*第七章 快递员—— 指针*

第一节：地址与指针

第二节：指针的声明与运算

第三节：指针与函数

第四节：指针常量与常量指针

第五节：字符指针

第六节：指针与数组

第七节：指针与内存操作

第八节：函数指针

指针是C语言的灵魂，C语言的很多灵活的操作都必须借助于指针。而事实上，C语言本身就是建立在指针的基础上的，我们也早就接触过指针，比如strcat、strlen等函数，它们处理的实际上不是数组而是指针（这也就是为什么我们说，我们自己编写的strCpy函数和标准库中的strcpy拥有很大的差别）。指针的高级应用比较复杂，但是基本概念的学习却是很简单的，本章将详细讲解指针的基础知识的方方面面，并且会有一些稍微高级一点的应用。

7.1指针与地址

C语言中的变量和常量都是储存在内存中的，内存中的每个字节都有唯一的一个地址作为标识。前面的章节中我们已经了解到，只要知道一个变量的地址，就能为它赋值，同时，只要知道一个数组的地址，就能访问这个数组中的元素。我们知道，变量的地址可以用&运算符来取得，而数组的地址就是数组名。那么有没有一种数据类型可以专门用来保存地址呢？答案是肯定的，保存地址的量就是指针（**pointer**）。

指针和快递员有类似的特点。快递员通常拥有客户的地址，地址不是客户本身，但是有了客户的地址就能访问客户。同样地，指针保存的量只是变量的地址，而不是变量本身，知道了地址就能访问对应的变量（访问方法将会在7.2节被介绍）。不管客户的货物是什么，它的地址都是很有限的一段字符串。而对于变量来说，不管它是什么类型的，它的地址的长度都是相同的。

7.2指针的声明与运算

指针也是变量，一种特殊的变量，它储存的不是具体的值，而是地址。既然指针也是变量，那么使用它之前也需要声明，声明指针的格式如下：

变量类型 \*指针名;

比如：

int \*a;

char \*b;

a是一个指向int型变量的指针，而b则是指向char类型的指针。如果要同时声明多个指针，那么每个指针名前面都应该加上\*。比如：

int \*x,\*y;

如果改写成int \*x,y;的话，y就不是int型指针，而是int型变量了。

变量在声明的时候可以初始化，指针也可以在声明的时候初始化。指针的初始化有多种方式，我们通过一个程序来具体介绍。

1. #include <stdio.h>
2. int main()
3. {
4. int \*p1 = NULL;
5. int num = 10;
6. int \*p2 = p1;
7. int \*p3=&num;
8. getchar();
9. return 0;
10. }

程序7-2-1

程序7-2-1中，我们先后声明并初始化了3个int型指针和1个int型变量。可以看到，一共有3种不同的方式可以初始化指针。

第一种是把指针初始化为NULL，也就是0。我们说过，当一个变量被声明的时候，它所储存的值是未知的，对于指针也同样如此，如果我们声明了一个指针但是没有初始化它，那么它所储存的地址也是未知的，如果我们通过这个指针来访问了未知地址中所储存的值，那么很可能造成难以察觉的错误。而如果我们把一个指针初始化为NULL，那么我们在通过这个指针访问对应的值的时候，程序就会立刻报错，从而提醒我们检查程序（地址为0处的内存是不可访问的）。

第二种初始化方式则是用一个相同类型的指针来初始化，初始化之后，p2和p1的值就相同了。我们在使用p1初始化p2的时候，并没有在p1前面加\*，因为p1就是它的名字，不需要也不应该带上\*，就像我们声明一个int形变量的时候要加上int，但是实际访问这个变量的时候却不会加上int。

第三种初始化方式则是通过变量的地址来初始化。我们用&运算符来得到num的地址，并用这个地址来初始化p3。

之前我们说过，可以通过指针来访问它指向的变量。通过指针来访问变量需要用到\*，这里的\*不是表示乘法运算，而是表示取值运算，它和&运算互为逆运算。\*是已知地址取变量，&则是已知变量取地址。比如有一个int型指针p，p指向一个int型变量num，则p表示num的地址，而\*p表示num。对\*p的操作就等同于对num的操作。

1. #include <stdio.h>
2. int main()
3. {
4. int num = 10;
5. int \*p = &num;
6. printf("num的地址是：%p\n",p);
7. printf("%p储存的值是：%d",p,\*p);
8. getchar();
9. return 0;
10. }

程序7-2-2

程序的某次运行结果如下：

num的地址是：0018F984

0018F984储存的值是：10

程序7-2-2中，我们声明了一个int型变量num和一个int型指针p，p储存的是num的地址。从运行结果我们不难看出，使用\*运算符可以根据地址取出变量。由于\*p和num是等价的，所以第7行也就能正确输出num的值了。

指针储存的是变量的地址，而不管是什么类型的变量，它的地址的长度应该都是相同的。但是这个相同也是有条件限制的，那就是在同一个程序中。计算机操作系统通常分为32位和64位的，同样地，程序也分为32位和64位。64位的操作系统可以运行32位和64位的应用程序，32位的操作系统则只能运行32位的程序。而32位的程序中，一个指针的大小是32位，在内存中，一个字节（Byte）由8个位（bit）组成，32位意味着4Byte，也就是4个字节。同样地，64位就是8个字节。

**小提示**

32位和64位的操作系统的区别在于它们的寻址能力，32位系统理论上讲最多能支持232个字节，也就是4GB，实际上通常只能支持3.25GB~3.75GB。而64位系统理论上讲则能支持264个字节，实际上由于受到系统本身以及主板、CPU等硬件的影响，所以达不到这么高。在默认情况下，我们用Visual Studio编译出的程序都是32位的，如果读者的计算机的操作系统是64位的，那么也可以选择编译出64位的程序。

我们知道sizeof运算符可以求出变量或变量类型的大小，我们不妨通过一个程序来验证上面的说法。

1. #include <stdio.h>
2. int main()
3. {
4. printf("三种类型的指针的大小：\n");
5. printf("int \*:%d\n",sizeof(int \*));
6. printf("double \*:%d\n",sizeof(double \*));
7. printf("char \*:%d\n",sizeof(char \*));
8. getchar();
9. return 0;
10. }

程序7-2-3

在默认情况下，程序7-2-3的运行结果如下：

三种类型的指针的大小：

int \*:4

double \*:4

char \*:4

可以看出，不管是什么类型的指针，在32位的程序中，所占的内存都是4个字节。

如果读者的计算机的操作系统的64位的，那么还可以尝试把程序7-2-3编译成64位的应用程序。如图7-2-1，在目标复选框中把x86改成x64即可（通常情况下，x86代表32位，x64代表64位）。

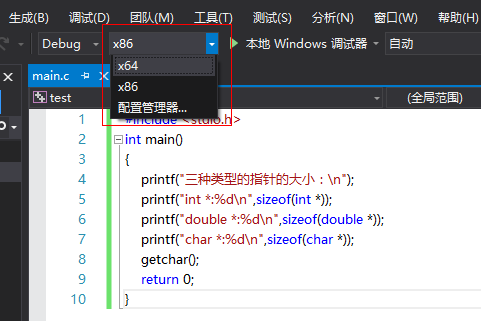


图7-2-1

如果读者的计算机能够支持64位的程序，那么将会看到和32位的程序不一样的运行结果。

既然所有的指针都一样大小，那么为什么指针还要区分不同的类型呢？这是因为指针有相应的运算，假设有一个int型指针和一个float型指针，它们指向同一个地址，但是即使如此，它们的在执行某些运算之后，运算的结果会因为它们类型不同而有所不同。

那么，除了赋值运算和取值运算，指针还有哪些运算呢？

我们先来看看指针的算术运算，一个指针可以加上或减去一个整数，得到的结果依然是一个地址。指针的算术运算的本质是指针的移动。

1. #include <stdio.h>
2. int main()
3. {
4. int num;
5. int \*p=&num;
6. printf("p的初始值：%p\n",p);
7. p++;
8. printf("p+1之后：%p",p);
9. getchar();
10. return 0;
11. }

程序7-2-4

程序的某次运行结果如下：

p的初始值：0018FEB4

p+1之后：0018FEB8

我们声明了一个int型变量num，之后声明了一个int型指针p并用num的地址来初始化了p。第6行输出p的初始值，第7行的p++;等价于p=p+1;，也就是用指针p加上一个整数。从运行结果中我们可以看出，p的初始值是0018FEB4，p+1的结果不是0018FEB5而是0018FEB8，也就是说p一次性增加了4个字节而不是1个字节，4个字节正好就是一个int型数据的大小。

我们不妨再试试用其它类型的指针来做这样的测试，看看指针类型是怎样影响到运算结果的。

1. #include <stdio.h>
2. int main()
3. {
4. int num, \*p1=&num;
5. char ch, \*p2=&ch;
6. double real, \*p3=&real;
7. printf("int:初始值：%p,+1之后:%p\n",p1,p1+1);
8. printf("char:初始值：%p,+1之后:%p\n",p2,p2+1);
9. printf("double:初始值：%p,+1之后:%p\n",p3,p3+1);
10. getchar();
11. return 0;
12. }

程序7-2-5

程序的某次运行结果如下：

int:初始值：0050F98C,+1之后:0050F990

char:初始值：0050F977,+1之后:0050F978

double:初始值：0050F958,+1之后:0050F960

可以看到，3种类型的指针+1之后与原来的值的差值分别是4、1、8。这正好是3种类型的数据所占的内存的大小。

一个指针必须有确定的类型，否则无法确定它指向的区域到底有多大，或者说无法确定它指向的区域的单位是多少。

指针除了可以和整数进行加减运算以外，还可以和另外一个指针进行减法运算。得到的结果当然就是一个整数了。

1. #include <stdio.h>
2. int main()
3. {
4. int num[10];
5. int \*p1 = &num[0];
6. int \*p2 = &num[4];
7. printf("%d",p2-p1);
8. getchar();
9. return 0;
10. }

程序7-2-6

p2-p1的结果是4。程序7-2-6中，我们首先声明了一个int型数组num，之后分别用num[0]和num[4]的地址来初始化p1和p2。我们不难看出，num[0]和num[4]之间相差了4个int型数据，因此&num[4]-&num[0]就是4，所以p2-p1的结果也就是4了。

此外，指针还能进行关系运算（>、<、==、>=、<=-、!=等）。关系运算的依据是地址的相对大小。关系运算的结果是真或者假。

1. #include <stdio.h>
2. int main()
3. {
4. int num[10];
5. int \*p1 = &num[0];
6. int \*p2 = &num[4];
7. if (p2 > p1)
8. {
9. printf("p2>p1");
10. }
11. getchar();
12. return 0;
13. }

程序7-2-7

第9行的printf语句会被执行，因为p2>p1的结果是真。

以程序7-2-7为例，用&num来初始化一个int型指针是可以的，那么用&num来初始化char指针是否合理呢？答案是可以的，只不过要经过类型转换，通常情况下，C语言允许某种类型隐式地转换成其它类型的指针，我们说过，数据类型转换可能造成数据丢失，而指针的转换是不会造成数据丢失的，因为所有的指针本质上都是相同的。

1. #include <stdio.h>
2. int main()
3. {
4. int num[10];
5. char \*p1 = &num[0],\*p2=&num[4];
6. int \*p3 = &num[0];
7. printf("int型指针：%p char型指针：%p\n",p1,p3);
8. printf("p2-p3：%d",p2-p1);
9. getchar();
10. return 0;
11. }

