语句中使用“==”时提到的，为了避免因疏忽导致难以查找的错误，我们要养成把常量写在左边，变量写在右边的习惯。这样，即使不小心少写了一个“=”，也会在编译时给出提示信息。如果把变量写在右边，当出现类似错误时，编译时不会给出任何的提示信息，会将其视为一个赋值语句，再将赋值后的值作为条件表达式的值。

如果在while循环语句中犯类似的错误，那么可能导致死循环，如：

```
#include＜stdio.h＞
#define N 3
int main（void）
{
int n,sum;
n=0;
sum=0;
while（n! =N）
{
n++;
sum+=n;
}
printf（"1～%d的整数之和为：%d\n"，N,sum）；
return 0;
}
```

运行结果：

```
1～3的整数之和为：6
```

分析上面的代码，其功能很清楚，如果在写while循环的时候不小心少写了其中的“！”号，那么while循环中的条件表达式就变为一条赋值语句“n=N”，如果给出的N非0，那么while循环就会变为死循环，在编译的时候也不会给出任何的错误或警告提示信息。如果按照上面的讲解，将N常量写在左边，这时如果少写了“！”号，那么编译的时候就会给出如下错误提示信息：

```
error C2106: '=': left operand must be l-value
```

这样就能够轻松地实现对错误的定位。

在条件语句中有一种特殊的短路逻辑表达式，对于这种短路逻辑表达式，如果条件语句中前面的条件已经满足整个表达式的要求，那么就不再执行后面的语句。例如下面的代码：

```
#include＜stdio.h＞
int main（void）
{
int i,a;
i=1;
a=7;
if（a＞9||i++）
printf（"i=%d\n", i）;
a=10;
if（a＞9||i++）
printf（"i=%d\n", i）;
return 0;
}
```

运行结果：

```
i=2
i=2
```

分析上面的运行结果，第一次将a赋值为7，这时在条件语句“a＞9||i++”中，仅根据前面的a＞9还不能判断这个表达式的值是否为真，所以还要继续执行接下来的i++。由于给定i的初始值为1，因此整个表达式的值为true，执行打印语句。接下来将a赋值为10，这时在条件语句中仅根据前面的a＞9即可判断整个条件语句的值为真，所以后面的i++被“短路”了，没有被执行，接下来打印输出的i值还是2。在上面的条件语句中采用的是或运算，接下来看与的情况。

```
#include＜stdio.h＞
int fun（char*str）
{
printf（"%s\n", str）;
return 0;
}
int main（void）
{
int a;
a=7;
if（a＞9＆＆fun（"a=7"））;
a=10;
if（a＞9＆＆fun（"a=10"））;
return 0;
}
```

运行结果：

```
a=10
```

分析上面的代码，定义了一个函数fun，该函数的返回值作为条件表达式的一部分，在将a赋值为7的时候，a＞9的值为false，所以整个表达式的值为false，条件语句中的后续语句不再被执行，这个时候并没有调用fun函数，但是接下来将a赋值为10，这时a＞9为true，会继续执行接下来的条件语句，函数fun得到调用，从而打印输出此时的信息。

在使用条件语句的时候还要注意的一点是，对于float或double类型的变量，不要使用！=或者==等比较符，而应该使用＞=、＞、＜等比较符，因为浮点运算存在精度问题，所以不能按照理论值来对其进行处理，应该根据计算机实际的处理方法来选用比较运算符。

10.7　函数realloc、malloc和calloc的区别

在C语言中，经常用于内存分配的函数有realloc、malloc、calloc，接下来看看它们的使用，以及它们之间的区别所在。

（1）realloc函数

原型：

```
extern void*realloc（void*mem_address,unsigned int newsize）；
```

语法：指针名=（数据类型*）realloc（要改变内存大小的指针名，新的大小）。

头文件：stdlib.h。

功能：先按照newsize指定的大小分配空间，将原有数据从头到尾复制到新分配的内存区域，而后释放原来mem_address所指向的内存区域，同时返回新分配的内存区域的首地址，即重新分配存储器块的地址。

返回值：如果重新分配成功，则返回指向被分配内存的指针，否则返回空指针NULL。由于该函数返回的是void型的指针，因此需要进行类型转换。

注意　在这里，原始内存中的数据保持不变。当不再使用内存时，应使用free函数将内存块释放。

（2）malloc函数

原型：

```
extern void*malloc（unsigned int num_bytes）；
```

头文件：在C++6.0中为malloc.h或stdlib.h。

功能：分配字节数为num_bytes的内存块。

返回值：如果分配成功，则返回指向被分配内存的指针，否则返回空指针NULL。当不再使用内存时，应使用free函数将内存块释放。

说明　关于该函数的原型，在旧版本中，malloc返回的是char型指针，而新的ANSIC标准规定，该函数返回的是void型指针，因此要进行类型转换。

（3）calloc函数

原型：

```
extern void*calloc（unsigned n,unsigned size）；
```

头文件：stdlib.h或malloc.h。

功能：在内存的动态存储区中分配n个长度为size的连续空间，该函数返回一个指向分配起始地址的指针；如果分配不成功，则返回NULL。

注意　calloc函数在动态分配完内存后，自动初始化该内存空间为零，而malloc不初始化，数据是随机的垃圾数据。另外，calloc函数返回的也是void类型的指针，也需要进行类型转换。

下面综合起来看这3个函数的使用。

```
#include＜stdio.h＞
#include＜stdlib.h＞
int main（void）
{
int num=5;
int i;
int*p=（int*）malloc（num*sizeof（int））；
int*p1=（int*）calloc（num,sizeof（int））；
printf（"采用malloc分配后的内存单元中的数据\n"）；
for（i=0；i＜num；i++）
{
printf（"%d\t"，p[i]）；
}
printf（"\n采用calloc分配后的内存单元中的数据\n"）；
for（i=0；i＜num；i++）
{
printf（"%d\t"，p1[i]）；
}
printf（"\n使用realloc函数前p的值为：%p\n"，p）；
for（i=0；i＜num；i++）
p[i]=i+1;
for（i=0；i＜num；i++）
printf（"p[%d]=%d\t"，i,p[i]）；
num=3;
p=（int*）realloc（p,num*sizeof（int））；
```

```
    printf ("\n\n使用realloc函数减小指针p所指向的内存单元后的p值为：%p\n",
p);
    for (i=0; i<num; i++)
    printf ("p[%d]=%d\t", i,p[i]);
    num=6;
    p= (int*) realloc (p,num*sizeof (int));
    printf ("\n\n使用realloc函数第一次扩展指针p所指向的内存单元后的p值为：
%p\n", p);
    for (i=0; i<num; i++)
    {
    printf ("p[%d]=%d\t", i,p[i]);
    if (0== (i+1) %3)
    printf ("\n");
    }
    num=12;
    p= (int*) realloc (p,num*sizeof (int));
    printf ("\n使用realloc函数第二次扩展指针p所指向的内存单元后的p值为：
%p\n", p);
    for (i=0; i<num; i++)
    {
    printf ("p[%d]=%d\t", i,p[i]);
    if (0== (i+1) %3)
    printf ("\n");
    }
    free (p);
    free (p1);
    return 0;
    }
```

运行结果：

```
    采用malloc分配后的内存单元中的数据
    -842150451-842150451-842150451-842150451-842150451
    采用calloc分配后的内存单元中的数据
    0 0 0 0 0
    使用realloc函数前p的值为：00382F48
    p[0]=1 p[1]=2 p[2]=3 p[3]=4 p[4]=5
    使用realloc函数减小指针p所指向的内存单元后的p值为：00382F48
    p[0]=1 p[1]=2 p[2]=3
    使用realloc函数第一次扩展指针p所指向的内存单元后的p值为：00382F48
    p[0]=1 p[1]=2 p[2]=3
    p[3]=-842150451 p[4]=-842150451 p[5]=-842150451
    使用realloc函数第二次扩展指针p所指向的内存单元后的p值为：00380FE0
    p[0]=1 p[1]=2 p[2]=3
    p[3]=-842150451 p[4]=-842150451 p[5]=-842150451
```

```
p[6]=-842150451 p[7]=-842150451 p[8]=-842150451
p[9]=-842150451 p[10]=-842150451 p[11]=-842150451
```

分析上面的运行结果，先采用malloc函数和calloc函数分别对int型指针p和p1分配内存空间，对比采用这两个函数进行内存分配后的内存单元中的值可知，采用malloc分配后的内存单元中的数据都是随机的垃圾数据，而采用calloc函数分配的内存空间中的数据都被自动初始化为0。

