*第八章 自制容器——结构体和枚举*

第一节：结构体的使用

第二节：typedef运算符

第三节：结构体与数组

第四节：结构体与函数

第五节：枚举类型

第六节：联合体

第七节：结构体与链表

前面的章节中，我们曾编写过一个猜拳的游戏，我们用了很多个变量来分别表示玩家的各个属性（剩余金钱、赌注等）。各个变量并非完全独立，它们彼此存在某种内在联系，而这样用一个个独立的变量来表示这些属性显然不利于开发和维护，那么，有没有办法让这些数据成为一个有机的整体呢？答案是肯定的，方法就是本章要讲到的结构体。

8.1结构体的使用

假设现在我们要编写一个程序来处理学生信息，一个学生通常有姓名、性别、年龄、学号、专业等信息，我们很容易想到的办法是使用字符数组表示姓名、性别、学号和专业；用整数来表示年龄。这样的话，表示一个同学的信息就需要同时使用4个不同的标识符。如果需要处理很多同学的信息，就必须采用数组，这样我们就需要维护4个数组，其中3个是二维字符数组，同时维护多个数组本身就是很麻烦的事，如果其中还有二维数组，难度就会变得更高。

采用结构体（**struct**）就能让事情变得简单，借助结构体，我们不仅不需要处理二维数组，甚至不需要处理多个数组，用一个数组就能表示所有的学生的信息。

那么，结构体到底怎么使用呢？

事实上，结构体是一种自定义的数据类型，声明结构体的语法格式如下：

struct 标识符

{

数据类型1 成员1;

数据类型2 成员2;

…

数据类型n 成员 n;

};

在声明一个结构体之后，它就可以当成一种数据类型来使用了，就像int、char等基本数据类型一样。需要注意的是，结构体的声明只是声明了一种数据类型，而没有声明一个变量。当然，别忘了最后的分号。

我们通过一个程序来演示结构体的声明和使用。

1. #include <stdio.h>
2. struct student
3. {
4. char name[20];
5. char gender[5];
6. char ID[15];
7. char major[20];
8. int age;
9. }; //声明了student结构体，但并没有声明相应的变量，student结构体本身不占内存
10. int main()
11. {
12. struct player
13. {
14. int money;
15. }; //在函数中声明的结构体，只在函数内有效，参考局部变量
16. struct student stu1, sut2; //struct student的两个变量，需要占用内存
17. struct player pla;
18. printf("结构体的声明成功\n");
19. getchar();
20. return 0;
21. }

程序8-1-1

程序8-1-1中，我们声明了两个结构体，struct student和struct player，其中struct studen声明在主函数前，而struct player则声明在主函数中。当前源文件中的所有函数都能定义struct student的变量，而只有主函数中（struct player的声明之后）能定义struct player的变量。

需要注意的是，第2到9行只是声明了struct student，而没有创建相关的变量，因此这样做是不消耗内存的。而之后我们声明了相应的变量stu1和stu2，这下stu1和stu2就分别需要占用内存了。

说到内存占用，每个结构体的变量应该占用多少内存呢？是每个成员的大小相加，还是其它，我们可以通过一个程序来观察一下。

1. #include <stdio.h>
2. struct type1
3. {
4. char name[20];
5. char ID[15];
6. };
7. struct type2
8. {
9. char name;
10. int age;
11. };
12. struct type3
13. {
14. char name[3];
15. int age;
16. char ID;
17. };
18. int main()
19. {
20. printf("struct type1的变量占用的内存是：%d\n",sizeof(struct type1));
21. printf("struct type2的变量占用的内存是：%d\n", sizeof(struct type2));
22. printf("struct type3的变量占用的内存是：%d\n", sizeof(struct type3));
23. getchar();
24. return 0;
25. }

程序8-1-2

程序的运行结果如下：

struct type1的变量占用的内存是：35

struct type2的变量占用的内存是：8

struct type3的变量占用的内存是：12

可以看到，struct type1的大小就等于它的成员大小之和，而struct type2和struct type3则不一样。那么这究竟是怎么呢？

事实上，结构体的大小和构成它的最大的基本数据类型有关，就程序8-1-2而言，构成type1的最大基本数据类型是char，char的大小是1，因为type1的每个成员占用的空间都应该是char的整数倍，所以最终的大小就是35.