程序7-2-8

程序的某次运行结果如下：

int型指针：0019FA30 char型指针：0019FA30

p2-p3：16

可以看到，p1和p3的值是相等的，同时p2-p1的值不再是4而是16，这是因为p2和p1都是char型指针而不是int型。

如果你发现程序7-2-8在你的计算机上不能通过编译，请检查源文件的后缀，如果源文件后缀是.cpp，那么把它改成.c即可通过编译。

**小提示**

通常情况下，人们习惯把C和C++统称为C/C++。之所以这样称呼，不是因为C++是C语言的“升级版”，而是因为它们的语法实在太像了，以至于会C++的程序员通常也会C语言，反之亦然。C语言和C++有着本质区别，初学者应该牢记它们是不同的语言。事实上，绝大多数合法的C语言程序都是合法的C++程序，但是并非所有。程序7-2-8是一个合法的C语言程序，但它不是合法的C++程序。C++里面，不支持一种类型的指针隐式地转换到另一种类型。就程序7-2-8而言，如果源文件的后缀是.cpp，那么编译器会把它当成C++程序来编译，如果是.c，则会被当成C语言程序。这也就是为什么，当后缀为.cpp时，程序7-2-8不能通过编译。

虽然两个不同类型的指针之间发生赋值的时候，C语言允许它们进行隐式的类型转换，但是考虑到程序的可读性以及与C++的兼容性，我们也可以不用隐式转换，而是采用强制的显式转换。

程序7-2-8中的第5行可以改成这样：char \*p1=(char \*)&num[0],\*p2=(char \*)&num[4];这样就能达到显式转换的目的了，同时程序也变成了合法的C++程序。

7.3指针与函数

指针本身作为一种变量，也可以作为函数的参数。早在前面第5章，我们就讲过函数中对参数的访问不会影响到主调函数中的实参。因为主调函数和被调函数中的参数只是值相等，本质上讲是两个不同的数据。而第6章我们又说，数组作为函数参数传递的时候，被调函数中对数组的操作会影响到主调函数中的数组，因为数组名是一个地址，我们把地址作为参数传递，那么被调函数中就能访问到主调函数中的数组了。如果我们把某个变量的地址当成参数传递，那么就能在被调函数中修改主调函数中的变量的值了。

首先我们来看看如何用指针作为参数传递。

1. #include <stdio.h>
2. void displayPointer(int \*);
3. int main()
4. {
5. int num ,\*p=&num;
6. printf("主调函数中的p的值：%p\n",p);
7. displayPointer(p);
8. getchar();
9. return 0;
10. }
11. void displayPointer(int \*p)
12. {
13. printf("被调函数中的p的值：%p", p);
14. }

程序7-3-1

程序的某次运行结果如下：

主调函数中的p的值：0018F724

被调函数中的p的值：0018F724

第2行是函数的声明，声明的时候可以省略形参的名字，但是一定要给出完整的类型，int \*表示int型指针。

我们知道，实参和形参的值是相等的，所以主调函数和被调函数中的p的值也就是相等的。它们储存着同样的地址，因此通过它们可以访问同一个变量。

1. #include <stdio.h>
2. void swap(int \*, int \*);
3. int main()
4. {
5. int a = 10, b = 5;
6. printf("交换之前：a=%d,b=%d\n",a,b);
7. swap(&a, &b);
8. printf("交换之后：a=%d,b=%d", a, b);
9. getchar();
10. return 0;
11. }
12. void swap(int \*a, int \*b)
13. {
14. int temp;
15. temp = \*a;
16. \*a = \*b;

如果我们试图通过一个普通的函数来交换两个数，那么肯定是不可以的，但是如果我们把待交换的两个数的地址作为参数传递，那么交换就能成功了。

1. \*b = temp;
2. }

程序7-3-2

从运行结果可以看出，这个程序能够实现交换两个数。通常情况下，被调函数不能直接访问主调函数中的变量，而有了指针之后，我们就可以通过传递地址的方式来达到间接访问的目的。

前面第5章，我们编写了一个猜拳游戏，为了能让程序正常运行，我们不得不采用全局变量来实现，而我们又说全局变量有着诸多的缺点，不应该大量使用。现在我们有了指针作为工具，我们不妨尝试改写那个游戏，并从中体会两种不同的实现方式的异同。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <conio.h>
4. #include <time.h>
5. void playerResult(char \*,int \*,int \*);
6. void AIResult(char \*);
7. void gameBalance(char \*,char\*,int \*,int \*);
8. int main()
9. {
10. char choice;
11. char guessPlayer, guessAI;
12. int money = 500, ante;
13. while (1)
14. {
15. printf("请选择操作\n");
16. printf("==============\n");
17. printf("1.进行游戏\n");
18. printf("2.退出游戏\n");
19. printf("==============\n");
20. do
21. {
22. choice = \_getch();
23. } while ((choice<'1') || (choice>'2'));//输入结果不对则一直等待输入
24. switch (choice)
25. {
26. case '1': //进行游戏的分支，分别调用3个相关的函数
27. {
28. playerResult(&guessPlayer,&money,&ante);
29. AIResult(&guessAI);
30. gameBalance(&guessPlayer,&guessAI,&money,&ante);
31. break;
32. }
33. case '2':
34. {
35. system("cls");
36. printf("感谢使用\n");
37. \_getch();
38. exit(0);//这个函数可以直接退出程序
39. }
40. }
41. system("cls");//清空屏幕，为了更美观
42. }
43. }
44. void playerResult(char \*guessPlayer,int \*money,int \* ante)
45. {
46. printf("请出招（1.石头 2.剪刀 3.布）\n");
47. do
48. {
49. \*guessPlayer = \_getch();
50. } while ((\*guessPlayer<'1') || (\*guessPlayer>'3'));
51. \*ante = -1;//将赌注设为-1，防止上一局游戏的干扰
52. while (1)
53. {
54. rewind(stdin);
55. printf("请下注，（您当前的财产为%d）\n", \*money);
56. scanf("%d", ante);
57. if (\*ante >= 0 && \*ante <= \*money)
58. {
59. break;
60. }
61. }
62. }
63. void AIResult(char \*guessAI)
64. {
65. srand((unsigned)time(NULL));
66. int temp = rand() % 3 + 1;//产生随机数，结果在1~3之间
67. \*guessAI = temp + '0'; //将随机数转换为字符
68. }
69. void gameBalance(char \*guessPlayer,char \*guessAI,int \*money,int \*ante)
70. {
71. int result = \*guessPlayer - \*guessAI;
72. switch (result)
73. {
74. case 0:
75. {
76. printf("平局，您当前的资产为：%d", \*money);
77. \_getch();
78. break;
79. }
80. case 1://石头克剪刀，剪刀克布
81. case -2: //布克石头
82. {
83. \*money += \*ante;
84. printf("您获胜了，您当前的资产为：%d", \*money);
85. \_getch();
86. break;
87. }
88. default:
89. {
90. \*money -= \*ante;
91. printf("您输了，您当前的资产为：%d\n", \*money);
92. if (\*money <= 0)
93. {
94. printf("您破产了，请下次再来");
95. \_getch();
96. exit(0);
97. }
98. \_getch();
99. break;
100. }
101. }
102. }

程序7-3-3

比起程序5-4-4，程序7-3-3的区别在于，guessPlayer、guessAI、money、ante这几个关键的数据从全局变量变成了局部变量。每个函数也都多了很多个指针作为参数。同时，在被调函数中，money等参数不再是普通变量，而是一个指针，因此通过它们来访问原有的数据时需要加上取值运算符\*。

看起来程序7-3-3不仅不比程序5-4-4简洁，反而麻烦了很多，并且几个关键数据依然会一直占用内存，即使它们是局部变量。

事实上，程序5-4-4只是一个特例，通常情况下，我们不会把这些数据一个一个地当成参数传递。guessPlayer、money、ante这些数据都是玩家的某些属性，因此我们通常把它们组合在一起，形成一个结构体（第8章的内容），然后传递一个结构体作为参数即可。

我们知道，主调函数和被调函数中的参数只是值相等，实际上并不是同一个变量，因此被调函数中对参数的操作不能影响到主调函数中。而如果参数是指针，那么主调函数中的参数和被调函数中的参数就表示同一个地址，对它们进行取值运算得到的结果也就是相同的了。很多初学者会因此形成一种错误的认识，认为主调函数中的指针和被调函数中的指针是同一个指针变量。实际上，它们同样只是值相等，只不过它们的值比较特殊，是地址而不是普通的值，这才导致它们的取值运算的结果相同。我们可以通过一个程序来验证这种说法。

1. #include <stdio.h>
2. void fun(int \*);
3. int main()
4. {
5. int num = 10;
6. int \*p = &num;
7. printf("主调函数中p的地址：%p，p的值：%p\n",&p,p);
8. fun(p);
9. printf("主调函数中p的地址：%p，p的值：%p\n", &p, p);
10. getchar();
11. return 0;
12. }
13. void fun(int \*p)
14. {
15. printf("被调函数中p的地址：%p，p的值：%p\n", &p, p);
16. p++;
17. printf("被调函数中p的地址：%p，p的值：%p\n", &p, p);
18. }

程序7-3-4

程序的某次运行结果如下：

主调函数中p的地址：0018FBF4，p的值：0018FC00

被调函数中p的地址：0018FB20，p的值：0018FC00

被调函数中p的地址：0018FB20，p的值：0018FC04

主调函数中p的地址：0018FBF4，p的值：0018FC00

不管是主调函数还是被调函数中的p，它们本身都是一个变量，因此对它们进行取地址运算也是可以的。

从运行结果我们可以看出，主调函数和被调函数中的指针的地址是不同的，但是它们储存的内容却是同一个地址，也就是num的地址，因此对它们进行取值运算会的到同样的结果。第16行执行了p++的操作，可以看到，被调函数中的p的值发生了变化，但是主调函数中却没有。这足以验证，主调函数中的p和被调函数中的p不是同一个指针了。

除了可以作为函数参数以外，指针还可以作为函数的返回值，因为指针本身也算一种变量。

使用指针作为返回值非常简单，在类型后面加上\*即可。

假设我们要编写一个函数来实现对一个大小为10（或其它大小）的数组的内容进行数据的合法性检查，并且不合法的数据需要重新输入。这时候，我们就可以用不合法的数据的地址作为函数的返回值，并直接用它作为scanf函数的参数，由此来让用户重新输入合法的数据。具体的实现如下：

1. #include <stdio.h>
2. int\* checkData(int[]);
3. #define ARRAYSIZE 10
4. int main()
5. {
6. int num[ARRAYSIZE],\*p;
7. printf("请输入%d个整数，要求大小在0~10之间\n",ARRAYSIZE);
8. for (int i = 0; i < ARRAYSIZE; i++)
9. {
10. scanf("%d",&num[i]);
11. }
12. while(1)
13. {
14. p = checkData(num);
15. if (p == NULL)
16. {
17. break;
18. }
19. printf("检测到不合法的数据，请重新输入该数据\n");
20. scanf("%d",p);
21. }
22. printf("数据输入成功\n");
23. rewind(stdin);
24. getchar();
25. return 0;
26. }
27. int \* checkData(int num[])
28. {
29. for (int i = 0; i < ARRAYSIZE; i++)
30. {
31. if ((num[i] < 0) || (num[i]>10))
32. {
33. return &num[i];
34. }
35. }
36. return NULL;
37. }

程序7-3-5

第10行我们用#define指令定义了一个常量ARRAYSIZE，之后在声明数组、循环输入数组元素以及检测每个元素是否合法的时候，都用这个常量作为一个指标。这样做的好处是我们可以随时修改数组的大小。程序7-3-5中默认数组大小为10，如果我们想把数组大小改成3的话，只需要修改第3行即可。如果我们没有采用这种方式来声明常量，而是直接用10表示，那么当我们想修改数组大小的时候，需要改的地方就特别多了。当然，我们目前接触到的这些的程序都比较小，因此为了方便，我们通常都没有采用#define指令来定义常量。