接下来，对p指针所指向的内存单元进行初始化处理，同时打印输出此时p的值，然后采用realloc函数对所分配的内存单元进行缩减。注意，缩减后的内存单元的起始地址并没有改变，但是后面通过realloc函数对其进行了两次扩展。对比两次扩展的变化，第一次扩展时分配的内存单元的地址其实并没有改变，但是再次进行扩展时其起始地址已经发生了变化，也就是说，现在所分配的内存单元已经不再是以前所分配的内存单元了，而是将原有数据从头到尾复制到新分配的内存区域中，同时释放原来mem_address所指向的内存区域，返回新分配的内存区域的首地址，即重新分配存储器块的地址。

两次扩展不同的原因是，realloc函数直接从堆上现存的数据后面试图获取所要附加的字节，如果原先分配的内存后面有足够的空闲空间用于扩展，那么就在其后面附上所需字节数的内存空间，得到的是一块连续的内存，因此返回原先分配的内存的起始地址。如果原先分配的内存后面没有足够的空闲空间用来分配，那么从堆中另外找一块扩展所需大

小的内存，并把原来内存空间中的内容复制到新扩展的内存中，同时释放原来所占用的内存空间，返回最新分配内存的起始地址。

10.8 函数和宏

在此之前已对函数和宏分别做了相应的讲解，但是没有对比分析它们之间的区别。接下来就来看函数和宏之间究竟有什么样的区别。先看下面的代码。

```
#include<stdio.h>
#define max(x,y) x>y?x:y
int fmax(int x,int y)
{
return x>y?x:y;
}
int main(void)
{
int x,y;
x=9;
y=6;
printf("max(x,y)=%d\n",max(x,y));
printf("fmax(x,y)=%d\n",fmax(x,y));
return 0;
}
```

运行结果：

```
max(x,y)=9
fmax(x,y)=9
```

分析上面的代码，分别采用宏和函数的方式来实现取两数中的较大值，下面就基于此来分析函数和宏之间的区别。

宏是在预处理阶段完成的，只做字符串的替换，而不求值，也不为

其分配内存。

函数是在程序运行阶段进行的，所以需要为其分配内存；宏在预处理阶段进行字符串替换时使代码变长，而函数不会。

由于宏是在预处理阶段进行的，所以宏不占用编译时间，而函数是在运行阶段进行的，同时在函数调用过程中涉及内存的分配、参数的传递等操作，所以会占用程序的运行时间。

由于预处理是在编译前进行的，所以预处理并不进行语法检查，而函数调用是在程序运行期间进行的，所以函数调用进行语法检查。

函数只有一个返回值，而利用宏可以得到多个返回值。

10.9　运算符==、=和！=的区别

先来看看这3个运算符各自的使用场合，==和！=是关系运算符，其优先级要高于赋值运算符=。由于三者都包含等号，因此在使用的时候要格外小心。下面来看在使用这三个运算符的过程中如果不小心写错会带来怎样的后果。下面的代码实现的是比较两个量之间的大小关系，如：

```
a=2;
b=3;
if（a==b）
printf（"a与b大小相等。"）；
```

但是在程序时不小心少写了一个“=”，上面的代码就变为：

```
if（a=b）
printf（"a与b大小相等。"）；
```

分析上面的变化，就因为少写了一个“=”，结果却截然不同。在使用关系运算符==时，由于a和b并不相等，因此不会打印出信息“a与b大小相等。”，接下来由于少写了一个“=”，关系运算符变为了赋值运算符，即变为了“a=b”，这是一个赋值表达式，最终的表达式的值就是b对a的赋值。此时表达式的值为真，因此这时会输出错误的信息。

还有一点不得不说的是优先级，由于对优先级的不清楚而导致错误

也是很常见的。这里以a=b！=c和（a=b）！=c之间的区别为例进行分析，假设a=1，b=2，c=2。先分析a=b！=c表达式，由于“！=”的优先级要高于“=”，因此先比较b与c的关系，因为b等于c，所以表达式的值为假，即a的值为0。接下来分析（a=b）！=c表达式，由于加了括号，因此此时a的值为2，而最终表达式（a=b）！=c的值为0。

10.10　类型转换

类型转换在C语言中是一个容易被人忽视的概念，却频繁地出现在程序的表达式中，所以C语言编程人员都应该掌握类型转换。类型转换可分为隐式转换和显式转换，接下来分别介绍这两种转换方式。

1.隐式转换

隐式转换又称为自动转换，这种转换会在计算时自动将表达式中的常量和变量转换成正确的类型，以确保计算结果的正确，而转换的方式则遵循由低类型向高类型的转换。我们可以通过图10-1来了解隐式转换过程。

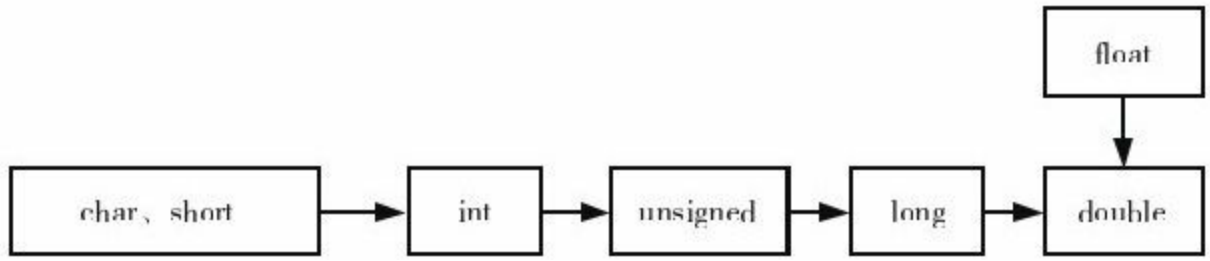

图　10-1　隐式转换

为了让读者深入理解隐式转换，看看以下两段代码及其运行结果。第一段：

```
#include<stdio.h>
int main()
{
unsigned short i=0;
```

```
unsigned short n=0;
while（i＜n-1）
{
printf（"进入while循环\n"）；
break;
}
return 0;
}
```

因为运行时没有任何输出，所以可以推断并没有进入while循环。

第二段：

```
#include＜stdio.h＞
int main（）
{
unsigned short i=0;
unsigned long n=0;
while（i＜n-1）
{
printf（"进入while循环\n"）；
break;
}
return 0;
}
```

运行结果：

```
进入while循环
```

对比以上两段代码，为什么第二段代码能够进入while循环，而第一段代码却不能呢？具体分析如下。

在第一段代码中，由于定义的i和n都是无符号短整型，因此在执行i

＜n-1时，n-1中的n为无符号短整型，而数字常量1为有符号整型，类型不匹配，会导致隐式类型转换。由于n的值为0，因此发生隐式转换后的值还是0，执行n-1之后的值为有符号整数-1，此时while循环中的条件为0＜-1，显然不成立，当然不会进入while循环。

在第二段代码中，定义的i同样为短整型，而n却为长整型，所以在执行i＜n-1时，同样会因为n和1的类型不匹配而发生隐式类型转换。由图10-1可知，转换方式为将1转换为无符号长整型，转换后的结果仍然为1，所以接下来执行n-1时得到的就是无符号长整型0xffffffff，此时while循环中的条件为0＜0xffffffff，显然成立，此时能够进入while循环。

分析上面的隐式转换，当表达式中出现多种类型时要格外注意，以免出现结果与期望的不符，所以建议在混有多种类型的表达式中采用显式转换。

2.显式转换

显式转换又称为强制转换，其一般格式为：

（类型说明符）（表达式）；

强制转换是将表达式的结果转换为类型说明符所表示的类型，当然这里所指的表达式也可以是变量或常量。下面就来看显式转换的常用方

法。

```
x=（int）23.49;
```

最终结果为23，从这里可以看出，将浮点数转换为整数时小数部分将被截除掉。

```
c=（int）a+b;
c=（int）（a+b）;
```

看看上面两行代码，第一行代码表示将a先转换为整型再与b相加，而第二行代码表示先将a与b相加，再将最终的结果转换为整型。

```
int a=（double）6/5;
```

在这一行代码中，先通过强制转换将6转换为双精度型，由于5是整型，因此在计算时会将其隐式转换为双精度型，计算结果为1.2。由于等号的左边是整型，因此又会对结果1.2隐式转换为整型，小数部分被截除掉，这样最终结果为a等于1。

第11章　必须掌握的常用算法

大多数的编程人员都知道，瑞士计算机科学家尼克劳斯·沃思提出了一个著名的公式：“数据结构十算法=程序”。从公式中可以看出，算法是构成程序的一个重要部分，而所谓算法，可以理解为解决某一确定问题所采用的具体步骤和方法，也就是说，给定初始状态或输入数据后，能够得出期望的输出结果。由于不同的算法完成同样的任务的效率可能是不同的，因此在解决问题时经常面对算法的选择问题，如何评判算法的优劣，以及必须掌握的常见算法有哪些，这将是本章所要讲解的重点。