而对于type2来说，最大成员是int，所以每个成员都应该占用sizeof(int)（4）的整数倍的空间，因此即使name是char类型的变量，它也需要占用4个字节的空间，因此总的空间就是8了。

至于type3，也是类似的分析方法，name、age、ID分别占用4个字节，总共就是12字节。

有的读者可能会有疑问，如果这样的话，那么以下的结构体不是会浪费很多空间吗：

struct type

{

char x1;

char x2;

…

char x10;  
 int t;

};

如果每个char都按4个字节来算，那么整个结构体就是44个字节，而其中有效的仅有14个字节，浪费了很多空间。

事实上，这种写法的结构体占用的空间只有16个字节。那么这到底怎么回事呢？其实，x1到x10这10个char型数据是连续声明的，因此它会被看作一个char [10]的数组。注意到一个细节，我们在用文字描述一个学生的信息的时候，顺序是：姓名、性别、年龄、学号、专业，而在结构体中，年龄作为唯一的int型数据，我们却故意把它写到了最后。这样做能尽可能地节约空间，所有的char型数据占用的空间的总的大小为60，正好是4的倍数，所以不再浪费多余的空间，加上age的大小，一共就是64。而如果我们按照顺序把age的声明放在最中间，那么结构体的大小就会变成68，因为这时候name和gender会占用28个字节，ID和major又会占用36个字节，总共算下来就多出了4个字节。因此一定要养成好习惯，相同的数据类型写到一起（或者说把占用空间较小的写在一起）。当然，结构体的大小和编译器本身是有关系的，而即使是同一个编译器，我们也可以通过一些预编译指令来修改这个规则，以上的规则仅适用于通常情况。

说了这么多细节上的问题，接下来我们说一说结构体的访问问题。我们已经知道，一个结构体变量中，包含有若干个成员，那么到底应该怎样访问这些成员？答案是.运算符，它被称为取成员运算符，作用就是得到结构体变量的某个成员。

以程序8-1-1为例，struct student的变量stu1和stu2都拥有name、ID等成员，那么访问stu1的name成员的方式就是stu1.name，stu2的name则是stu2.name。

接下来我们用一个程序演示这种操作。

1. #include <stdio.h>
2. struct student
3. {
4. char name[15];
5. int age;
6. };
7. int main()
8. {
9. struct student t;
10. printf("请输入一个学生的姓名和年龄：\n");
11. scanf("%s%d",t.name,&t.age);
12. printf("学生的姓名是：%s，年龄是：%d\n",t.name,t.age);
13. rewind(stdin);
14. getchar();
15. return 0;
16. }

程序8-1-3

程序的某次运行结果如下：

请输入一个学生的姓名和年龄：

**余悦 19**

学生的姓名是：余悦，年龄是：19

为了简化程序，我们省去了不必要的多余的操作，因此只保留了两个成员，name和age。可以看到，由于name本身是一个字符数组名，所以我们用.运算符把它从t中提取出来后，可以直接使用它来输入字符串，同样地，age作为int型的数据，我们可以用&来取它的地址，&和.都是运算符，只是.的优先级更高，所以&t.age等价于&(t.age)（优先级请参考附录）。

需要注意的是，上面的输入名字和年龄的方式是不对数据的合法性加以检查的方式，这样做有潜在的危险，如果如果读者想要对数据的合法性加以检查，可以使用前面章节中编写的getNum和getStr函数。

结构体变量的成员访问没有什么太大的难度，把它们当成普通的变量即可。

前面在说结构体的声明方式的时候，格式是：数据类型n 成员n。数据类型没有规定一定要是基本数据类型，事实上，完全可以是结构体。用一个结构体的变量作为另一个结构体的成员的方式称为结构体的嵌套。

显然，带有嵌套的结构体的声明很简单，读者不难想到具体的形式。问题是嵌套的结构体的大小以及访问方式。

假设我们要编写一个struct student，其中有一项是生日，我们知道生日需要用年月日3个变量来限定，而年月日正好又可以看成一个结构体——struct date。接下来编写对应的程序，来解决之前的疑问。