第8行开始的for循环中，程序循环接受用户的输入，用户输入的可能是合法的数据，也可能是不合法的。因此，第12行用了一个while循环来检查数据的合法性，在这个循环里面，我们首先调用了checkData函数，并用p来接受这个函数的返回值，如果返回值是NULL，意味着所有数据都已经合法，这时候只需要跳出循环并提示用户输入成功即可。如果返回值不为NULL，则说明还有不合法的数据，这时候，需要提示用户重新输入该数据，在输入的时候，我们就可以直接用函数的返回值作为scanf的参数来输入新的数据了，接收了新的数据之后，进入下一次循环，直到所有数据都合法为止。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <conio.h>
4. char \*strCpy(char[], char[]);
5. int main()
6. {
7. char str1[10] = "Hello";
8. char str2[10];
9. printf("%s", strCpy(str2, str1));
10. getchar();
11. return 0;
12. }
13. char \* strCpy(char destination[], char source[])
14. {
15. if (destination == NULL || source == NULL)
16. {
17. printf("数据有误，不能正确复制，为了防止更大的错误，程序将异常退出");
18. \_getch();
19. exit(0);
20. }
21. for (int i = 0;; i++)
22. {
23. if (source[i] == 0)
24. {
25. destination[i] = 0;
26. break;
27. }
28. destination[i] = source[i];
29. }
30. return destination;
31. }

程序7-3-5只是简单地介绍了指针作为函数返回值的基本做法，可能很多读者还不理解这样做到底有什么意义。前面第6章我们曾经自己编写过strCpy函数，当时我们编写的strCpy函数的返回类型是void，我们说过，标准库中的strcpy的返回值不是void。事实上，标准的strcpy的返回值就是一个指针，一个char型的指针。接下来我们将用指针的相关知识来重新实现strCpy函数，读者可以从中体会具体的差别。

程序7-3-6

程序7-3-6和程序6-4-5很相似，只是strCpy的实现方式不同了。可以看到，程序7-3-6中的strCpy函数的返回值是一个char型的指针。所谓指针，也就是地址，数组名本身代表一个地址，因此第30行我们直接以destination数组的数组名作为返回值（事实上这个返回值等价于main函数中的str2）。

我们知道，destination和source都是地址，因此它们的值也可能为NULL，所以我们需要对它们的值进行判断，如果它们中有一个或两个为NULL，那么我们就不应该执行接下来的操作。在C语言的程序中，需要经常判断一个指针或地址是否为NULL，这种做法提高程序的稳定性和安全性。

我们说，destination和source是数组名，又说它们是地址，我们还说指针也是一个地址，那么，这是否意味着指针和数组间有某种联系呢？答案是肯定的，这也是第6节和第7节将会详细介绍的内容。

7.4指针常量与常量指针

普通的数据分常量和变量，同样地，指针也分变量和常量。有趣的是，指针中的常量分三种，指针常量、常量指针以及二者的结合体。

所谓指针常量，就是指针中的常量，它本身是常量而非变量，它的值一旦确定就不应该再被改变，但是我们仍然可以通过指针常量来修改它指向的变量。

声明指针常量的语法格式如下：

变量类型 \* const 标识符=初始值;

我们通过一个简短的程序来了解指针常量的用法。

1. #include <stdio.h>
2. int main()
3. {
4. int num = 10;
5. int \* const p = &num;
6. (\*p)++; //必须给\*p加上括号
7. printf("num的值：%d",num);
8. getchar();
9. return 0;
10. }

程序7-4-1

程序运行结果如下：

num的值：11

p是一个常量，因此我们在声明它的时候就应该初始化它，这里我们用变量num的地址作为它的初始值。

\*p表示对p进行取值运算，得到的结果当然就是num。如果第6行改写成\*p++，即不加括号，那么程序将不能通过编译。因为\*运算符和++运算符的优先级相同，但是它们的结合性是从右到左的，因此++会优先和p结合，而p++是非法的运算（p是常量），运算符的优先级问题可以参考附录3，其中有详细的解释。

我们虽然不可以修改p的值，但是却通过它来修改它的目标，因为目标是一个变量。

如果我们想要用一个指针来指向一个常量，用普通的指针是不可行的，这时候，我们就需要常量指针。

常量指针的声明方式如下：

const 数据类型 \* 标识符;

或：数据类型 const \* 标识符;

const在数据类型前或者数据类型后都是可以的，但是一定要在\*的前面。

常量指针和指针常量看起来区别很小，似乎难以辨认，其实不然。区分常量指针和指针常量的方法非常简单。我们只需要看const的位置即可，如果const在\*前面，const实际上修饰的是数据类型，表示指针的目标是一个常量；而如果const在\*后面，则const直接修饰的是指针本身，这时候指针本身是一个常量，但它指向的依然是变量。

1. #include <stdio.h>
2. int main()
3. {
4. int num1;
5. const int num2 = 10;
6. const int \*p;
7. p = &num1;//常量指针可以指向变量
8. printf("p的目标的值：%d\n",\*p);
9. p = &num2;
10. printf("p的目标的值：%d\n", \*p);
11. getchar();
12. return 0;
13. }

程序7-4-2

程序运行结果如下：

p的目标的值：5

p的目标的值：10

从程序7-4-2中可以看出，常量指针不仅可以指向常量，也同样可以指向变量（这一点不同于指针常量只能指向变量）。事实上，之所以说常量指针指向常量不是因为它真的只能指向常量，而是因为我们不能通过常量指针来修改它的目标，不管目标是常量还是变量，而不能被修改的量通常都是常量，由此才称常量指针指向常量。

常量指针本身是一个变量，虽然我们不能通过它来修改它的目标，但我们可以修改它本身。可以看到，程序7-4-2中，我们并没有对常量指针p进行初始化，并且在对它赋值之后又改变了它的值。如果我们在程序中增加形如(\*p)++的代码，程序就不能通过编译，因为p的目标被视为常量。

指针常量和常量指针是可以结合在一起的，也就是说，一个指针可以既是指针常量，又是常量指针。对于这样的指针而言，它自身的值不能改变，同时也不能通过它来改变它的目标的值。这种指针的声明方式如下：

const 数据类型 \* const 标识符;

它的使用方法和指针常量、常量指针非常类似，这里不再累述。

7.5字符指针

字符指针，也就是char型的指针，字符指针除了可以像正常指针那样被访问以外，它还具有一些特别的功能。

通常，我们储存字符串的时候，需要用到字符数组。而如果借助字符指针，我们可以不用数组就能储存一个字符串。

用字符指针来“储存”字符串的方式如下：

字符指针=字符串;

我们通过一个程序来演示这种用法。

1. #include <stdio.h>
2. int main()
3. {
4. char \*str = "string";
5. printf("str指针指向的字符串是：%s\n",str);
6. getchar();
7. return 0;
8. }

程序7-5-1

从运行结果可以看出，字符串的输出是正常的，这也就意味着str确实指向一片储存着字符串的内存空间。

要注意的是，字符串并不是储存在字符指针里面的。字符指针里面储存的只是字符串的地址。

事实上，程序7-5-1中的第4行代码可以理解成这两句代码：

char s[]=”string”;

char \*str=s;

即首先声明一个字符串数组，然后把数组的首地址赋值给str指针，这样str指针就指向”string”字符串了。

除了可以在初始化的时候指定字符指针指向的字符串以外，我们还可以在任意时刻改变字符指针指向的字符串。

1. #include <stdio.h>
2. int main()
3. {
4. char \*str = "string1";
5. printf("str指针指向的字符串是：%s\n",str);
6. str = "string2";
7. printf("str指针指向的字符串是：%s\n", str);
8. getchar();
9. return 0;
10. }

程序7-5-2

程序7-5-2中，我们首先用”string1”来初始化了str指针，然后输出了str指向的字符串。接下来我们重新指定了str的值，并重新输出了它指向的字符串。

读者可能会有疑问，”string1”到底储存在哪里的呢？我们给str重新赋值之后，”string1”的地址就再也找不到了，那么它是否依然占用着内存呢？

事实上，不管是”string1”还是”string2”，它们的值都是编译阶段就能确定的，它们存在于“常量区”里面（而不是所谓先声明数组，再赋值给字符指针），也就是说，它们的存在会一直占用内存空间。

关于字符指针，初学者经常会犯一些错误。其中一点是，用字符指针作为地址来输入字符串。

假设有这么一段代码：

char str[10];

scanf("%s",str);//正确

rewind(stdin);

char \*p = "string";

scanf("%s",p);//错误

在这段代码中，我们声明了一个字符数组，并用它来接受用户输入的字符串，这样做是合理的。之后我们又声明了一个字符指针并初始化了它，然后我们试图用字符指针作为scanf的参数来输入字符串，这样做就是错误的了。

读者可能会有疑问，凭什么给p随便赋值都是对的，而用它来输入字符串却是不正确的。这里我们需要再次看到程序7-5-2，可以看到，p的值都是在一开始就指定好的，编译阶段就可以确定p指向的字符串是什么，因此编译器把这些字符串存到了常量区里，也就是说编译器实际上是为目标字符串分配了储存空间的，所以我们才能用这种方式来“储存”字符串。而如果我们用p来输入字符串，就不可行了，因为输入的字符串在编译阶段不能确定，而p不是一个数组，它没有属于它自己的储存空间，因此没有空间可以存放输入的字符串，所以scanf(“%s”,p);这样的做法就是不合理的。如果我们用str数组的地址来初始化p指针，那么就可以通过p来输入字符串了，因为这样p就有了储存空间。

除此之外，还有一种常见的错误就是直接给字符数组赋值。我们知道，在声明一个字符数组的时候，可以直接用一个字符串来初始化它。而初始化一个数组之后，我们就不能用字符串来赋值给字符数组了。

char str[10] = "string1",\*p="string1";

p = "string2"; //正确

str = "string2";//错误

我们首先用”string1”来分别初始化了str数组和p指针，然后我们改变p的目标为”string2”，这样做是合理的。但是如果我们直接把”string2”赋值给str数组，就会出现错误了。我们说，p只是储存着字符串的地址，它本身并没有储存字符串，而数组则不同，数组是自身作为容器储存着字符串。当p的目标发生改变时（从”string1”变成”string2”时），p的值本身也发生了变化（目标的地址改变了）。p是一个变量，所以它的值可以随时改变。而str则不同，str本身表示数组的地址，因此它更像是一个指针常量（本身的值不能改变，但可以通过它来改变目标）。当然，str严格来讲不算指针常量，指针常量通常都需要占用储存空间，并且用储存空间来储存目标地址，而str本身就是数组的地址，所以它是没有储存空间的。总之，str是一个常量而不是变量，而str=”string2”这样的操作相当于把”string2”的地址赋值给str，这样当然就是不可行的了。

就目前而言，我们想要把”string2”赋值给str数组，最快捷的方法是使用strcpy函数。事实上，C语言也不能提供更为便捷的方法。

7.6指针与数组

前面第3节我们曾经说到，数组名和指针之间有些联系。指针可以指向变量，而数组作为变量的集合，指针自然也可以指向它，这一点是容易想通的。

让一个指针指向一维数组的方式有很多种，只要把数组的地址赋值给指针即可。比如这样：

int num[10],\*p;

p=num;

p=&num[0];

p=&num;

这3种方式都是可行的，第一种是直接把数组名赋值给指针p，我们知道，数组名本身就代表数组的地址。第二种方式是把num[0]的地址赋值给p，而num[0]的地址当然也就是数组num的地址。第三种方式则是直接取出num数组的地址，结果与前两种方式一致。

如果指针与一维数组的应用仅限于指针可以指向数组中的某个元素的话，那么我们根本没必要专门把指针与数组作为一节来讲。为了说明这个问题，我们来看看如果通过指针来访问数组元素。