11.1 时间复杂度

在讲解常用算法之前，不得不提的一个概念就是时间复杂度，它是用来衡量一个算法优劣的重要指标。下面就来介绍什么是时间复杂度以及如何计算时间复杂度。

提到时间复杂度，不得不提一下语句频度，它指的是算法中每条语句被执行的次数。如果这样简单地讲解不够明朗，那么通过一段代码具体介绍什么是语句频度，看看下面的代码。

```
for（i=0；i＜n；i++）
a+=1;
```

在上面的代码中，a+=1总共被执行了n次，所以a+=1语句的语句频度为n。

时间复杂度指的是算法所需的执行时间，而算法的执行时间就是代码中每条语句执行时间的总和，记为T（n），其中，参数n是问题规模，当n变化时，T（n）也会随之变化。

还有一个概念不得不提，那就是渐进时间复杂度，它指的是当表示问题规模的n值趋于无穷大时的时间复杂度，将其记做：T（n）=O（f（n）），其中，f（n）和T（n）是同数量级的，即当n趋向于无穷大的时候，T（n）/f（n）的极限值为非零常数。因此，f（n）越大，

所对应的时间复杂度就越大，而f（n）的取值通常用执行语句中语句频度最大的值的数量级来描述。如果代码中某条语句的语句频度最大，其值为$3n^4+6n+9$，那么f（n）的取值就为其数量级n^4。

由于在实际运行中算法执行时间受多方面因素的影响，不可能得到精确的计算时间，加之，引入时间复杂度的目的仅仅是为了对比算法的优劣性，并不是要计算出算法实际的运行时间，只需要知道哪种算法在执行时间上有优势即可。因此在计算时间复杂度时采用的都是理想情况，假设每条语句每次的执行时间完全一致，都为单位时间，算法的执行时间等于所有被执行语句的语句频度之和乘以执行每条语句所需的单位时间。由于单位时间都是一样的，因此对比时间复杂度就转换为对比算法中所执行语句的语句频度之和。在实际对比算法的时间复杂度时，主要参照标准是算法的渐近时间复杂度，对比其中的f（n）的大小，因此经常将渐近时间复杂度T（n）=O（f（n））简称为时间复杂度。接下来通过代码示例介绍如何计算渐进时间复杂度。

示例1：

```
for（i=0；i＜n；i++）
{
a+=i;
for（j=0；j＜n；j++）
{
b+=i+j;
}
}
```

在上面的代码中使用了一个二重循环，要想知道该算法的时间复杂度，就需要找出该语句中语句频度最大为多少。由于a+=i语句的语句频度为n，而b+=i+j的语句频度为n的平方，因此可以将其时间复杂度表示为T（n）=O（n^2）。

示例2：

```
for（i=0; i＜n; i++）
for（j=0; j＜n; j++）
for（k=0; k＜n; k++）
sum+=1;
```

从上面的三重循环中可以看出，语句频度最大的语句是sum+=1，其值为n^3，所以，可以将其时间复杂度表示为T（n）=O（n^3）。适当地修改上面的代码，得到的代码如下：

示例3：

```
for（i=1; i＜=n; i++）
for（j=1; j＜=i; j++）
for（k=1; k＜=j; k++）
sum+=1;
```

在上面的代码中，仅对内层循环做了简单的修改，此时的三重循环的时间复杂度又是多少呢？当最外层循环中i值为1时，sum+=1被执行了1次，当i值为2时，sum+=1被执行了1+2次，依此递推，当i值为n时，sum+=1被执行了1+2+……+n次，由此可以得出sum+=1的语句频度为

1+（1+2）+（1+2+3）+……+（1+2+3+……+n）=n（n+1）（n+2）/6，所以其时间复杂度可表示为T（n）=O（n^3）。

对比上面代码的时间复杂度可以发现，对于多重循环语句，其时间复杂度主要由最内层循环体中的语句来决定。同时在使用时间复杂度的时候需要注意的一点就是，渐进估计是对算法的理论分析和大致比较，并不能够准确表示算法之间的关系。对于上面的示例2和示例3，它们的时间复杂度都是T（n）=O（n^3），但是通过代码可知它们实际的执行时间并不相同；还有就是时间复杂度为O（n^2）算法在n较小的情况下可能比时间复杂度为O（nlogn）的算法运行得更快。当然，随着n值的增大，复杂度变化较慢的算法必然工作得更快。

对于常见的时间复杂度，按数量级递增排列可依次表示为：常数阶O（1）、对数阶O（log2n）、线性阶O（n）、线性对数阶O（nlog2n）、平方阶O（n^2）、立方阶O（n^3）、指数阶O（2^n）、阶乘O（n！）、n次方阶O（n^n）。它们之间的大小关系如下：

O（1）＜O（log2n）＜O（n）＜O（nlog2n）＜O（n^2）＜O（n^3）＜O（2^n）＜O（n！）＜O（n^n）

从上面的大小关系中发现，基本满足这样一个关系：

常数＜对数＜多项式＜指数

11.2 冒泡法排序

本节介绍冒泡法排序的实现方法。首先确定所要采用的排序规则是升序还是降序。所谓升序，就是将给定的一组数据按照由小到大的顺序排列，而降序则与之相反。冒泡法排序就是依次比较相邻的两个数，根据采用的排序规则决定是否交换两个数的位置。

在此以升序为例来进行讲解。

第一轮：先比较第一个数和第二个数，将小数放前，大数放后，然后比较第二个数和第三个数，将小数放前，大数放后，如此继续，直至比较最后两个数，将小数放前，大数放后。到第一轮结束的时候，已经成功地将最大的数放到了最后。

第二轮：可能由于第二个数和第三个数的交换，使第一个数不再小于第二个数，所以新一轮的比较仍从第一对数开始比较，将小数放前，大数放后，一直比较到倒数第二个数，由于最后一个位置上的数已经是最大的，所以无须再对其进行比较。到第二轮结束的时候，在倒数第二个位置上得到一个新的最大数，也就是给定数据中的第二大的数。

重复以上过程，直到最后一轮比较完第一个数和第二个数，完成整个排序功能。接下来看下面的代码。

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

```
void bubble_sort（int arr[]，int n）
{
int i,j，temp;
for（i=0；i＜n-1；i++）
for（j=0；j＜n-i-1；j++）
{
if（arr[j]＞arr[j+1]）
{
temp=arr[j];
arr[j]=arr[j+1];
arr[j+1]=temp;
}
}
return;
}
int main（）
{
int a[]={91，54，32，87，46};
int i;
printf（"排序前\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
bubble_sort（a，5）；
printf（"\n排序后\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
return 0;
}
```