1. #include <stdio.h>
2. struct date
3. {
4. int year;
5. int month;
6. int day;
7. };
8. struct student
9. {
10. char name[15];
11. struct date birthday;
12. };
13. int main()
14. {
15. struct student t;
16. printf("struct student的大小是：%d\n",sizeof(struct student));
17. printf("请输入一个学生的姓名和生日（年、月、日）：\n");
18. scanf("%s%d%d%d"
19. ,t.name,&t.birthday.year, &t.birthday.month, &t.birthday.day);
20. printf("学生的姓名是：%s，生日是：%d年%d月%d日\n"
21. ,t.name,t.birthday.year,t.birthday.month,t.birthday.day);
22. rewind(stdin);
23. getchar();
24. return 0;
25. }

程序8-1-4

程序的某次运行结果如下：

struct student的大小是：28

请输入一个学生的姓名和生日（年、月、日）：

**余悦 1996 7 1**

学生的姓名是：余悦，生日是：1996年7月1日

首先，从第一行的运行结果来看，struct student的大小并不是struct date的大小（16）的整数倍，而是所有struct student的成员和所有struct date的成员中的最大的基本数据类型的大小的整数倍，而最大的是int，所以最终的结果也就是28了。

而访问方式也很简单了，逐层地用.来提取，我们知道.的结合性是从左到右的，所以解析顺序也是从左到右，这样最终就能提取出birthday中的各个成员。

任何变量都应该能初始化和赋值，结构体变量也不例外。结构体的初始化方式有两种，一种是用其它相同类型的结构体变量来初始化，另一种是用成员的集合列表来依次初始化。至于赋值运算，一个结构体变量可以直接赋值给另外一个相同类型的结构体变量。我们通过几个程序来演示这些操作。

1. #include <stdio.h>
2. struct date
3. {
4. int year;
5. int month;
6. int day;
7. };
8. int main()
9. {
10. struct date d = {2015,12,30};
11. struct date t = d;
12. printf("d的各个成员：%d %d %d\n",d.year,d.month,d.day);
13. printf("t的各个成员：%d %d %d\n", t.year, t.month, t.day);
14. rewind(stdin);
15. getchar();
16. return 0;
17. }

程序8-1-5

程序的运行结果如下：

d的各个成员：2015 12 30

t的各个成员：2015 12 30

程序8-1-5展示了两种结构体变量的初始化方式，也是两种比较常用的方式，不过在有些时候，这样却是不合理的，至于怎么不合理，在说完了结构体的赋值之后，我们会有所讲解。

就像初始化一样，我们可以直接用一个结构体变量赋值给另外一个相同类型的结构体变量，但是我们不能用列表来为一个结构体赋值，即：

struct date d,t={2015,12,30};

d=t;//正确

d={2015,12,30};//错误

此外，如果结构体中还有嵌套，那么初始化的时候，在列表内再加一个列表即可（也可以不用），类似于二维数组。

1. #include <stdio.h>
2. struct date
3. {
4. int year;
5. int month;
6. int day;
7. };
8. struct student
9. {
10. char name[20];
11. struct date birthday;
12. };
13. int main()
14. {
15. struct student t = { "余悦",{1996,7,1} };
16. struct student s = { "余悦", 1996,7,1 };
17. printf("t中的数据：%s %d %d %d\n"
18. ,t.name,t.birthday.year,t.birthday.month,t.birthday.day);
19. printf("d中的数据：%s %d %d %d\n"
20. , s.name, s.birthday.year, s.birthday.month, s.birthday.day);
21. rewind(stdin);
22. getchar();
23. return 0;
24. }

程序8-1-6

程序的运行结果如下：

t中的数据：余悦 1996 7 1

d中的数据：余悦 1996 7 1

从程序8-1-6中可以看出，我们可以加上嵌套的列表，也可以不加，因为储存空间在内存上是连续的，这一点和二维数组的初始化的原理是相同的。

接下来我们要说的就是为什么有的时候直接用一个结构体赋值给另外一个结构体会出现问题。

很明显结构体的成员中可以有指针，而前面第7章我们说过，指针可以用来申请动态内存，而动态内存需要程序员手动分配和释放。假设有这样一个结构体：

struct t

{

int \*p;

};

如果我们创建了一个struct t的变量q，并且为q.p分配了内存，那么我们肯定会采取一定措施在q不再使用时候释放掉q.p指向的空间。而如果我们同时也创建了一个struct t的变量m，并且把q直接赋值给m，那么m.p和q.p就指向同一片内存空间了，这样就会导致问题的发生，如果q不再使用，那么q.p会被释放，这时候m.p依然指向原来的位置，实际上m.p就已经成了迷途指针了，反过来也是一样的。同时，m不再使用时，我们也会尝试释放m.p的空间，而第7章我们研究过，由于m.p前16个字节的保存信息的区域已经被“抹除”，这时候再释放m.p就会出问题了。