1. #include <stdio.h>
2. int main()
3. {
4. int num[10],\*p = num;
5. for (int i = 0; i < 10; i++)
6. {
7. num[i] = i + 1;
8. }
9. for (int i = 0; i < 10; i++)
10. {
11. printf("%d ",\*(p+i));
12. }
13. getchar();
14. return 0;
15. }

程序7-6-1

程序运行结果如下：

1 2 3 4 5 6 7 8 9 10

p表示num的首地址，而p+i自然就表示num[i]的地址，\*(p+i)也就是num[i]了。这种通过指针来表示数组元素的方法是我们很容易想到的，但是却不是非常方便的方法。事实上，我们完全可以通过另外一种更简洁的方式来达到用指针访问数组元素的目的，也就是直接使用下标的方法。

前面第6章我们简单提到过访问数组元素的原理。我们说，num[i]（假设num是int型数组）表示在num数组的首地址的基础上移动i个单位的地址并取出对应的值。而\*(num+i)也表示同样的意思。也就是说，我们完全可以用\*(num+i)这样的方式来取代传统的下标（如果不嫌麻烦的话）。而反过来，我们同样可以用指针+下标的方式来访问数组元素，比如\*(p+i)可以改写成p[i]。原理我们已经很清楚了，p[i]和\*(p+i)都表示同样的意思，只要p表示一个地址就足够了，我们不必关心它到底是一个指针还是数组名。

接下来我们通过一个程序来演示这种操作。

1. #include <stdio.h>
2. int main()
3. {
4. int t = 10, num[10],\*p;
5. p = &t;
6. printf("%d\n",p[0]);
7. for (int i = 0; i < 10; i++)
8. {
9. num[i] = i + 1;
10. }
11. p = num;
12. for (int i = 0; i < 10; i++)
13. {
14. printf("%d ", p[i]);
15. }
16. getchar();
17. return 0;
18. }

程序7-6-2

程序运行结果如下：

10

1 2 3 4 5 6 7 8 9 10

程序第5行，我们把int型变量t的地址赋值给p，然后访问了p[0]，不难理解p[0]和t是等价的。接下来的第7行的for循环中，我们对num数组进行了赋值操作，num储存着1~10的自然数。然后我们让p指向num数组，这下就可以用p来访问num数组了。在第10行的for循环中，我们逐一输出了num[0]~num[9]的值。从运行结果来看，这样的操作是成功的。

指针和数组名是否是等价的呢？实际上，数组名虽然可以赋值给指针，但它本身却不是一个指针（甚至也不算指针常量），因为指针拥有一片内存区域来储存它的目标的地址，而数组名代表一个地址，它不占用内存空间。而且，数组一定是存在于栈空间里的，而指针却可以指向堆空间里的一片内存，并且把这片内存当成数组来使用（下一节将会介绍这种做法）。同时，当程序在运行过程中，会有一片内存空间专门记录每个数组的大小，然后当数组的生命周期结束时，就从数组首地址开始，回收相应的内存。

以程序7-6-1和7-6-2为例，num和&num是等价的，而p（假设p指向num数组）和&p却不等价。num本身表示数组的地址（可以理解为num是一个常量），而&num的意义则是对num数组进行取地址，两个地方的num含义不同（事实上，参考其它变量的取地址的方式，&num才应该是正确的写法，只不过为了表示方便，C语言也就规定数组名本身代表地址）。p和&p不等价更好解释，就像对于一个普通变量来说，它的值和它的地址不相等一样。

我们可以通过一个程序验证这一点。

1. #include <stdio.h>
2. int main()
3. {
4. int num[10], \*p = num;
5. printf("num:%p\n",num);
6. printf("&num:%p\n", &num);
7. printf("p:%p\n", p);
8. printf("&p:%p\n", &p);
9. getchar();
10. return 0;
11. }

程序7-6-3

程序的某次运行结果如下：

num:0018F9D8

&num:0018F9D8

p:0018F9D8

&p:0018F9CC

可以看出，num、&num、p三者的值都是相等的，而&p的值则不同。

早在前面第6章，我们就介绍过如何把数组作为函数参数传递。我们知道，指针和数组名都可以被当成数组来使用。而就我们目前的知识储备来看，形参似乎是一个数组而不是指针（因为形参中含有[]，没有\*）。那么，形参真的是一个数组名吗？稍加分析我们便可以得出否定的结论。我们知道，形参和实参在储存上是分离的，形参需要占用储存空间（在C语言里面，形参一定等同于一个局部变量，一定要占用内存，而C++和其它一些高级语言里面则未必）。所以，函数形参中的数组都是可以改写成指针形式的，而事实上，它们也应该是指针而不是数组名。

程序7-6-3已经验证了，如果p和&p等价，那么p就是数组名，否则就是指针。我们可以利用程序7-6-3的思路，编写一个程序来验证。

1. #include <stdio.h>
2. void fun(int []);
3. int main()
4. {
5. int num[10];
6. printf("num数组的地址：%p\n",num);
7. fun(num);
8. getchar();
9. return 0;
10. }
11. void fun(int num[])
12. {
13. printf("参数的值：%p\n",num);
14. printf("参数的地址：%p\n",&num);
15. }

程序7-6-4

程序的某次运行结果如下：

num数组的地址：0018FE9C

参数的值：0018FE9C

参数的地址：0018FDC8

从这个运行结果我们可以看出，fun函数的参数num是一个指针而不是一个数组，只是这个指针的值是一个数组的地址而已。

**试一试**

用指针做形参，改写strLen、strCpy等函数。

指针和数组名有着紧密的联系，但是又不完全相同。而对于函数参数来讲，形参和实参不需要都是指针或者都是数组。形参是指针的时候，实参也可以是指针也可以是数组，反之亦然。接下来我们将通过一个程序来分别展示一下这4种组合。

1. #include <stdio.h>
2. void fun1(int \*);
3. void fun2(int[]);
4. int main()
5. {
6. int num1[10], num2[10], \*p = num2;
7. fun1(num1);
8. fun2(num1);
9. fun1(p);
10. fun2(p);
11. getchar();
12. return 0;
13. }
14. void fun1(int \*num)
15. {
16. printf("fun1函数，参数的值是：%p\n",num);
17. }
18. void fun2(int num[])
19. {
20. printf("fun2函数，参数的值是：%p\n", num);
21. }

程序7-6-5

程序的某次运行结果如下：

fun1函数，参数的值是：0018F780

fun2函数，参数的值是：0018F780

fun1函数，参数的值是：0018F750

fun2函数，参数的值是：0018F750

fun1和fun2两个函数分别以指针和数组作为形参，我们用指针p和数组num1分别作为fun1和fun2的参数来调用这两个函数，一共有4次调用。可以看到，形参是指针实参是数组、形参是数组实参是数组、形参是指针实参是指针、形参是指针实参是数组这4种不同的搭配都是合理的，程序都能够正常运行。

不知道读者是否考虑过，如果要指向一个二维数组，应该用什么样的指针呢？显然，用普通的指针是不够的。因为二维数组有两个下标，而普通的指针只能支持一个下标。这时候，我们需要用到行指针。

声明行指针的语法格式如下：

变量类型 (\*指针名)[表达式];

其中，表达式表示行指针的目标二维数组的第二维的大小。比如这样声明了一个二维数组：int num[2][3]，那么指向它的行指针的声明方式就是int (\*p)[3];。

1. #include <stdio.h>
2. int main()
3. {
4. int num[2][3] = {1,2,3,4,5,6};
5. int(\*p)[3] = num;
6. for (int i = 0; i < 2; i++)
7. {
8. for (int j = 0; j < 3; j++)
9. {
10. printf("%d ",p[i][j]);
11. }
12. printf("\n");
13. }
14. getchar();
15. return 0;
16. }

程序7-6-6

程序的运行结果如下：

1 2 3

4 5 6

可以看出，p能够被当成二维数组来使用，就像普通的指针能够被当成数组一样（如果这个普通的指针指向一个数组的话）。

我们说过，形如p[n]这样的表达式可以被理解成\*(p+n)，那么p[m][n]又应该怎样理解呢？

要解决这个问题，首先我们应该知道，对一个行指针进行取值运算（解除引用）的结果是什么。

事实上，行指针解除引用之后的就是一个普通的指针。前面第6章我们说过，对于一个二维数组p[m][n]来说，我们可以认为数组p是由m个长度为n的一维数组组成的。而p[m]就表示第m个一维数组的首地址，p[m]可以改写成\*(p+m)，因此不难理解为什么行指针解除引用之后的结果是一个普通的指针。

1. #include <stdio.h>
2. int main()
3. {
4. int num[2][3],(\*p)[3]=num;
5. for (int i = 0; i < 2; i++)
6. {
7. printf("%p %p\n", num[i], \*(p + i));
8. }
9. getchar();
10. return 0;
11. }

程序7-6-7

程序的某运行结果如下：

0018F9B8 0018F9B8

0018F9C4 0018F9C4

可以看出，\*(p+i)和num[i]是等价的，都表示num数组中的第i个一维数组的首地址。

既然知道了行指针解除引用的结果是一个普通的指针，那么我们就不难理解p[m][n]了，它可以被改写成\*(p[m]+n)，也就是第m个一维数组中的第n个元素，p[m]可以理解成一个普通的指针，对普通的指针进行取值运算当然是合理的。同样地，p[m][n]还可以改写成\*(\*(p+m)+n)，只不过这样写显然太麻烦了。我们同样可以通过一个程序来演示这一说法。

1. #include <stdio.h>
2. int main()
3. {
4. int num[2][3] = { 1,2,3,4,5,6 };
5. int(\*p)[3] = num;
6. for (int i = 0; i < 2; i++)
7. {
8. for (int j = 0; j < 3; j++)
9. {
10. printf("%d %d %d\n",num[i][j],p[i][j],\*(\*(p+i)+j));
11. }
12. }
13. getchar();
14. return 0;
15. }

程序7-6-8

程序的运行结果如下：

1 1 1

2 2 2

3 3 3

4 4 4

5 5 5

6 6 6

从程序7-6-8中可以看出，访问数组中的元素可以有多种方式，直接通过数组名+下标的方式、通过行指针+下标的方式、对行指针解除引用的方式。由此可见指针的操作是非常灵活的。

我们曾说过，二维数组实际上也是按照一维数组的方式来储存的。以程序7-6-8为例，num[0][3]实际上和num[1][0]是等价的，而num[0][3]这种访问方式实际上是把二维数组num当成一维数组来处理了，这当然是可行的。但是这样会有两个下标，非常不方便。那么，有没有办法让一个普通指针指向二维数组，然后把这个指针当成一维数组来使用，从而间接地达到把二维数组转换成一维数组的目的呢？答案是肯定的。

1. #include <stdio.h>
2. int main()
3. {
4. int num[2][3] = { 1,2,3,4,5,6 };
5. int \*p = (int \*)num; //将num转换成一个普通指针
6. for (int i = 0; i < 6; i++)
7. {
8. printf("%d ",p[i]);
9. }
10. getchar();
11. return 0;
12. }

程序7-6-9

程序的运行结果如下：

1 2 3 4 5 6

程序第5行就是问题的关键，可以看到，我们用到了类型转换，num本应该是一个行指针，但是我们把它转换成了一个普通的指针。我们说过，任何指针的本质都是相同的，所以这个转换的过程不会导致任何数据丢失。

**小提示**

在类型转换的过程中，数据本身并没有发生改变。就程序7-6-9而言，(int \*)num这样的转换不会使num从一个数组名变成一个普通的指针，转换的过程的实质是转换num的值。首先把num的值取到内存中，此时num的值是一个行指针，接着把num的值转换成一个普通指针，并且把转换后的结果赋值给p指针。整个过程中，num没有发生变化。