运行结果：

```
排序前
91 54 32 87 46
排序后
32 46 54 87 91
```

上面的代码，成功地实现了对给定数组的排序，接下来通过图来分析每轮的排序工作，图11-1演示的是第一轮的比较操作。

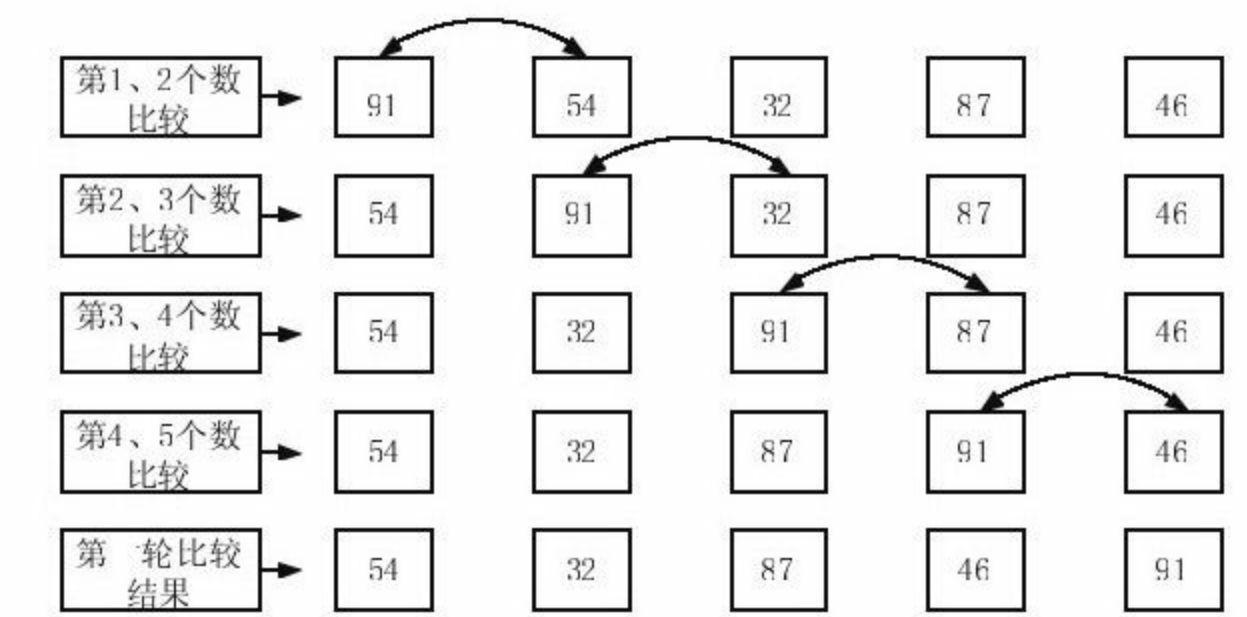

图　11-1　第一轮的比较操作

图11-1中第一轮比较后的结果是进行下一轮排序的基础，接下来通过图11-2来了解第二轮的比较操作。

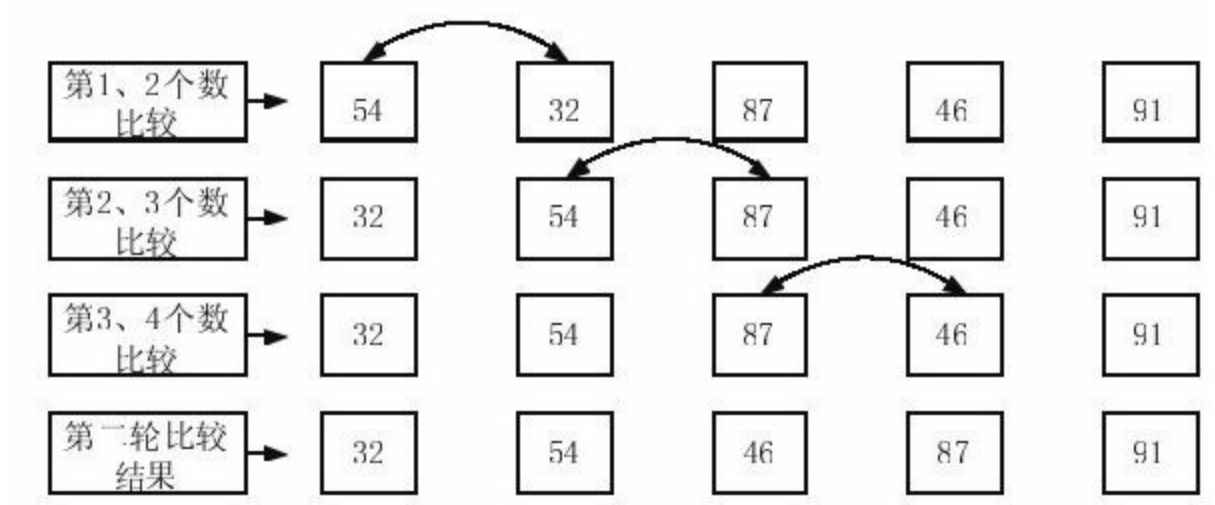

图　11-2　第二轮的比较操作

下面在第二轮比较结果的基础上进行下一轮比较。图11-3为第三轮

的比较操作。

分析上面的比较结果可以发现，第三轮结束之后的比较结果就是最终的结果了，没有必要再进行接下来的比较操作了，但是计算机并不知道，它还会进行接下来的比较操作。为了改进冒泡法排序的这一缺点，防止对已经完成排序功能的数据进行没有任何意义的比较操作，我们定义一个标识符flag，在每轮比较操作之前将flag赋值为1，如果在该轮比较操作中发现排序没有完成，那么就在该轮结束时将其赋值为0，否则不改变其flag值，在下一轮开始前对flag进行判断，如果其值为1，表示在上一轮排序中发现已经完成了排序功能，那么就结束冒泡法排序的比较操作。接下来看改进后的代码。

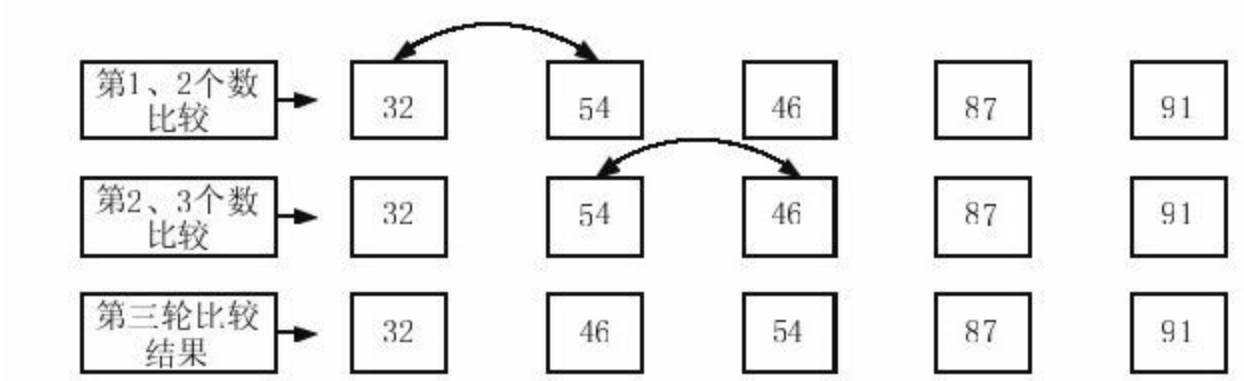

图　11-3　第三轮的比较操作

```
#include＜stdio.h＞
void bubble_sort（int arr[], int n）
{
int i,j, flag,temp;
for（i=0; i＜n-1; i++）
{
flag=1;
for（j=0; j＜n-i-1; j++）
{
if（arr[j]＞arr[j+1]）
```

```
{
temp=arr[j];
arr[j]=arr[j+1];
arr[j+1]=temp;
flag=0;
}
}
if（1==flag）
break;
}
printf（"\n比较操作进行了%d轮"，i+1）；
return;
}
int main（）
{
int a[]={12，54，67，89，96};
int i;
printf（"排序前\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
bubble_sort（a，5）；
printf（"\n排序后\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
return 0;
}
```

运行结果：

```
排序前
12 54 67 89 96
比较操作进行了1轮
排序后
12 54 67 89 96
```

在上面的代码中，在排序中加了一个打印输出语句，用于输出比较操作进行了多少轮。由于最初给定的数组是一个有序的数组，因此比较操作只进行了一轮，但是如果没有在代码中使用标识符flag，那么即使是对有序的数组进行排序，同样要进行5轮排序操作。使用标识符之

后，算法的效率得到了提高。

根据前面讲解的时间复杂度可以得出，冒泡法排序的时间复杂度是O（n^2）。

11.3 选择法排序

选择法排序同样是一种相对较为简单和容易实现的排序方法，下面介绍这种排序的实现方法。首先同样需要确定排序的规则，然后以给定排序数组的第一个数为基准，将其后的数依次与其进行比较，根据排序规则来确定是否交换两个数。

在此同样以升序为例来讲解。

第一轮：以第一个数为基准，从第二个数开始，将后面的数据与第一个数一一进行对比，如果有比第一个数还小的，那么就将第一个数与该数进行交换，直到比较到最后一个数为止。

第二轮：由于第一轮比较结束后，第一个数是给定排序数中最小的数，因此第二轮以第二个数为基准，将其后的数依次与其进行比较操作，满足条件则进行交换，直到比较到最后一个数为止。

重复以上操作，直到以倒数第二个数为基准，进行最后一轮比较操作为止。分析下面的实现代码。

```
#include<stdio.h>
void sort(int a[], int n)
{
int i,j, k,tmp;
for(i=0; i<n-1; i++)
{
k=i;
for(j=i+1; j<n; j++)
```

```
{
if（a[j]＜a[k]）
k=j;
}
if（k! =i）
{
tmp=a[i];
a[i]=a[k];
a[k]=tmp;
}
}
}
int main（void）
{
int i;
int a[5]={32，12，56，78，43};
printf（"排序前\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
sort（a，5）；
printf（"\n排序后\n"）；
for（i=0；i＜5；i++）
printf（"%d\t"，a[i]）；
return 0;
}
```