1. #include <stdio.h>
2. #include <stdlib.h>
3. struct t
4. {
5. int \*p;
6. };
7. int main()
8. {
9. struct t q,m;
10. q.p = malloc(10 \* sizeof(int));
11. m = q;
12. printf("q.p的值：%p,m.p的值：%p\n",q.p,m.p);
13. q.p[0] = 10;
14. printf("q.p[0]的值：%d,m.p[0]的值：%d\n", q.p[0], m.p[0]);
15. free(q.p);
16. printf("释放q.p之后，m.p的值：%p\n",m.p);
17. rewind(stdin);
18. getchar();
19. return 0;
20. }

程序8-1-7

程序的某次运行结果如下：

q.p的值：0064A608,m.p的值：0064A608

q.p[0]的值：10,m.p[0]的值：10

释放q.p之后，m.p的值：0064A608

从这个运行结果可以看出，我们之前说的威胁都是真是存在的，而且还有一点就是，虽然表面上看起来，q和m分别拥有一个独立的成员p，而事实上q.p和m.p也确实是独立的，但是由于p的特殊性，所以我们实际上想访问的是p的目标而不是p本身，这样就会造成一个问题，那就是q和m“共享”一个成员，这显然是和我们的预期不符的。所以，针对成员有指针的情况，我们就应该分别为每个成员赋值，或者在直接赋值之后，再重新修改每个指针成员的值。

**小提示**

为或含有指针成员的结构体赋值（拷贝）的时候，通常都会有两种处理方式，分别称为浅拷贝和深拷贝，所谓浅拷贝，就是指用赋值运算符直接赋值，或者逐个复制成员，而不管成员是否为指针。而深拷贝则需要针对某些指针成员做更多的操作，比如重新分配内存等。

在面向对象的编程中，有一种和结构体很类似的东西，叫做类（**class**），而类有这么三个术语：声明、定义、实例化。所谓声明，就像函数声明一样，比如struct t;就声明了一个结构体struct t，这时候还不知道它有什么成员。而像本章之前的程序都列出所有的成员，我们虽然称其为声明，但是实际上已经是定义了，声明和定义被放在了一起。而所谓实例化，就类似于声明结构体变量。C语言中没有实例化这个词，但是由于声明结构体变量的方式实在和实例化太像了，本质也是相同的，所以后面的内容中我们把声明结构体变量称为实例化，而每个结构体变量我们称为实例变量。

接下来我们要说的就是前面5.6节曾经提到过的，重复包含头文件的危害。结构体的声明（也就是定义），通常我们会把它放在头文件里面，这样几个不同的源文件就能通过包含同一个头文件来使用同一个结构体了。

而我们说，C语言中的一切，声明可以声明很多次，但是定义只能有一次，也就是说，如果我们在某个头文件中定义了一个结构体，那么任何一个源文件都只能包含一次这个头文件，否则就会出现重定义的情况，要识别一个头文件是否被包含了两次看起来似乎很简单。可是，有时候，事情不总是那么简单，假设有a.h和b.h，里面分别定义了一些结构体，而b.h由于某种需要，包含了a.h，这时候，如果我们试图在一个源文件中同时包含a.h和b.h，那么这个a.h就相当于被包含了两次，a.h中的结构体就重定义了。

那么，怎样可以防止头文件被重复包含呢？通常情况下，我们需要自己动手写一些代码来实现这个功能，而Visual Studio 2015作为一款非常人性化的编译器，每当我们创建一个头文件的时候，它就会自动在头文件的第一行写上这句代码：#pragma once，我们说过，这是一个预编译指令，它的存在让该头文件只会被编译一次，所以不管我们怎么包含都不会出错了。

而我们自己手动写代码的话，有两种情况，第一种情况，如果我们使用的是微软的其它C/C++编译器，那么我们可以直接在头文件前面加上#pragma once（只有微软的C/C++支持这种写法，目前为止只有Visual Studio 2015会自动添加这行代码）。