我们说过，在函数的形参中，一维数组的本质是指针，那么以二维数组为形参的函数的形参的本质是否也是指针呢？我们同样可以通过一个程序来得到问题的答案。和一维数组类似地，假如形参的值和它的地址一样，那么形参就是一个数组名，否则它就是一个指针。

我们可以提前做一点分析，我们说过，C语言中的形参本身是要占用内存的，而数组名则不会占用内存，所以我们先假设形参的本质是指针。

1. #include <stdio.h>
2. void fun(int[][3]);
3. int main()
4. {
5. int num[2][3];
6. printf("num的值：%p，num的地址：%p\n", num, &num);
7. fun(num);
8. getchar();
9. return 0;
10. }
11. void fun(int num[][3])
12. {
13. printf("参数的值：%p，参数的地址：%p\n",num,&num);
14. }

程序7-6-10

程序的某次运行结果如下：

num的值：0018FB30，num的地址：0018FB30

参数的值：0018FB30，参数的地址：0018FA5C

运行结果验证了我们之前的猜想。

和一维数组一样，参数含有二维数组的函数也能实现4种组合（即形参是指针实参是数组、形参是数组实参是数组、形参是指针实参是指针、形参是指针实参是数组）。我们同样通过一个简单的程序来演示这4种情况。

1. #include <stdio.h>
2. void fun1(int (\*)[3]);
3. void fun2(int[][3]);
4. int main()
5. {
6. int num1[2][3], num2[2][3], (\*p)[3] = num2;
7. fun1(num1);
8. fun2(num1);
9. fun1(p);
10. fun2(p);
11. getchar();
12. return 0;
13. }
14. void fun1(int (\*num)[3])
15. {
16. printf("fun1函数，参数的值是：%p\n", num);
17. }
18. void fun2(int num[][3])
19. {
20. printf("fun2函数，参数的值是：%p\n", num);
21. }

程序7-6-11

程序的某次运行结果如下：

fun1函数，参数的值是：0018FE1C

fun2函数，参数的值是：0018FE1C

fun1函数，参数的值是：0018FDFC

fun2函数，参数的值是：0018FDFC

可以看到，二维数组和一维数组是相同的。

指针本身也是一种变量，变量的集合就是数组。同样地，指针也有数组，指针数组的声明方式如下;

变量类型 \*数组名[表达式];

可以看到，指针数组的声明方式和行指针非常类似，它们的区别在于是否有圆括号。我们不妨这样理解指针数组和行指针的区别：int \*和int一样（或者其它数据类型），是一种数据类型，如果没有圆括号，则\*和int结合，表示这是一个int \*型变量的数组。而如果有小括号，则\*直接和指针名结合，那么就不再是int \*型的指针了。

指针数组的使用和普通数组差不多，只不过每个元素都是指针。

1. #include <stdio.h>
2. int main()
3. {
4. int a, b, c;
5. int \*p[3] = {&a,&b,&c};
6. for (int i = 0; i < 3; i++)
7. {
8. \*p[i] = i + 1;
9. }
10. printf("%d %d %d ",a,b,c);
11. getchar();
12. return 0;
13. }

程序7-6-12

程序的运行结果如下：

1 2 3

在程序的第5行，我们声明了一个指针数组p，并且分别用a、b、c的地址来初始化了p中的各个元素。之后的循环语句中，我们对p[0]~p[2]分别进行取值运算（解除引用），实际上，第8行的取值运算等价于这种形式：\*(p[i])，也就是先计算下标，再取值。而我们知道p[i]可以改写成\*(p+i)，所以\*p[i]实际上也等于\*(\*(p+i))，对此只需了解即可，如果在程序设计中写出这样的代码，那么代码的可读性将会非常低。

至于为什么\*p[i]等价于\*(p[i])而不是(\*p)[i]，这和运算符的结合性有关，读者可以参考附录。

当然，指针数组中的每个元素也可以指向数组。

1. #include <stdio.h>
2. int main()
3. {
4. int a, b, c[3] = {1,2,3};
5. int \*p[3] = {&a,&b,c};
6. printf("%d",p[2][2]);
7. getchar();
8. return 0;
9. }

程序7-6-13

程序7-6-13中，我们声明了一个指针数组p，分别用变量a、b以及数组c的地址来初始化了它。有意思的是第6行，这里p突然变得像个二维数组了。明明p就不是行指针，为什么还能有这种写法呢？要理解这个问题，就一定不能拘泥于表面现象，从底层的原理看来，这种写法是完全合理的。p[2][2]可以改写成\*(p[2]+2)，而p[2]是一个指针，所以这样写就是可行的，即使p是指针数组而不是行指针。

那么，如果要把指针数组当成函数参数，应该应该怎么做？我们知道，当传递普通的一维数组的时候，传递实际上是一个指针。这是否意味着传递指针数组的时候也应该传递一个指针呢？答案是肯定的，只不过这个指针就不是普通的指针了，而是二级指针，也就是指向指针的指针。

1. #include <stdio.h>
2. void fun(int \*\*);
3. int main()
4. {
5. int a, b, c;
6. int \*p[3] = { &a,&b,&c };
7. fun(p);
8. printf("%d %d %d",a,b,c);
9. getchar();
10. return 0;
11. }
12. void fun(int \*\*p)
13. {
14. \*p[0]= 10;
15. \*p[1] = 9;
16. \*p[2] = 8;
17. }

程序7-6-14

程序的运行结果如下：

10 9 8

初学者通常会觉得难以理解程序7-6-14，事实上，二级指针已经是C语言语法基础中最难的部分了。

如果我们仔细分析程序7-6-14，那么读者将会发现，它的原理也并非多么复杂。这里同样要用到类比的分析方法。首先我们可以确定，fun函数中的p[0]和主函数中的p[0]是等价的，这就好像程序7-6-4等程序中，p[i]和num[i]等价一样，只不过程序7-6-4中的num是int型数组，而程序7-6-14中的p数组是一个int \*型数组。而p[0]本身也是一个指针（一级指针，也就是我们前面说的“普通指针”），p[0]指向主函数中的变量a，而\*p[0]自然就是a了。

为了便于理解，读者也可以把程序7-6-14的第2和第12行改成void fun(int \*[]);和void fun(int \*p[]);。

此外，读者可以思考一下，如果把fun函数中的\*p[0]改成(\*p)[0]，会有什么不同。答案是，\*p[0]和(\*p)[0]没有不同，但是\*p[1]和(\*p)[1]就完全不同了。(\*p)等价于&a，因此(\*p)[0]等价于(&a)[0]，也就等价于a。而\*p[1]是等价于b的，但是(\*p)[1]却等价于(&a)[1]，很明显(&a)和b不等价。

前面我们曾讲到过字符指针，在某些情况下，用字符指针来“储存”字符串比用字符数组更方便。

如果我们想要储存10个人的名字，而这10个人的名字有长有短。首先排除用10个不同的字符数组来储存，因为同时维护10个数组是很困难的事。所以使用一个二维字符数组是更优的解决方案，假设一个人的名字最多占用20个字符，我们就可以这样声明数组：char name[10][20];。显然，维护一个二维数组的成本远远低于同时维护10个数组。如果这10个名字随时可能发生变化，那么二维数组就肯定是最好的办法，而如果他们的名字是一开始就能确定并且不会发生更改的，那么用字符指针数组就更为合理。

1. #include <stdio.h>
2. #include <string.h>
3. int main()
4. {
5. char \*name[3] = {"张三","李四","丹尼斯·麦卡利斯泰尔·里奇"};
6. for (int i = 0; i < 3; i++)
7. {
8. printf("%d ",strlen(name[i]));
9. }
10. getchar();
11. return 0;
12. }

程序7-6-15

程序7-6-15中，我们声明并初始化了一个字符指针数组，并且用3个不同的字符串来初始初始化了它，这3个字符串的长度分别是4、4、26，同时，我们知道字符串中有个隐藏的结束符，所以储存这3个字符串一共用了5+5+27+3\*4=49个字节。如果我们用一个二维数组来储存，将会用到26\*3=78个字节。很明显用二维数组将会浪费大量的空间。

这里再次强调一点，需要注意的是，用字符指针数组来“储存”字符串仅适用于在编译阶段就能确定字符串内容，并且在程序运行过程中字符串不会发生改变的情况。如果我们试图用指针数组的某个成员作为scanf的参数来输入字符串，都会导致错误的发生。

char name1[3][20];

scanf(“%s”,name1[0]); //正确

char \*name2[3]={“字符串”,”字符串”,”字符串”};

scanf(“%s”,name2[0]); //错误

字符指针数组作为函数参数和普通的指针数组是类似的。

1. #include <stdio.h>
2. void fun(char \*\*);
3. int main()
4. {
5. char \*name[3] = {"张三","李四","丹尼斯·麦卡利斯泰尔·里奇"};
6. fun(name);
7. getchar();
8. return 0;
9. }
10. void fun(char \*\*name)
11. {
12. printf("主函数中的字符指针数组：\n");
13. for(int i = 0; i < 3; i++)
14. {
15. printf("%s\n",name[i]);
16. }
17. }

程序7-6-16

程序的运行结果如下：

主函数中的字符指针数组：

张三

李四

丹尼斯·麦卡利斯泰尔·里奇

name[i]是一个指针，它指向某个字符串，用name[i]作为printf的参数就能输出一个字符串了。

7.7指针与内存操作

没有了指针，C语言的“威力”就会减少一大半，而没有了内存操作，指针的威力又会减少一大半。

前面的章节中，我们提到过堆和栈的概念。在这一节之前，我们所处理的变量都是栈空间里面的变量。而这一节，我们将讲解如何使用堆空间中的内存。

栈空间由程序自动维护，栈空间一般比较小（空间小有助于提高访问速度），所以栈空间里一般存放一些关键的数据。试想，如果我们要处理1000个学生的某一门学科的成绩，那么我们至少需要1000个int型变量，这些变量将会占用4000字节的内存空间，把它们放在栈空间里显然是不明智的。而堆空间和栈空间则正好相反，堆空间的容量非常大，但是访问速度比栈空间更小，并且需要程序员亲自来维护。

那么，到底应该怎样使用堆空间中的内存呢？这里我们需要用到一个C语言标准库的函数：malloc（需要包含头文件stdlib.h），它是最简单的内存分配函数。

malloc函数的返回值是一个void类型的指针，在C语言里面，void类型的指针可以被隐式地转换成任何其它类型的指针。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. p = malloc(sizeof(int));
7. \*p = 10;
8. printf("p的值：%p，p的目标所储存的值：%d\n",p,\*p);
9. free(p);
10. getchar();
11. return 0;
12. }

程序7-7-1

程序的某次运行结果如下：

p的值：00A58F10，p的目标所储存的值：10

如果读者发现程序7-7-1在自己编译器上不能正常地通过编译，那么读者可以检查当前源文件的后缀是否为.cpp，如果是，改成.c即可（这是C++不能兼容C语言的一个地方）。或者把第6行改成：p=(int \*)malloc(sizeof(int));。我们说，C语言中，void类型的指针可以隐式转换成其它类型的指针，因此我们可以直接把malloc的返回值赋值给int型指针p，程序运行过程中会自动发生隐式转换，把void \*转换成int \*。而C++则没有这一特性，因此我们必须采用显示转换，在malloc前加(int \*)， 强制地将malloc的返回值转换为int \*。

在第6行，我们通过malloc申请了一片堆空间中的内存，这片内存的大小是sizeof(int)，也就是int型数据所占的内存空间的大小。可以看出，malloc只有一个参数，这个参数就表示要申请的内存的大小（单位为Byte，也就是字节）。我们用p指向申请出来的内存空间，并且通过p来操作这片空间，我们把这片内存当成一个int型的数据来使用，将整数10赋值给它，并用printf来输出它。