运行结果：

```
排序前
32 12 56 78 43
排序后
12 32 43 56 78
```

对比排序前后的打印结果发现，成功地实现了通过选择排序法对给定的数组进行排序操作。下面先通过图11-4来了解第一轮的比较操作。

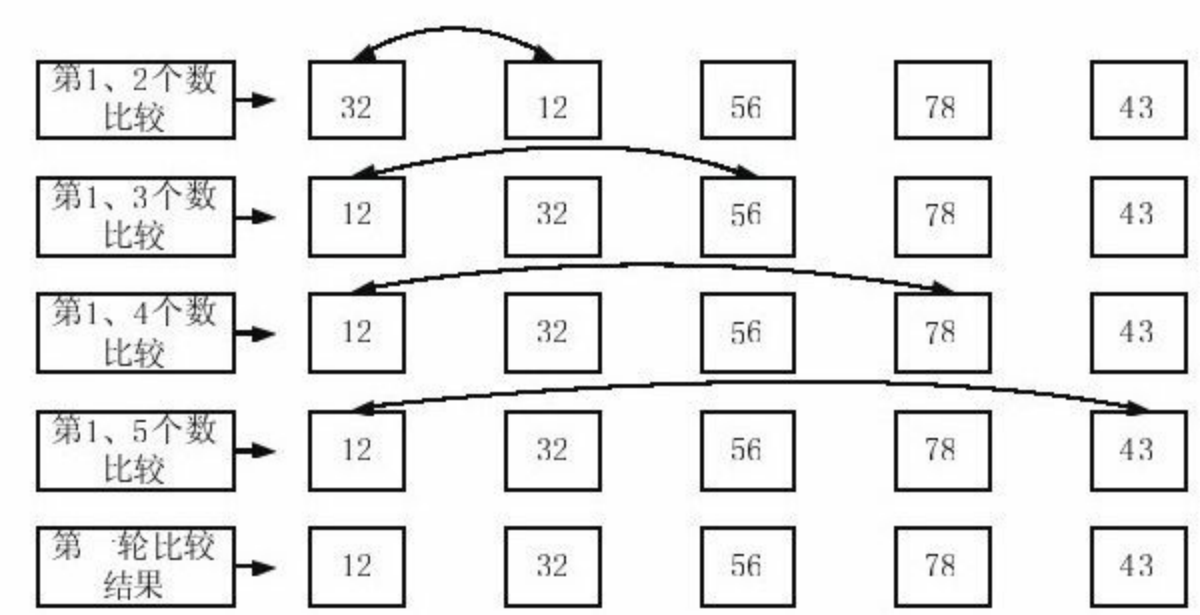

图　11-4　第一轮的比较操作

接下来在第一轮比较结果的基础上进行第二轮比较操作，如图11-5所示。

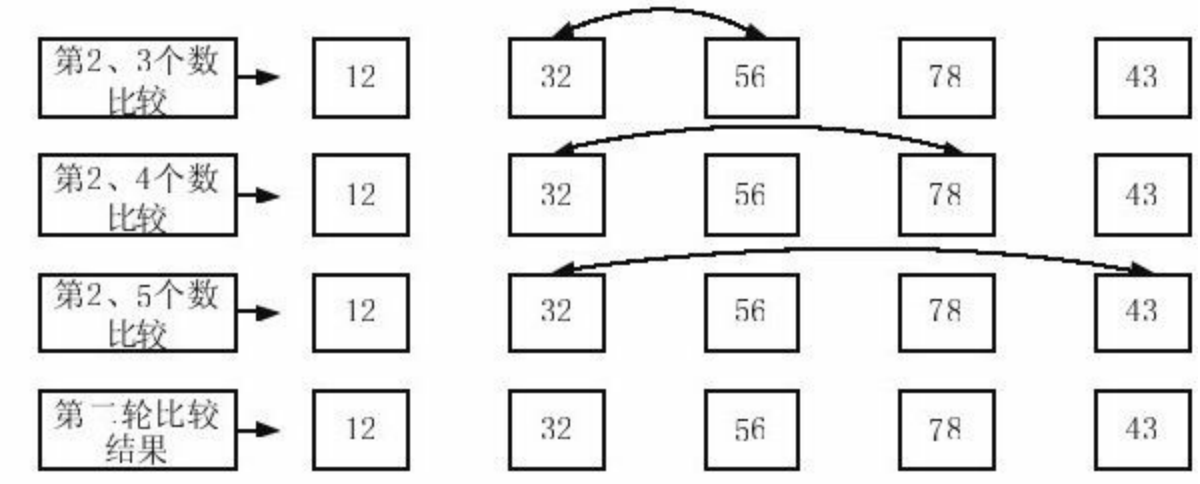

图　11-5　第二轮的比较操作

分析上面的比较结果，第二轮比较结果与第一轮的比较结果完全一

致，没有将第二个数与接下来的任何一个数进行交换操作。下面通过图11-6来了解在此基础上进行的第三轮的比较操作。

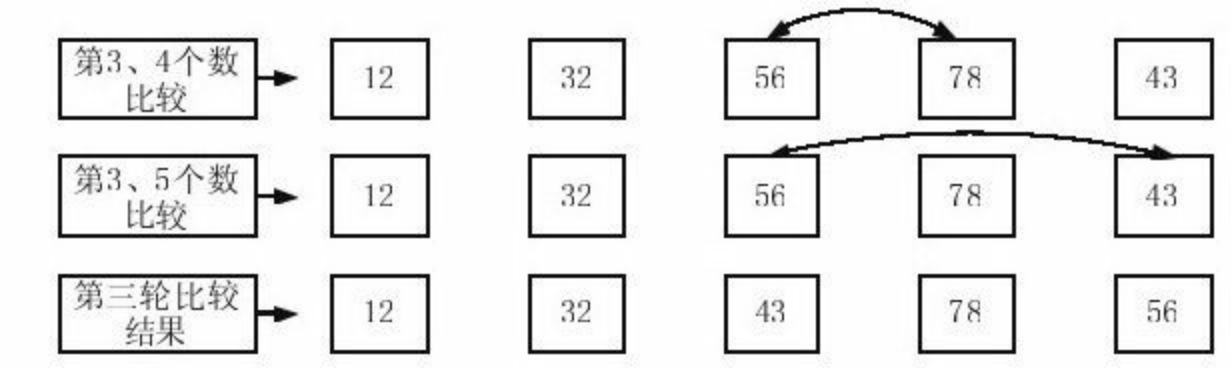

图　11-6　第三轮的比较操作

在第三轮的比较操作中，由于第5个数比第三个数要小，因此进行了一次交换操作。接下来再来通过图11-7看看第四轮的比较操作。

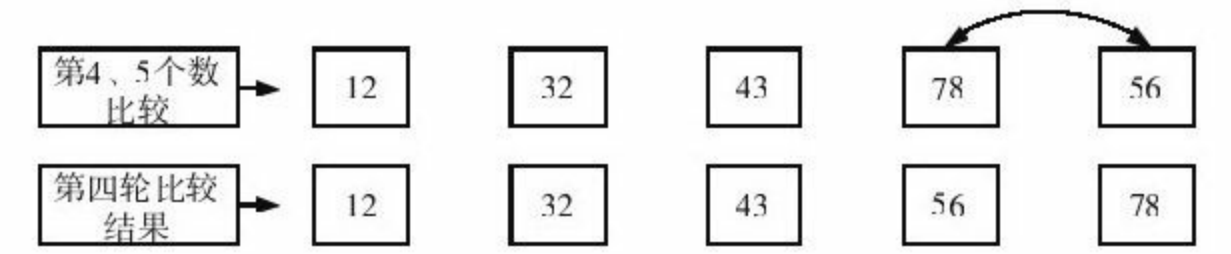

图　11-7　第四轮的比较操作

由上面的比较结果可知，这时已经成功地实现了对数组的排序。

11.4　快速排序

快速排序是对冒泡法排序的一种改进，该方法由C.A.R.Hoare在1962年提出。此方法的基本思想是：将所要进行排序的数分为左右两个部分，其中一部分的所有数据都比另外一部分的数据小，然后将所分得的两部分数据进行同样的划分，重复执行以上的划分操作，直到所有要进行排序的数据变为有序为止。

可能仅根据基本思想对快速排序的认识并不深，接下来以对n个无序数列A[0]，A[1]……，A[n-1]采用快速排序方法进行升序排列为例进行讲解。

（1）定义两个变量low和high，将low、high分别设置为要进行排序的序列的起始元素和最后一个元素的下标。第一次，low和high的取值分别为0和n-1，接下来的每次取值由划分得到的序列起始元素和最后一个元素的下标来决定。

（2）定义一个变量key，接下来以key的取值为基准将数组A划分为左右两个部分，通常，key值为要进行排序序列的第一个元素值。第一次的取值为A[0]，以后每次取值由要划分序列的起始元素决定。

（3）从high所指向的数组元素开始向左扫描，扫描的同时将下标为high的数组元素依次与划分基准值key进行比较操作，直到high不大于

low或找到第一个小于基准值key的数组元素，然后将该值赋值给low所指向的数组元素，同时将low右移一个位置。

（4）如果low依然小于high，那么由low所指向的数组元素开始向右扫描，扫描的同时将下标为low的数组元素值依次与划分的基准值key进行比较操作，直到low不小于high或找到第一个大于基准值key的数组元素，然后将该值赋给high所指向的数组元素，同时将high左移一个位置。

（5）重复步骤（3）（4），直到low的值不小于high为止，这时成功划分后得到的左右两部分分别为A[low……pos-1]和A[pos+1……high]，其中，pos下标所对应的数组元素的值就是进行划分的基准值key，所以在划分结束时还要将下标为pos的数组元素赋值为key。