还有一种通用的情况，利用预编译指令来实现。

1. #ifndef STUDENT
2. #define STUDENT
3. struct student
4. {
5. char name[20];
6. };
7. #endif

程序8-1-8 student.h

程序8-1-8中，我们定义了一个struct student结构体，并且添加了一些预编译指令。那么，为什么说这些指令就能实现防止重复包含呢？

首先，第1行的#ifndef和最后一行的#endif是配套的，有#ifndef或者有#if就必须要#endif（详情参考第10章），#ifndef的解释就是“如果没有定义某某宏”。如果#ifndef后面的条件成立，那么编译器将会编译#ifndef和#endif之间的代码，否则就不会编译。首先，假设我们在某个源文件里写了两次#include “student.h”，那么第一次包含的时候，编译器会判断是否已经定义了STUDENT这个宏，如果没有（通常情况下是不会有的，除非用户在#include 语句之间定义过），则编译第2到第8行的内容。第2行直接定义了一个STUDENT宏，需要注意的是，定义宏的时候可以不用定义具体的内容，即不一定非要写成#define NUM 10这样的形式，不用NUM后面的数字，也能成功定义NUM宏。之所以要定义STUDENT宏，就是为了防止重复包含，这个我们马上就会说到。在定义完STUDENT后，编译器又会编译struct student的定义。而第二次包含student.h的时候，由于STUDENT宏已经在之前一次编译的时候定义了，所以#ifndef STUDENT的条件不满足，所以第2到第8行的代码也就不会被编译了，间接地起到了防止重复包含的作用，进而防止了struct student的重定义。

防止重复包含的这两种方法其实各有各的好处，通过宏来判断的方式具有很高的可移植性，这样的代码不仅可以在非微软编译器下通过编译，更可以移植到其它系统（如果头文件中没有包含Windows系统的“特色内容”的话）。但是#pragma once也并非一无是处，首先，它的意思是，当前头文件只会被打开一次，而通过宏来判断的方式每次都会打开头文件来检查，因此前者的速度更快。再有就是#pragma once是绝对安全的，而形如#ifndef STUDENT这样的语句会使用到STUDENT这个宏，而我们不能保证用户100%不会自定义一个这样的宏。虽然第二种方法是通用的方法，但是如果保证某个头文件只会在MSVC（**Microsoft Visual C++**编译器，即VS系列）下使用，那么使用第一种情况也是合理的。

8.2 typedef运算符

尽管很多人都把C++称为C语言的加强版，但是我们已经多次强调这是两种不同的语言，前面的部分中，我们也给出了很多C++不兼容C的例子（如函数指针和malloc）。

而事实上，就连结构体，C和C++中也是不同的。主要有这样几点：第一，C++中的结构体可以直接用花括号加成员列表的方式来赋值；第二，C++中声明结构体变量的时候，可以直接用结构体名，而不用加上struct；第三，C++中的结构体不仅可以有数据作为成员，甚至还可以有函数作为成员。当然，最根本的不同在于第3点，C++是一门面向对象的语言，而我们说过，面向对象本身是在结构体的基础上发展而来的，C++则干脆把结构体也改造成了对象，这样做的确带来了很大的方便。

我们要说的重点是，难道每次声明结构体的实例变量的时候，都要用这种形式吗：struct student t;？可不可以像C++里面那样省掉struct，少做一点无意义的工作呢？答案是可以的，问题的关键就在于typedef运算符。

那么，typedef到底有什么用呢？它的作用就是给一个已有的变量类型起一个“别名”，然后我们就能通过这个别名来创建对应的类型了。

typedef int State;

State x; //State x等价于int x

如果我们在声明一个结构体struct student之后，写上这样一句代码：typdef struct student student，这样我们就能直接使用student代替struct student了。

1. #include <stdio.h>
2. struct student
3. {
4. char name[20];
5. };
6. typedef struct student student;
7. int main()
8. {
9. struct student stu1;
10. student stu2;
11. printf("stu1的大小：%d，stu2的大小：%d\n",sizeof(stu1),sizeof(stu2));
12. getchar();
13. return 0;
14. }

程序8-2-1

程序的运行结果如下：

stu1的大小：20，stu2的大小：20

可以看出，使用typedef之后，我们用struct student和用student都能正确地声明struct student的实例变量。

事实上，上面的写法可以替换成下面的写法：

typedef struct

{

char name[20];

}student;

或者

typedef struct student

{

char name[20];