最后第9行的free是释放内存的函数，我们说，堆空间中的内存是由程序员自己来维护的，我们负责申请了内存，也应该在这片内存不再使用时释放它，就像程序自动维护栈空间中的内存一样。free函数只有一个参数，这个参数就是要释放的内存的起始地址。

**小提示**

内存的申请与释放永远是C语言程序员头疼的问题。堆内存被申请出来后不会自动释放，如果我们申请了一片内存，那当我们不再使用这片内存时，应该释放掉它，否则这片内存就会一直占用，不能被重新分配，也不能被我们使用，相当于少了一块可用的内存空间，这种情况被称为内存泄漏。通常情况下，当一个程序运行完毕的时候，操作系统会自动回收程序“泄漏”出来的内存，但是我们不能保证所有操作系统都是如此，甚至有时候C语言程序运行在没有操作系统的环境中，更何况，有些程序会运行很长一段时间（例如服务器上的程序），所以我们不应该指望操作系统来为我们收拾烂摊子。很多高级语言都提供自动回收堆内存的机制，这种机制当然有助于程序开发，但是会浪费大量的性能，而C语言主要应用于需要节约性能的地方，所以不提供垃圾回收的功能。

程序7-7-1中的动态内存操作只能算一个“动态变量”，动态变量确实有很广泛的应用，只不过那只针对于复合类型（后面章节中的结构体），程序7-7-1中的基本数据类型的动态变量是没有什么利用价值的。

就基本数据类型而言，动态数组比动态变量使用更加广泛。我们知道，数组通常会占用较多的内存空间，如果能把它们从栈空间转移到堆空间，那么将会有效地节约栈空间中有限的资源，让这些资源被用于更需要的地方。接下来我们通过一个程序来演示动态数组的用法。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p,size=10;
6. p = malloc(size\*sizeof(int));
7. for (int i = 0; i < size; i++)
8. {
9. p[i] = i + 1;
10. printf("%d ",p[i]);
11. }
12. free(p);
13. getchar();
14. return 0;
15. }

程序7-7-2

程序的第6行，我们使用malloc为p分配了size个int型变量的空间。size是一个变量，也就是说，动态数组的大小不必在编译阶段就确定，而是可以在程序运行时决定。事实上，普通的数组也可以这样，只不过不是所有的编译器都支持C语言的这一特性，在前面的章节中我们就已经提到过这一点，当时我们还说，这一特性的作用并不是特别大，因为我们有取代它的更好的办法，动态数组就是取代它的办法。

第12行的free用来释放分配给p的内存空间。细心的读者可能会问了，不管p是像程序7-7-1那样指向一个int型变量大小的区域还是像程序7-7-2那样指向更大的一片区域，我们都用free来释放对应的内存空间，而free只有一个参数，那么free函数怎么知道要释放多大的内存空间呢？

事实上，malloc分配内存的时候，实际分配的大小比我们申请的更大一些。在这些“多出”的内存空间中，就储存有我们申请的大小。p指向的地址是malloc分配给用户的可用的内存空间，而在p之前紧邻的16个字节中，就储存有这片内存空间的各种信息。我们知道，32位的程序最多能处理4GB的内存，而4GB用一个int型变量就能表示，所以我们可以确定，在p之前的16个字节中，只有其中4个字节储存着内存空间的大小。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p,\*q,size=10;
6. p = malloc(size\*sizeof(int));
7. q= malloc(2\*size\*sizeof(int));
8. printf("%d %d %d %d\n",\*(p-4),\*(p-3),\*(p-2),\*(p-1));
9. printf("%d %d %d %d\n", \*(q - 4), \*(q - 3), \*(q - 2), \*(q - 1));
10. getchar();
11. return 0;
12. }

程序7-7-3

程序在Visual Studio 2015的环境下的运行结果如下：

1 40 65 -33686019

1 80 66 -33686019

我们为p分配了40字节的内存，为q分配了80字节。可以看到，\*(p-3)和\*(q-3)的值就是分配到的内存空间的大小，而剩下的3个数据则有其它意义，具体有何用处与编译器本身有关，这也就是为什么，我们特别强调运行结果是在Visual Studio 2015下的。即使是在Visual Studio 2015的“前辈”，Visual Studio 2013上，程序的结果都会有所不同。虽然malloc函数的具体实现多种多样，但是它们的原理都是类似的。同时，对于同一个编译器来讲，它们的malloc和free函数都是配对的，所以这个差距不会对程序员造成影响。

使用free函数时要特别注意，不能对同一片区域进行两次释放操作，原因其实很简单，任何一片从堆空间里分配出来的内存空间的前面的16个字节都会用来储存这片内存空间的各种信息，而调用free会“抹除”这些信息，这样，再次调用free的时候，就会因为不能解析出相关的信息而出错。

这么说读者可能不会有什么深刻的理解，我们不妨写一个程序来做实验。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. p = malloc(sizeof(int));
7. printf("释放内存前：%d %d %d %d\n",\*(p-4),\*(p-3),\*(p-2),\*(p-1));
8. free(p);
9. printf("释放内存后：%d %d %d %d\n", \*(p - 4), \*(p - 3), \*(p - 2), \*(p - 1));
10. getchar();
11. return 0;
12. }

程序7-7-4

在Visual Studio 2015的环境下，程序的运行结果如下：

释放内存前：1 4 65 -33686019

释放内存后：-572662307 -572662307 -572662307 -572662307

可以看出，在释放内存前，保存信息的16个字节中的信息很正常，而在释放内存之后，这些数据全部改变了，相关信息全部丢失。这时候，再次调用free函数，free函数将会在对这16个字节进行信息提取的时候出错。

在释放内存之后，原有的信息通常也会丢失，但是一个指针指向的堆内存被释放后，它依然会指向原来的内存空间，如果这时候我们试图通过它来访问目标内存空间，就可能导致错误的发生。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. p = malloc(sizeof(int));
7. \*p = 4;
8. printf("释放内存前：%d\n",\*p);
9. free(p);
10. printf("释放内存后：%d\n", \*p);
11. getchar();
12. return 0;
13. }

程序7-7-5

程序的运行结果如下：

释放内存前：4

释放内存后：-572662307

可以看到，我们储存的信息的确是丢失了。这里我们储存的是一个整数，整数不会导致很大的问题的发生，如果我们在这里储存的是一个指针，那么问题就大了。

**小提示**

在一个指针指向的堆内存空间被释放之后，如果它依然指向原处，那么这时候它可以被称为“迷途指针”，或者“野指针”。迷途指针指向的内存已不可用，通过它来访问目标内存会导致程序存在潜在的威胁，因此，在一个指针被free之后，我们通常会选择把它的值设为空指针NULL，这样程序就会在我们对该指针解除引用的时候崩溃（因为空指针不能进行解除引用）。虽然程序崩溃不是什么好事，但是它至少能给我们明确的提示我们到底哪里错了，相比那些看起来正常的“不动声色”的错误，程序崩溃带来的代价是相对较小的。因为前者会让我们花费大量时间去寻找错误所在。

既然一维数组可以有动态数组，那么二维数组可不可以是动态的呢？当然是可以的，二维的动态数组分为两种。我们可以通过行指针或是二级指针来创建二维动态数组。

首先来看第一种，通过行指针来创建，这种方式相对比较简单，但是不够灵活。按照之前创建一维动态数组的思路，我们很容易想到，创建二维动态数组的方式是申请对应的空间，并把这片内存空间的地址赋值给某个行指针，这样该行指针就可以被当成二维数组来使用了。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int (\*p)[3];
6. p = malloc(2 \* 3 \* sizeof(int));
7. for (int i = 0; i < 6; i++)
8. {
9. p[0][i] = i;
10. }
11. for (int i = 0; i < 2; i++)
12. {
13. for (int j = 0; j < 3; j++)
14. {
15. printf("%d ",p[i][j]);
16. }
17. }
18. free(p);
19. getchar();
20. return 0;
21. }

程序7-7-6

程序运行结果如下：

0 1 2 3 4 5

程序第6行我们为p分配了内存空间，malloc返回的类型是void \*，而p是int (\*)[3]型的，我们说过，在C语言中，void \*可以隐式地转换成其它类型的指针（在C++中必须写成(int (\*)[3])malloc(2\*3\*sizeof(int));），可以看出，p相当于是一个2\*3的int型二维数组。

第7行的for循环中，我们为数组p的元素依次赋值，需要注意的是，这里我们利用的是“越界访问”的方式，这种越界不会造成负面影响，就像普通的二维数组一样。而接下来的第11行的循环中，我们则换了一种循方式来访问数组中的元素，这次我们用到了双重循环。而最后释放内存的操作也没有什么特别的地方。

总得说来，通过行指针创建的二维数组和普通的二维数组在使用上几乎是一致的。不过通过行指针创建二维动态数组有个明显的问题，那就是必须知道第二维的大小。我们说，虽然C99已经规定，形如这样的写法是正确的：int x=2;int num[x];，但是就目前而言，很多编译器都还不支持这种写法。所以我们想要创造一个“变长数组”，通常都需要使用动态数组，而通过行指针显然不能达到二维数组的两维都可变的目的，这时候我们就只能借助二级指针了。

**小提示**

所谓变长数组（**variable-length array**，简称VLA），是指大小不必在编译阶段就能确定的数组，变长数组的实际大小可以等到运行时才确定。这就好像我们这样声明一个数组：int num[3];那么不管程序运行多少次，num的大小始终为3。而如果我们这样声明：int x;

scanf(“%d”,&x); int (\*)p[3]=malloc(x\*3\*sizeof(int));，数组p的大小就不能在编译阶段确定了，多次运行程序之后，p的实际大小也可能会有出入，虽然第二维的大小依然是固定的。

那么，通过二级指针创建动态数组的过程是怎样的呢？首先，我们要为二级指针分配内存，这样二级指针就指向一片内存空间了，这片空间用来储存一级指针（指针数组的数组名相当于二级指针），这片空间中肯定储存着不少的一级指针，而每个一级指针又可以指向一个一维数组，因此我们还要为这些一级指针分别分配内存，最终就能形成一个二维数组了。事实上，通过二级指针创建出来的已经不是传统意义上的二维数组了，因为储存空间不是连续的，但是依然是若干个一维数组的集合。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*\*p;
6. p = malloc(2\*sizeof(int \*));
7. for (int i = 0; i < 2; i++)
8. {
9. p[i] = malloc(3 \* sizeof(int));
10. }
11. for (int i = 0; i < 2; i++)
12. {
13. for (int j = 0; j < 3; j++)
14. {
15. p[i][j] =i\*3+j;
16. printf("%d ",p[i][j]);
17. }
18. printf("\n");
19. }
20. for (int i = 0; i < 2; i++)
21. {
22. free(p[i]);
23. }
24. free(p);
25. getchar();
26. return 0;
27. }

程序7-7-7

程序运行结果如下：

0 1 2

3 4 5

p是一个二级指针，我们为它分配了2个int \*型数据的空间，然后p就可以当成指针数组来使用了。指针数组p中的储存着2个int \*数据（2个int型指针），而每个int \*型数据又可以指向一个int型数组，因此接下来的第7行的for循环中，我们为指针数组p中的2个指针分别分配了一定的内存。

而为什么p可以当成二维数组来使用呢？我们说过，p[i][j]可以解析成\*(\*(p+i)+j)，也可以解析成\*(p[i]+j)，比如p[0][0]，就表示指针数组p中储存的第1个一级指针所指向的内存空间中的第1个元素。

假设有一个普通的二维数组：int num[2][3]，那么num[0][3]和num[1][0]实际上是等价的，通过行指针创建的动态二维数组中也一样。但是在程序7-7-7中，通过二级指针创建的二维动态数组中p中，p[0][3]和num[1][0]就不等价了，因为p[0]所指向的内存空间和p[1]指向的内存空间不连续。