（6）将划分得到的左右两部分A[low……pos-1]和A[pos+1……high]继续采用以上操作步骤进行划分，直到得到有序序列为止。

为了能够加深读者的理解，接下来通过一段代码来了解快速排序的具体实现方法。

```
#include＜stdio.h＞
#define N 6
int partition（int arr[], int low,int high）
{
int key;
key=arr[low];
while（low＜high）
{
```

```
while（low＜high＆＆arr[high]＞=key)
high--;
if（low＜high)
arr[low++]=arr[high];
while（low＜high＆＆arr[low]＜=key)
low++;
if（low＜high)
arr[high--]=arr[low];
}
arr[low]=key;
return low;
}
void quick_sort（int arr[]，int start,int end)
{
int pos;
if（start＜end)
{
pos=partition（arr,start,end)；
quick_sort（arr,start,pos-1)；
quick_sort（arr,pos+1，end)；
}
return;
}
int main（void)
{
int i;
int arr[N]={32，12，7，78，23，45};
printf（"排序前\n")；
for（i=0；i＜N；i++)
printf（"%d\t"，arr[i])；
quick_sort（arr，0，N-1)；
printf（"\n排序后\n")；
for（i=0；i＜N；i++)
printf（"%d\t"，arr[i])；
printf（"\n")；
return 0;
}
```

运行结果：

```
排序前
32 12 7 78 23 45
排序后
7 12 23 32 45 78
```

在上面的代码中，根据前面介绍的步骤一步步实现了快速排序算法。接下来通过图11-8来演示第一次划分操作。

在第一次划分操作中，先进行初始设置，key的值是进行划分的基准，其值为要划分数组的第一个元素值，在上面的排序序列中为第一个元素值32，同时将low设置为要排序数组中第一个元素的下标，第一次排序操作时其值为0，将high设置为要排序序列最后一个元素的下标，在上面的排序序列中其第一次取值为5。先将下标为high的数组元素与key进行比较，由于该元素值大于key，因此high向左移动一个位置继续扫描。由于接下来的值为23，小于key的值，因此将23赋值给下标为low所指向的数组元素。接下来将low右移一个位置，将low所指向的数组元素的值与key进行比较，由于接下来的12、7都小于key，因此low继续右移扫描，直至下标low所指向的数组元素的值为78即大于key为止，将78赋值给下标为high所指向的数组元素，同时将high左移一个位置。接下来由于low不再小于high，划分结束。需要注意的是，在进行划分的过程中，都是将扫描的值与key的值进行对比，如果小于key，那么将该值赋值给数组中的另外一个元素，而该元素的值并没有改变。从图11-8中可以看出这一点，所以需要在划分的最后将作为划分基准的key值赋值给下标为pos的数组元素，这个元素不再参与接下来的划分操作。

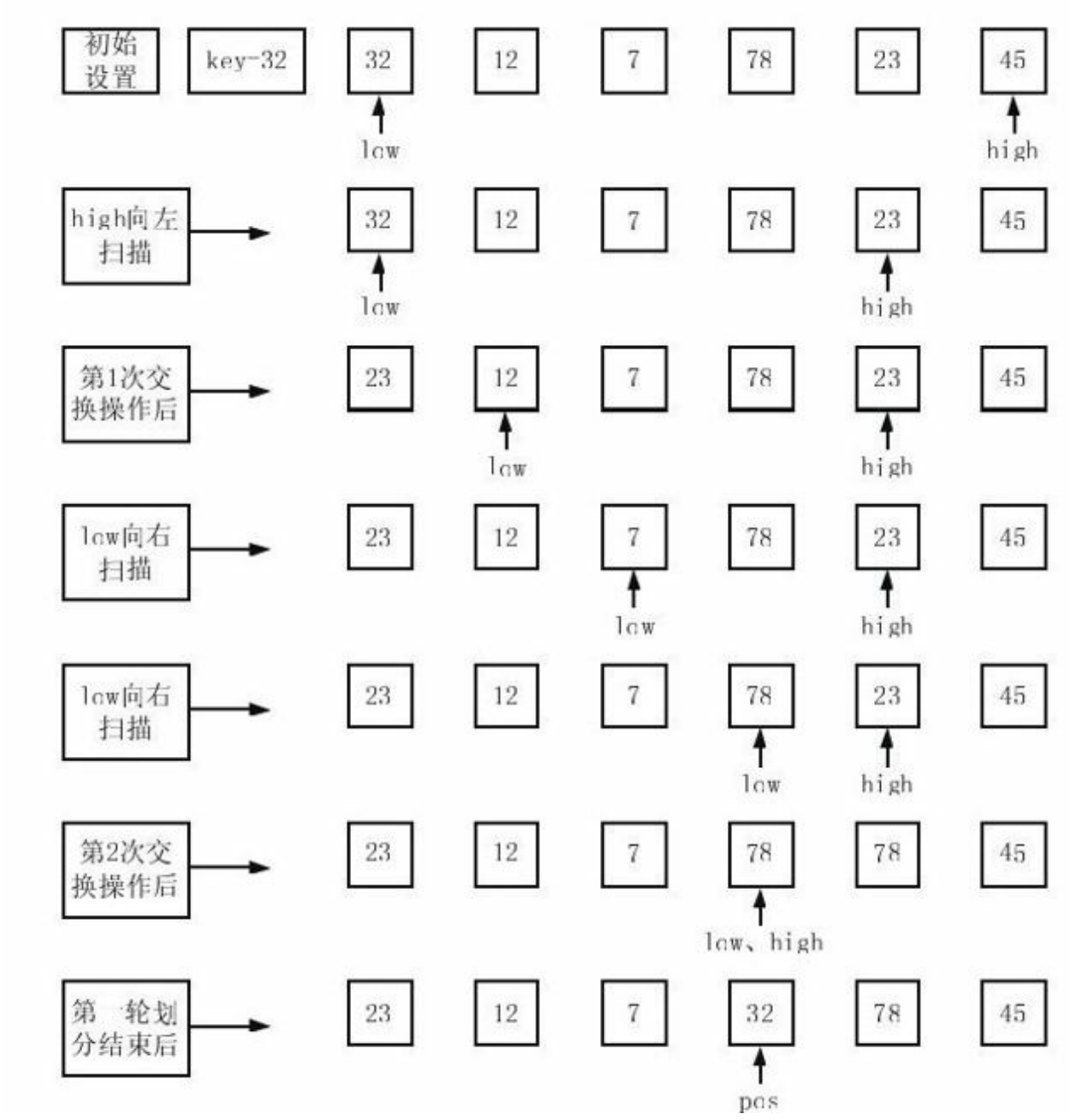

图　11-8　第一次划分操作

第一轮划分结束后，得到了左右两部分序列A[0]、A[1]、A[2]和A[4]、A[5]，继续进行划分，即对每轮划分后得到的两部分序列继续划分，直至得到有序序列为止。

11.5 归并排序

归并排序也称合并排序，其算法思想是将待排序序列分为两部分，依次对分得的两个部分再次使用归并排序，之后再对其进行合并。仅从算法思想上了解归并排序会觉得很抽象，接下来就以对序列A[0]，A[1]……，A[n-1]进行升序排列来进行讲解，在此采用自顶向下的实现方法，操作步骤如下。

（1）将所要进行的排序序列分为左右两个部分，如果要进行排序的序列的起始元素下标为first，最后一个元素的下标为last，那么左右两部分之间的临界点下标mid=（first+last）/2。这两部分分别是A[first……mid]和A[mid+1……last]。

（2）将上面所分得的两部分序列继续按照步骤（1）继续进行划分，直到划分的区间长度为1。

（3）将划分结束后的序列进行归并排序，排序方法为对所分的n个子序列进行两两合并，得到n/2或n/2+1个含有两个元素的子序列，再对得到的子序列进行合并，直至得到一个长度为n的有序序列为止。下面通过一段代码来看如何实现归并排序。

```
#include<stdio.h>
#include<stdlib.h>
#define N 7
void merge(int arr[], int low,int mid,int high)
```

```
    {
    int i,k;
    int*tmp=（int*）malloc（（high-low+1）*sizeof（int））；
    //申请空间，使其大小为两个
    int left_low=low;
    int left_high=mid;
    int right_low=mid+1;
    int right_high=high;
    for（k=0；left_low＜=left_high＆＆right_low＜=right_high；k++）//比较
    两个指针所指向的元素
    {
    if（arr[left_low]＜=arr[right_low]）
    {
    tmp[k]=arr[left_low++];
    }
    else
    {
    tmp[k]=arr[right_low++];
    }
    }
    if（left_low＜=left_high）
    //若第一个序列有剩余，直接复制出来粘到合并序列尾
    {
    //memcpy（tmp+k,arr+left_low，（left_high-left_low+1）
*sizeof（int））；
    for（i=left_low；i＜=left_high；i++）
    tmp[k++]=arr[i];
    }
    if（right_low＜=right_high）
    //若第二个序列有剩余，直接复制出来粘到合并序列尾
    {
    //memcpy（tmp+k,arr+right_low，（right_high-right_low+1）
*sizeof（int））；
    for（i=right_low；i＜=right_high；i++）
    tmp[k++]=arr[i];
    }
    for（i=0；i＜high-low+1；i++）
    arr[low+i]=tmp[i];
    free（tmp）；
    return;
    }
    void merge_sort（int arr[]，unsigned int first,unsigned int last）
    {
    int mid=0;
    if（first＜last）
    {
    mid=（first+last）/2；/*注意防止溢出*/
    /*mid=first/2+last/2；*/
```

```
//mid=（first&last）+（（first^last）>>1）;
merge_sort（arr,first,mid）;
merge_sort（arr,mid+1，last）;
merge（arr,first,mid,last）;
}
return;
}
int main（）
{
int i;
int a[N]={32，12，56，78，76，45，36};
printf（"排序前\n"）;
for（i=0; i<N; i++）
printf（"%d\t"，a[i]）;
merge_sort（a，0，N-1）;
printf（"\n排序后\n"）;
for（i=0; i<N; i++）
printf（"%d\t"，a[i]）;
printf（"\n"）;
return 0;
}
```

运行结果：

```
排序前
32 12 56 78 76 45 36
排序后
12 32 36 45 56 76 78
```

分析上面的运行结果，通过归并排序成功地实现了对给定序列的排序操作。接下来通过图11-9来了解归并排序的具体操作流程。

在图11-9中，先对所要进行排序的序列进行分解，直到分为单个元素为止，然后将其进行两两合并。由于最终分解成单个元素，因此在合并的时候，将小数放在前面，大数放在后面，得到一个有序序列。接下来对两个相连的有序序列进行排序，先比较有序序列中的第一个元素，

将较小的元素放入临时数组中，接着将较小元素所在数组的下一个元素与另一个数组中的较小元素比较，同样将较小元素放入临时数组中，依次进行，直到两个数组的所有元素都放入临时数组中，最后再将临时数组的元素放入原始数组中的对应位置。

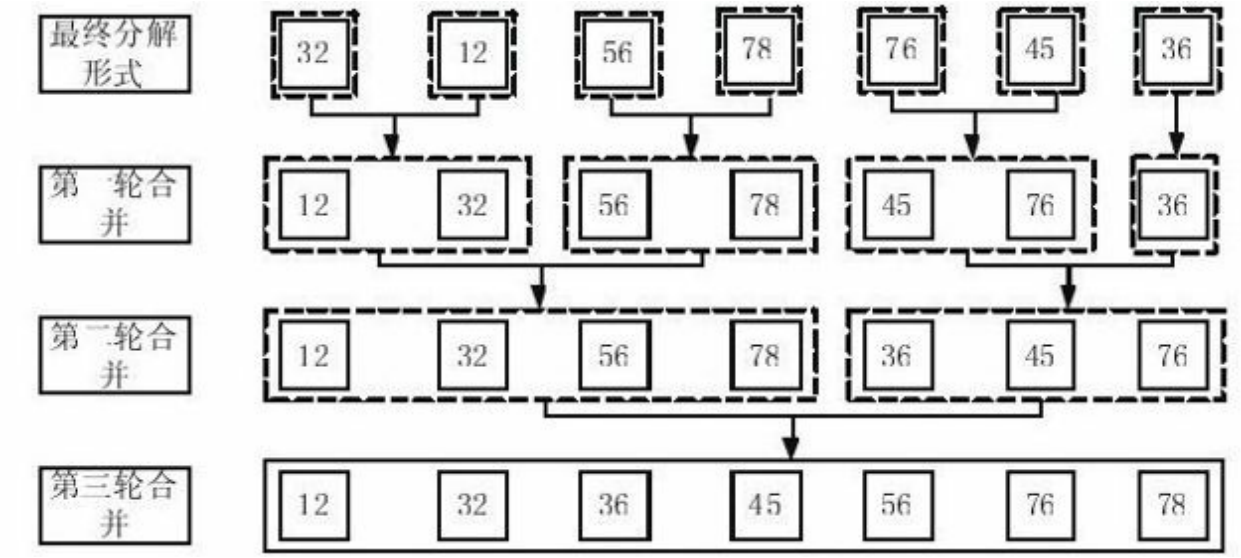

图　11-9　归并排序的操作流程

11.6 顺序查找

顺序查找是一种简单的查找算法，其实现方法是从序列的起始元素开始，逐个将序列中的元素与所要查找的元素进行比较，如果序列中有元素与所要查找的元素相等，那么查找成功，如果查找到序列的最后一个元素都不存在一个元素与所要查找的元素值相等，那么表明查找失败。接下来通过一段代码来了解顺序查找的具体使用。

```
#include<stdio.h>
#include<stdlib.h>
#include<memory.h>
int ordersearch(int a[], int n,int des)
{
int i;
for(i=0; i<n; i++)
if(des==a[i])
return 1;
return 0;
}
int main()
{
int i;
int a[8]={32, 12, 56, 78, 76, 45, 43, 98};
for(i=0; i<8; i++)
printf("%d\t", a[i]);
while(getchar())
{
printf("\n请输出所要查找的元素:");
int val;
scanf("%d", &val);
int ret=ordersearch(a, 8, val);
if(1==ret)
printf("查找成功!");
else
printf("查找失败!");
}
return 0;
}
```

运行结果：

```
32 12 56 78 76 45 43 98
请输出所要查找的元素：78
查找成功!
请输出所要查找的元素：5
查找失败!
```

分析上面的运行结果，首先输入所要查找的元素为78，该数在所要查找的序列中是存在的，所以打印输出“查找成功！”。接下来输入的数值5在所要查找的序列中并不存在，因此打印输出“查找失败！”。

11.7　二分查找

二分查找也称折半查找，其优点是查找速度快，缺点是要求所要查找的数据必须是有序序列。该算法的基本思想是将所要查找的序列的中间位置的数据与所要查找的元素进行比较，如果相等，则表示查找成功，否则将以该位置为基准将所要查找的序列分为左右两部分。接下来根据所要查找序列的升降序规律及中间元素与所查找元素的大小关系，来选择所要查找元素可能存在的那部分序列，对其采用同样的方法进行查找，直至能够确定所要查找的元素是否存在，具体的使用方法可通过下面的代码具体了解。

```
#include<stdio.h>
binarySearch (int a[], int n,int key)
{
int low=0;
int high=n-1;
while (low<=high)
{
int mid= (low+high) /2;
int midVal=a[mid];
if (midVal<key)
low=mid+1;
else if (midVal>key)
high=mid-1;
else
return mid;
}
return-1;
}
int main ()
{
int i,val,ret;
int a[8]={-32, 12, 16, 24, 36, 45, 59, 98};
for (i=0; i<8; i++)
```

```
printf（"%d\t", a[i]）;
printf（"\n请输入所要查找的元素："）;
scanf（"%d", &val）;
ret=binarySearch（a, 8, val）;
if（-1==ret）
printf（"查找失败\n"）;
else
printf（"查找成功\n"）;
return 0;
}
```

运行结果：

```
-32 12 16 24 36 45 59 98
请输入所要查找的元素：12
查找成功
```

在上面的代码中，我们成功地通过二分查找算法实现了查找功能，其实现过程如图11-10所示。

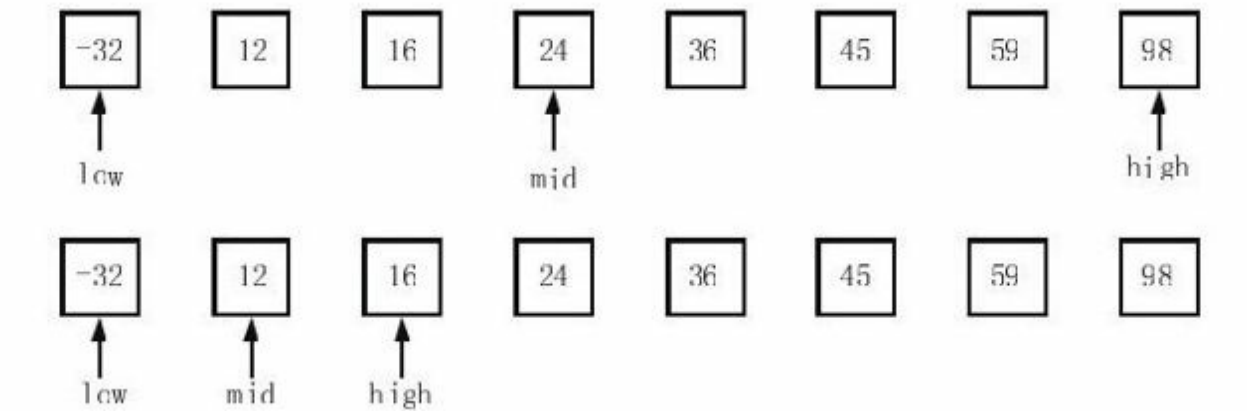

图　11-10　二分查找算法的查找过程

在如图11-10所示的查找过程中，先将序列中间位置的元素与所要查找的元素进行比较，发现要查找的元素位于该位置的左部分序列中。

接下来将mid的左边一个元素作为high，继续进行二分查找，这时mid所对应的中间元素刚好是所要查找的元素，查找结束，返回查找元素所对应的下标。在main函数中通过返回值来判断查找是否成功，如果查找成功，就打印输出“查找成功”的信息，否则输出“查找失败”的信息。