而在释放内存的时候，二级指针显然也比行指针更麻烦。就像申请内存的时候要分两步一样，释放内存的时候同样要分两步，首先释放p[0]和p[1]指向的内存，然后再释放p指向的内存，这个顺序是不能交换的，否则将会导致错误的发生。

可以看出，比起行指针，二级指针更灵活，行和列都是可以任意指定的，但是代价是代码更加复杂。

在分配动态内存的时候，分配的空间的大小是由我们任意指定的，我们很可能将这个大小设置得很大，如果可用内存不足的话，会怎样呢？答案当然是会分配失败，而内存分配失败的标志就是malloc的返回值为NULL。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. unsigned int size =0;
6. size--; //size的大小为4G
7. printf("size的大小：%u\n",size);
8. int \*p=malloc(size);
9. if (p == NULL)
10. {
11. printf("内存分配失败");
12. }
13. else
14. {
15. printf("内存分配成功:%p\n",p);
16. }
17. getchar();
18. return 0;
19. }

程序7-7-8

程序的运行结果如下：

size的大小：4294967295

内存分配失败

第5行我们声明了一个usigned int类型的变量size。我们知道，int型变量可以同时表示负数和非负数，而usigned int则只能表示非负数（ungifned的意思是无符号）。int和usigned int型变量所占的内存都是4字节，也就是32位。由于usigned int不能表示负数，因此它能表示的最大的整数就是2的32次方，也就是4G。关于int和usigned int在内存中的具体表示，请参考第10章。

我们将size初始化为0，并且在第6行让size执行了自减运算。由于size只能表示非负数，所以size-1之后，size的值就从最小变成最大了（这是前面章节中提到过的“绕回处理”）。size能表示的最大值是4G，所以此时size的值也就变成了4G。

第7行输出了size的值，这里的格式控制字符是%u，%u用于输出usingned类型的数据，如果这里用的格式控制字符是%d的话，输出的值就是一个负数（原理参考第10章）。

我们知道，32位的程序理论上最多只能支持4GB的内存，所以我们尝试在一个程序中分配4GB的内存肯定会失败，即使机器的可用内存足够大。而内存分配失败后，malloc会返回NULL。第9行的if-else语句用来验证这一点。

现在知道了内存分配失败后，malloc的返回值是NULL，我们在分配动态内存时，应该编写额外的代码来检测是否分配成功，如果分配失败应该及时采取应对措施（退出程序等），以避免发生更大的错误。

除了malloc和free以外，C语言标准库还提供了很多内存操作函数，接下来我们将介绍一些常用的内存操作函数，它同样声明在stdlib.h中。

首先来看第一个，和malloc长得很像的calloc。calloc的函数原型如下：

void \* calloc(size\_t n,size\_t size);

size\_t是一种数据类型，通常情况下它和int是等价的。size表示要分配的内存的单位大小，而n则表示要分配多少个这样的内存单元。因此，分配的总的内存的大小就是n\*size。此外，calloc还会对内存进行初始化操作，分配的内存会被初始化为0。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. p = calloc(10, sizeof(int));
7. printf("%d %d %d %d\n",\*(p-4),\*(p-3),\*(p-2),\*(p-1));
8. for (int i = 0; i < 10; i++)
9. {
10. printf("%d ",p[i]);
11. }
12. free(p);
13. getchar();
14. return 0;
15. }

程序7-7-9

程序在Visual Studio 2015的环境下的运行结果如下：

1 40 65 -33686019

0 0 0 0 0 0 0 0 0 0

从这个运行结果，我们可以推测出一些信息。通过calloc分配出来的内存和通过malloc分配出的内存一样，都有16个“附加字节”用来储存该区域的信息，这意味着calloc和malloc分配的内存都能用free来释放。同时，所有的数据也都被初始化为0了，这说明calloc除了执行分配内存的操作外，还执行了内存初始化操作。

除了malloc和calloc，常用的内存分配函数还有realloc。realloc的作用是重新分配内存。假设我们创建了一个一维动态数组，一段时间之后发现空间不足或者空间过剩，这时候我们就需要重新调整内存。函数原型如下：

void \* realloc(void \*\_Block,size\_t \_Size);

其中，\_Block是需要重新分配内存的地址，而\_Size则是重新分配后的大小。而返回值则是重新分配内存后，新的地址。

需要注意的是，重新分配的内存的地址可能和以前一样，也可能不同。

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int \*p;
6. printf("减小分配的内存的情况：\n");
7. p= malloc(10 \* sizeof(int));
8. printf("p的值：%p\n",p);
9. p = realloc(p, 8\*sizeof(int));
10. printf("重新分配内存后，p的值：%p\n",p);
11. free(p);
12. printf("增大分配的内存的情况：\n");
13. p = malloc(10 \* sizeof(int));
14. printf("p的值：%p\n", p);
15. p = realloc(p, 18 \* sizeof(int));
16. printf("重新分配内存后，p的值：%p\n", p);
17. free(p);
18. getchar();
19. return 0;
20. }

程序7-7-10

程序的某次运行结果如下：

减小分配的内存的情况：

p的值：00429E10

重新分配内存后，p的值：00429E10

增大分配的内存的情况：

p的值：00429E10

重新分配内存后，p的值：00429E98

程序运行结果表明，重新分配内存后，地址可能与原来一样，也可能不一样。不过我们不能断定，减小分配内存的情况就一定一样，增大内存的情况就一定不一样。上面的运行结果只是一种特殊情况。

在重新分配内存后，如果地址发生了变动，那么以前的地址处的内存就会被自动回收，并且以前的数据会被拷贝到新的地址。

除了calloc和realloc以外，常用的内存操作函数还有memset和memcpy，前者用于初始化内存，后者用于复制内存中的值。

首先来看memset，memset的原型如下：

void \* memset(void \*\_Dst,int \_Val,size\_t \_Size);

其中，\_Dst是需要初始化的内存的地址，\_Val则是初始化的值，\_Size是初始化的字节的数量。返回值就是初始化之后的内存的地址。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <memory.h>
4. int main()
5. {
6. int \*p;
7. p = malloc(10\*sizeof(int));
8. memset(p,0,10\*sizeof(int));
9. for (int i = 0; i < 10; i++)
10. {
11. printf("%d ",p[i]);
12. }
13. free(p);
14. getchar();
15. return 0;
16. }

程序7-7-11

程序的运行结果如下：

0 0 0 0 0 0 0 0 0 0

需要注意的是，使用memset需要包含头文件memory.h。此外，虽然memset的第2个参数的类型是int，但是实际上，它被当成char来处理。

我们完全可以自己编写代码来实现memset函数，不过这是没有意义的，因为很多编译器都提供汇编语言版的memset函数的实现，通过这种方式实现的memset函数与用C语言实现的在原理上存在区别，因此执行效率的差别也比较大。

最后要说的就是memcpy函数，它的原型如下：

void \* memcpy(void \*\_Dst,const void \*\_Src,size\_t \_Size);

\_Dst是目标地址，\_Src是源地址，函数将会把\_Src处的数据复制到\_Dst处。需要注意的是，\_Src被const修饰，这样做的意义是防止memcpy函数中通过\_Src来改变目标区域的值（参考前面的“常量指针”）。\_Size则是需要复制的内存字节数。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <memory.h>
4. int main()
5. {
6. int \*p,\*q;
7. p = malloc(10\*sizeof(int));
8. q = malloc(10 \* sizeof(int));
9. for (int i = 0; i < 10; i++)
10. {
11. p[i] = i + 1;
12. }
13. for (int i = 0; i < 10; i++)
14. {
15. printf("%d ",p[i]);
16. }
17. printf("\n");
18. memcpy(q,p,10\*sizeof(int));
19. for (int i = 0; i < 10; i++)
20. {
21. printf("%d ", q[i]);
22. }
23. printf("\n");
24. memcpy(p,p+1, 9\*sizeof(int));
25. for (int i = 0; i < 10; i++)
26. {
27. printf("%d ", p[i]);
28. }
29. printf("\n");
30. free(p);
31. free(q);
32. getchar();
33. return 0;
34. };

程序7-7-12

程序的运行结果如下：

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

2 3 4 5 6 7 8 9 10 10

使用memcpy函数需要包含memory.h头文件

我们声明了两个int型指针p和q，并且为它们分别分配了内存。然后我们初始化了“数组”p的值。

接下来的第18行，我们把p中的值拷贝到了q中，拷贝的字节数是10\*sizeof(int)。然后我们用循环语句输出了q中各个元素的值，结果表明复制是成功的。

需要注意的是第24行，为什么说这里需要注意呢。因为我们要复制的字节数是9个int型变量的大小。而源地址和目标地址分别是p和p+1，这样就会导致一个问题就是两个区域有重合的地方，当两个区域发生重合时，就会导致问题的发生。假设某个地址x处有待复制的值，如果地址和目标地址重合的话，就可能导致x在某个时刻被拷贝新的值，而这时候x处的值还没来得及复制出去，这样就造成了数据的丢失。而memcpy的实现非常智能，函数充分考虑到了发生重合的情况，因此采用了一点特殊的手段来防止错误的发生，所以我们可以看到第24行的代码执行后，程序照常运行，并且内存拷贝的结果也符合预期。

我们同样可以自己编程实现memcpy函数，不过同样没有什么意义，因为memcpy通常也会有汇编语言实现的版本。

和memcpy类似地，memmove也能实现内存拷贝，只不过memmove不能正确地处理两个区域重合的情况。

内存操作函数非常丰富，除了我们提到的这几个以外，常见的还有memcmp等函数，这些函数的使用都是非常简单的，读者有兴趣可以查阅相关资料学习。

虽然我们自己实现的memset和memcpy没有什么利用价值，但是为了让读者加深理解，我们不妨动手实现一下这些函数。

首先来看memset的实现，为了与库函数区分开来，我们把自己编写的函数命名为memSet。

1. #include <stdio.h>
2. #include <stdlib.h>
3. void \* memSet(void \*, char, int);
4. int main()
5. {
6. char \*p = malloc(10 \* sizeof(char));
7. memSet(p, '2', 10);
8. for (int i = 0; i < 10; i++)
9. {
10. printf("%c ",p[i]);
11. }
12. free(p);
13. getchar();
14. return 0;
15. }
16. void \*memSet(void \*src, char ch, int size)
17. {
18. if (src == NULL)
19. {
20. return NULL;
21. }
22. char \*p = src;
23. for (int i = 0; i < size; i++)
24. {
25. p[i] = ch;
26. }
27. return src;
28. }

程序7-7-13

程序的运行结果如下：

2 2 2 2 2 2 2 2 2 2

memSet函数中首先我们对src进行判断，如果src为NULL则不能对它进行操作。如果src不为NULL，则把它复制给char型指针p(因为src是void型指针，不能进行取值运算)，然后就能对目标区域依次进行赋值操作了，最后把src当作函数返回值返回。

再来看memcpy函数的实现，我们很容易想到的方法是从源地址开始，逐个为目标地址赋值，但是我们说，当源区域和目标区域重合时，这种逐个复制的操作可能会导致问题的发生。那么，什么情况可能会导致问题的发生呢？比如有两个char型指针p和q，p是源地址，q是目标地址，假设p表示的地址是0001，而q表示的是0005，而我们想要复制6个字节（size为6）。首先我们会把0001处的数据复制到0005处，接着0002复制到0006处，而当我们试图把0005处的数据复制到0009处时，会发现原来0005处的数据已经被覆盖了。所以，当且仅当p<q，并且p+size>q的时候，逐个复制会导致问题的发生，那么这种情况下，我们应该怎么办呢？我们知道，当p>q的时候，逐个复制不会出现问题，所以，当p+size>q的时候，我们不妨假设p<q，也就是说，我们可以“倒过来”复制。就前边的例子而言，0001+6>0005，逐个复制会导致错误的发生，而如果我们从0007开始复制，就不会有问题了（这时候相当于p<q了）。