附录　如何养成良好的编程习惯

要想提高编程能力，养成良好的编程习惯是非常必要的，这是因为良好的编程习惯不仅仅使编写的代码看上去有规范的美感，还使编写的代码具有良好的易读性、扩展性和维护性等优点。那么，有哪些编程习惯是比较重要的呢？这里就简要介绍如何养成良好的编程习惯。

1.增强代码的可读性

一个优秀程序员编写的代码不仅体现在代码的质量上，而且体现在代码的可读性上。由于我们编写的代码是给计算机运行和供别人阅读的，所以在编写代码时要兼顾这两方面。

对于如何提高代码的可读性，大致可以从以下5个方面进行。

（1）格式化代码

可能有不少人对于代码的格式不屑一顾，但是杂乱无章的代码会让阅读的人心情烦乱，而那些井井有条的代码会给人一种赏心悦目的感觉。因此在编写代码的时候要遵循正规的编写规范，注意逻辑上的缩进。

（2）适当的注释

有人很少对自己编写的代码进行注释，但是我们要牢记，所编写的代码是给别人阅读的，这些人包括参与项目的其他编程人员和维护人员。要想让别人能够清楚地了解所编写代码的功能，注释是必不可少的。

（3）有含义鲜明的标识符

在代码中会涉及取变量名、函数名等，这时最好能够做到见名识意，通过名字就知道其所体现的功能。

（4）良好的逻辑性

前面讲解goto语句的时候就特别强调，要慎用goto语句，因为goto语句大大降低了代码的可读性，同时goto语句不容易控制，所以不要在代码中随便使用goto语句进行任意跳转。

（5）附加文档

有些程序员很少对所编写的代码附加说明文档，充其量写上注释，但是如果为所编写的代码加上一个附加文档，就能够让别人对代码有个快速的认识。同时，对于编写代码的人来说，在很久之后再次接触该代码时，能够通过附加文档快速对代码有个整体性认识。

2.防止内存错误

内存错误可以说是程序员最为头疼的问题，其实很多内存问题都是在编程的时候埋下的种子，往往是可以避免的。出现最多的一种内存错误就是对于指针的引用。要想尽可能地避免内存错误，我们就要养成一些良好的编程习惯。

（1）指针的初始化

对于指针，我们在定义的时候要养成将其初始化为NULL的习惯。

（2）释放申请的内存空间

经常会用到malloc等函数来进行堆内存的分配，但是经常会发生有始无终的情况，即申请的内存没有及时释放，会导致内存泄露，最终出现内存错误。

（3）文件指针的打开

很多时候，我们没有判断文件的打开操作是否成功就开始继续读写操作，这也是经常发生的内存错误之一。

（4）文件指针的关闭

文件指针的关闭相对来说不是那么严格，但是需要留意的是，在向打开的文件中写数据的时候，如果在退出之前没有关闭文件，缓冲区中的数据就没有写入文件中，进而导致数据丢失。

（5）判断是否成功分配内存

堆内存的分配并不是每次都能成功，所以在分配堆内存空间时需要注意判断分配操作是否成功，以防止接下来的操作不能有效进行。

3.提高代码的运行速度

多数的程序员在完成代码的编写后都会对代码进行多次优化，使代码的运行速度更快，提高代码的运行速度是每个程序员都追求的目标。提高代码的运行速度不仅仅体现在对算法的选择上，还有以下这些因素。

（1）学会使用内联函数

如果只有短短几行代码的函数在运行过程中被多次调用到，并且该函数也符合内联函数的规则，那么可以将其定义为内联函数，这样可以减小函数调用的时间开销。

（2）合理选用运算符

采用不同的方法可以实现相同的效果，但是需要记住，乘法和除法运算的执行时间比进行移位运算要多，所以在能够用移位替代乘除法的时候，要采用移位运算来处理。

（3）定义合适的变量

变量的类型对代码的执行时间也有很大的影响，典型的情况是图像的处理，采用单精度或者双精度时的处理时间比采用整型时的处理时间要长，若是对精度要求不高，对实时性要求较高的场合，如果可以采用short型来进行处理，可以大大节省处理时间。

（4）学会使用volatile

在前面的代码中讲解了如何使用volatile，对于同样的代码，采用volatile会使运行时间大为减少。

4.高内聚低耦合

“高内聚低耦合”对于每个程序员来说是耳熟能详的，它是软件工程中的概念，是判断设计好坏的标准，主要在面向对象的设计中用于判断类的内聚性是否高，耦合度是否低。但这并不意味着在C语言这种面向过程的程序设计中，我们可以不用关注高内聚低耦合这一设计思想。那么到底什么是高内聚低耦合呢？所谓高内聚，指的是模块中元素之间的紧密程度，而低耦合则指不同模块间的互连程度。

那么高内聚低耦合的程序设计到底有什么优势呢？如果仅从开始的过程来看，短期内，它并没有表现出什么优点，甚至会影响程序的开发进度，因为高内聚低耦合的程序对开发设计人员提出了更高的要求。高内聚低耦合的好处体现在程序持续发展的过程中，它具有更好的重用性、维护性、扩展性，可以更高效地完成维护开发，持续支持业务的发

展，而不会成为业务发展的障碍。

鉴于高内聚低耦合的以上优点，我们该如何在编程中实现高内聚低耦合呢？首先需要明白的一点就是高内聚低耦合本身就是矛盾的共存体，所以在编程的时候只能找到一个平衡点，使设计尽可能达到最佳。为了实现高内聚，就要在编程的时候尽可能地实现模块内各个元素彼此结合的紧密程度，高内聚也常被用来度量内部元素之间的紧密程度；而低耦合则要求尽可能减少模块之间的接口，使模块之间独立地工作，独立地完成其功能，模块之间接口的数量可用来衡量模块之间的依赖程度。

5.提高代码的复用率

提到如何提高代码的复用率，很多人的第一反应可能是采用泛型方法，遗憾的是，C语言并不支持该方法。提高代码的复用率一直是大多数程序员所热衷追求的，很多有经验的程序员在编程的过程中会将那些时常用到的函数模块提取出来，由于经过多次的编写，因此这些功能模块相对都比较稳定，在查错的时候也可以跳过复用的这部分代码，进而加快了对程序的调试。

当然，除了对这种模块的复用外，在编写的代码中还存在其他复用的地方。在写完代码之后发现代码中有很多相似的地方，这种情况非常常见，如何去掉这些重复的代码呢？这是在编写代码的过程中所要解决

的问题之一。提高代码的复用率就是一个消除代码中那些重复出现的代码块，将其提炼出来作为一个单独的处理方法，这样在精简代码的同时，也使代码的复用率得以提高。在最初编程的时候先按照一个最佳方案进行，之后分析其中的功能模块，看看功能模块之间是否可以进行划分，同时将那些具有相同功能的代码提炼出来作为一个新的功能模块，进而使代码的复用率得以大大提高。

6.养成重构的习惯

我们更多的时候是在面向对象的编程中听到重构一词，但是并不是说在C语言这种面向过程的编程语言中就没有重构，那么，到底什么是重构呢？所谓的重构，指的是在不改变软件现有功能的基础上，通过调整程序代码改善软件的质量和性能，使程序的设计模式和架构更趋合理，以提高软件的扩展性和维护性。

C语言虽然是面向过程的，但是我们仍然有必要在编写C语言程序时养成一种重构的习惯，从而改善程序的性能且便于日后维护。对于C语言这种面向过程的语言，我们虽然不可能完全将面向对象的方法照搬过来，但是它们的思想都是一样的。重构给人的感觉可能是重新构造，但是重构并不是对所编写代码整体性的颠覆，而是在不改变现有功能的基础之上对其进行一些修补工作，进而使代码具有良好的可扩展性和可维护性。

最初重构时需要花费一些开发时间，但是综合来看，这对开发来说绝对是值得的，因为在重构的时候我们不仅仅可以发现其中的Bug，还可以改善设计，使设计更加符合要求，进而改善程序的扩展性和维护性。

由于在开始编程的时候有些问题无法预见，可能只是从众多的设计方案中选择了一个最佳的方案，这就使选择的方案不一定就是完全符合要求的，难免需要在开发的过程中进行适当修改，因此我们可以将项目分为多个阶段来进行，在每个阶段都对现有的代码进行重构。