1. #include <stdio.h>
2. #include <stdlib.h>
3. void \* memCpy(void \*,void \*,int);
4. int main()
5. {
6. char \*p = malloc(10 \* sizeof(char));
7. char \*q = malloc(10 \* sizeof(char));
8. for (int i = 0; i < 10; i++)
9. {
10. p[i] = '0' + i;
11. }
12. memCpy(q,p,10);
13. for (int i = 0; i < 10; i++)
14. {
15. printf("%c ",q[i]);
16. }
17. printf("\n");
18. memCpy(p+1,p,9);
19. for (int i = 0; i < 10; i++)
20. {
21. printf("%c ", p[i]);
22. }
23. free(p);
24. free(q);
25. getchar();
26. return 0;
27. }
28. void \*memCpy(void \*des,void \*src, int size)
29. {
30. if (src == NULL || des == NULL)
31. {
32. return NULL;
33. }
34. char \*p = src;
35. char \*q = des;
36. if ((p> q )||( p + size <= q))
37. {
38. for (int i = 0; i < size; i++)
39. {
40. q[i] = p[i];
41. }
42. }
43. else
44. {
45. for (int i = size-1; i >=0; i--)
46. {
47. q[i] = p[i];
48. }
49. }
50. return des;
51. }

\

程序7-7-14

程序的运行结果如下：

0 1 2 3 4 5 6 7 8 9

0 0 1 2 3 4 5 6 7 8

首先依然对src和des进行判断，如果有一个为NULL，则返回NULL。如果src和des复合条件，则对它们分情况讨论，正面逐个复制，或者是倒过来逐个复制。从运行结果可以看出，这个memCpy能够适应各种情况。

7.8函数指针

接下来我们将介绍本章最后的内容，函数指针，函数指针就是指向函数的指针。我们说，一个程序占用的内存除了堆区、栈区、常量区以外，还有储存代码的区域，代码区里面就储存着每个函数的代码，所以函数也是有地址的，既然它有地址，那么我们就能用指针指向它。

函数指针的声明方式如下：

返回类型 (\*标识符)();

“函数名”和”&函数名”都可以表示函数的地址，这点和数组很像。

接下来我们通过一个程序来演示函数指针的声明、初始化以及调用。

1. #include <stdio.h>
2. void fun1(int,int);
3. void fun2(int);
4. int main()
5. {
6. void(\*p)() = fun1;
7. p(10,9);
8. p = fun2;
9. p(8);
10. getchar();
11. return 0;
12. }
13. void fun1(int a,int b)
14. {
15. printf("这里是fun1，参数是:%d %d\n",a,b);
16. }
17. void fun2(int a)
18. {
19. printf("这里是fun2，参数是:%d\n", a);
20. }

程序7-8-1

程序的运行结果如下：

这里是fun1，参数是:10 9

这里是fun2，参数是:8

如果你发现程序7-8-1不能编译成功，请检查源文件后缀是否是.c，函数指针里面，C和C++有一些不兼容的地方。

初始化函数指针和初始化普通的变量一样，把一个合理的值写在赋值运算符（=）右边即可。第6行代码也可以把fun1改写成&fun1，两种写法是等价的。在调用的时候，我们把函数指针当成一个函数来处理就行。而函数指针本身是变量，所以它的值也是可以改变的，因此第8行我们改变了p的值，让它指向了fun2，并且通过p来调用了fun2。

程序7-8-1展示了函数指针的基本用法，但是依然存在两个个疑问。第一是，声明函数指针的时候，可不可以加上参数？第二是，通过函数指针调用函数的时候，参数应该如函数指针的参数列表匹配，还是应该与函数指针的目标匹配？

事实上，声明函数指针的时候，是可以加上参数列表的，而一旦加上参数列表，调用的时候就必须按照声明的时候的参数顺序来调用。而第二个问题，在调用的时候，参数只需与函数指针的参数匹配即可，不一定要和函数指针的目标相匹配。

1. #include <stdio.h>
2. void fun1(int,int);
3. void fun2(int);
4. int main()
5. {
6. void (\*p)(int) = fun1;
7. void(\*q)() = fun1;
8. q();
9. q(12, 11);
10. p(10);
11. p(10, 9);
12. p = fun2;
13. p(8);
14. getchar();
15. return 0;
16. }
17. void fun1(int a,int b)
18. {
19. printf("这里是fun1，参数是:%d %d\n",a,b);
20. }
21. void fun2(int a)
22. {
23. printf("这里是fun2，参数是:%d\n", a);
24. }

程序7-8-2

程序的某次运行结果如下：

这里是fun1，参数是:14422091 14422091

这里是fun1，参数是:12 11

这里是fun1，参数是:10 14422091

这里是fun1，参数是:10 9

这里是fun2，参数是:8

首先我们声明了两个函数指针，p和q，然后分别用fun1来初始化了它们，p和q的区别在于p的参数列表中有一个参数，而q则没有。

接下来，我们通过q来调用了fun1，可以看到我们没有提供任何参数，调用是成功的，但是参数的值似乎变成了随机值，而9行，我们通过q来调用fun1的时候，给出了两个参数，这下就正常了。

第10行则是用p来调用fun1，这里我们只给出了一个参数，所以参数b变成了随机值。那么这个地方可不可以改写成p();，就像q();一样呢？答案是不可以，当函数指针的参数列表中给出了参数之后，通过函数指针来调用函数就必须提供参数列表中的参数，可以多，但不能少。

此外，函数指针还可以作为函数参数，用法和其它指针基本一致，我们在此不再累述。

现在我们对函数指针有了基本的了解了，但是读者可能会有疑问，函数指针到底有什么意义，有什么用处呢？这里可以举一个很简单的例子，大家都玩过RPG（**Role-playing game，角色扮演类游戏**）游戏，经典的RPG游戏中，通常会有剑士、法师、射手3个职业可供我们选择，每个职业拥有不同的技能。如果我们想写一个RPG游戏，首先我们需要创建一个变量（不妨起名为occupation）来保存职业，同样地，我们还需要为每个角色的技能编写函数，每个角色都有攻击、特殊攻击、使用道具、升级（提升属性）等技能（假设所有的职业拥有相同数量的技能并且除了技能不同其它都一样）。在进行游戏的过程中，玩家可能需要经常释放各种技能，这时候，我们就不得不通过occupation来判断了：

switch(occupation)

{

case 剑士:…..;

case 法师:…..;

case 射手: …..;

}

显然，每次释放技能都要这样判断一次，程序的运行效率就变低了，而且当我们想要添加新的职业的时候，除了需要编写新的函数，还需要改写这个switch语句。而如果我们用一系列的函数指针来表示这些技能，在玩家选择职业的时候就分别为各个函数指针赋值，这样在游戏过程中就可以直接调用，而不用每次都去判断一下，这样执行判断的次数就减少了，程序的运行效率也就随之提升了。

要编写一个有可玩性的RPG游戏需要大量的元素，因此也会需要很长的代码，并且，由于C语言本身缺乏面向对象的特性，它也不适合用于编写RPG游戏这样的需要大量面向对象思想的程序。所以，我们并不提供一个RPG游戏作为实例，而是提供一个更简单的例子，读者理解其中的原理即可。

1. #include <stdio.h>
2. #include <conio.h>
3. #include <stdlib.h>
4. #define DEEP 1
5. #define LIGHT 2
6. void cold(int);
7. void warm(int);
8. void menu(void (\*)());
9. int main()
10. {
11. char choice;
12. printf("请选择颜色模式：\n");
13. printf("1.暖色模式\n");
14. printf("2.冷色模式：\n");
15. do
16. {
17. choice = \_getch();
18. } while (choice<'1'||choice>'2');
19. void(\*p)();
20. if (choice == '1')
21. {
22. p = warm;
23. }
24. else
25. {
26. p = cold;
27. }
28. menu(p);
29. }
30. void cold(int mode)
31. {
32. if (mode == DEEP)
33. {
34. system("color 0A");
35. return;
36. }
37. system("color FA");
38. }
39. void warm(int mode)
40. {
41. if (mode == DEEP)
42. {
43. system("color 0C");
44. return;
45. }
46. system("color FC");
47. }
48. void menu(void(\*p)())
49. {
50. while (1)
51. {
52. char choice;
53. printf("请选择操作：\n");
54. printf("1.将背景设为深色\n");
55. printf("2.将背景设为浅色：\n");
56. printf("3.退出程序\n");
57. do
58. {
59. choice = \_getch();
60. } while (choice<'1' || choice>'3');
61. if (choice == '3')
62. {
63. printf("感谢使用\n");
64. exit(0);
65. }
66. p(choice - '0');
67. system("cls");
68. }
69. }

程序7-8-3

这个程序的作用是切换控制台窗口的前景色和背景色。我们设定了两种模式，暖色调和冷色调，暖色调对应的颜色是红色，而冷色调则对应绿色，假设这两种颜色可以分别作为前景色。

主函数中有个菜单选择，让用户选择冷色模式或是暖色模式，冷色模式和暖色模式分别对应一个函数，然后我们根据用户的选择来为函数指针p复制，接着就把p作为参数传递给menu菜单，在menu菜单中，我们可以设定背景色是深色或是浅色，而前景色则是之前选好的冷色或暖色，在这里，我们通过p来改变颜色的时候，没有再判断用户的选择，因为我们使用了函数指针，而如果我们没有函数指针，则每次都要判断一下。

有几个地方需要注意一下，一个是system(“color 0A”);等设置颜色的语句，color是一个命令，就像cls一样，而空格之后紧跟的0A则是执行color命令的参数，0是前景色，A是背景色，0~F一共16个数字（字母，参考16进制），分别表示16种不同的颜色，有兴趣的读者可以自已测试每个数字分别代表什么颜色。程序7-8-3中，0代表白色，A代表绿色，F则代表黑色，C表示红色。另外就是第66行，我们调用函数的时候，参数是choice-‘0’而不是choice，这是因为cold和warm的参数都是整数，而我们知道’1’和1是不一样的，它们相互转换的关键就是’0’ ，假设choice为’1’，当它减去’0’，就变成了对应的整数。

练习题

1.编写memAlloc函数，该函数的作用是分配内存并初始化，返回类型为void\*,memAlloc接受一个参数，该参数表示要分配的内存的大小，要求在分配完内存之后自动将数据设置为0，内存分配成功则返回申请到的地址，否则返回NULL，主函数中调用memAlloc函数。（内存申请和内存初始化可以调用malloc和memset函数）。

2.编写程序，要求用户输入两个1~10之间的整数r和c（可以不对数据的合法性做检查，如果需要检查可以用前面章节中编写的函数），用二级指针创建一个r\*c的二维数组（第一维为r，第二维为c），然后用从0开始的自然数分别初始化所有的数据，最后输出所有数据（输出格式不做要求）。

3.编写memAlloc函数，该函数的作用是分配内存，返回类型为void，memAlloc接受两个参数，一个是要分配的内存的大小，一个是储存申请到的地址的指针的地址（二级指针），内存申请失败时将储存地址的指针设为NULL，主函数中调用memAlloc函数。（内存申请可以调用malloc函数）提示：本题可能有一定的难度，如果没有思路请仔细查看参考答案中的分析。

4.编写memCmp函数，该函数接受3个参数，前两个参数为void型指针，分别名为p和q，第3个参数为int型，名为size。memCmp的作用是比较p和q的前size个字节，如果这size个字节全都相等，则返回0；如果出现了\*(p+n)>\*(q+n)（n为小于等于size的自然数），则返回1；如果出现了\*(p+n)<\*(q+n)（n为小于等于size的自然数），则返回-1；。提示：本题可能有一定的难度，如果没有思路请仔细查看参考答案中的分析。