}student;

可以看到，这两种写法中，有一种是没有结构体名的。

在声明结构体的时候使用typedef可以让代码更简短，但是由于我们会把结构体的声明写在头文件中，所以这样的typedef会使“别名”在所有直接或间接包含该头文件的源文件中生效，可能会造成命名冲突，因此两种写法不存在谁好谁坏。

我们说，typedef除了可以给结构体起别名，还可以用于基本数据类型，对于基本数据类型而言，使用typedef有什么用呢？

假设我们编写了这样一个函数：

void sort(int \*num,int size)

{

…//函数内容

}

这个函数的作用是为一个int型数组排序，可以想象，这个函数的作用很小，因为它只能用于对int型数组排序，如果我们想要排序double类型的，就不得不按照相同的框架重新编写代码，即使通过复制粘贴的办法，我们也得依次修改很多地方。即使排序double型数组的函数与排序int型数组的函数是如此相似，但我们却没法直接把现成的代码利用起来。

而如果我们换种方法来写：

typedef int element;

void sort(element \*num,int size)

{

…//函数内容

}

这样，当我们需要排序double型数据的时候，直接复制已经写好的代码，然后将int改成double即可。这一点类似于用#define来定义常量。

说到#define，有的读者可能会问，我们可不可以编写这样的语句来达到起别名的效果呢: #define student struct student？答案当然是可以的，只不过如果我们这样做了的话，就不能再使用struct student了，因为#define只是简单的文本替换，它会把所有的student都替换成struct student（当然，我们也可以改写成#define stu struct student来避免这种情况）。需要注意的是，typedef和#define有深层次的区别，typedef发生在编译阶段，而#define则发生在预编译阶段，使用typedef的时候，student和struct student不会冲突，而使用#define则会。

我们再来看一个程序，进一步理解两者的不同。

1. #include <stdio.h>
2. #define type double \*
3. int main()
4. {
5. type a, b;
6. printf("a的大小：%d，b的大小：%d\n",sizeof(a),sizeof(b));
7. getchar();
8. return 0;
9. }

1. getchar();
2. return 0;
3. }

程序8-2-1

程序的运行结果如下（程序为x86）：

a的大小：4，b的大小：8

c的大小：4，d的大小：4

可以看到，我们用type1来声明了两个变量a和b，但是a和b的大小却不一样，这说明了a和b的类型是不同的，那么为什么会出现这种情况？原来，type1 a,b;在预编译阶段会被解析成double \*a,b;，这样，a就是double \*类型，而b则是double类型。至于type2，由于typedef发生在编译阶段，所以type2变成了一个新的数据类型，这个数据类型就是double \*，因此c和d就都是double \*类型的了。

8.3 结构体与数组

我们知道，不仅基本数据类型可以有数组，指针类型也可以有（指针数组），而结构体作为一种数据类型，同样可以有数组。

结构体数组的声明方式和普通的数组很类似：

结构体名 标识符[表达式];

其中，表达式可以含有变量（虽然很多编译器不支持，但是C99标准明确规定含有变量是合法的）。

我们说过，在声明结构体的同时，可以用typedef来为它起别名，事实上，声明结构体的时候，同样可以将它实例化（实例化为单个实例变量或一个数组）。

1. #include <stdio.h>
2. #define NUM 3
3. struct
4. {
5. char name[20];
6. int age;
7. }stu[NUM];
8. int main()
9. {
10. for (int i = 0; i < NUM; i++)
11. {
12. printf("stu[%d]的地址：%p\n",i,&stu[i]);
13. }
14. getchar();
15. return 0;
16. }

程序8-3-1

程序的某次运行结果如下：

stu[0]的地址：008C94E0

stu[1]的地址：008C94F8

stu[2]的地址：008C9510

可以看到，我们没有给结构体专门命名，虽然这种做法欠妥，但是如果我们只需要一个该结构体的数组即可，不需要以后再实例化，那么这样做也是可行的，因为这样能省下一个标识符。

不难算出，该结构体的大小是24，而我们观察数组中的3个变量的地址，正好彼此间相差24个字节（如008C94F8=008C94E0+16+8=008C94F0+8）。事实上，程序8-3-1那样的声明结构体数组的方式很少用。

结构体数组的初始化和普通的数组很类似。

1. #include <stdio.h>
2. #define NUM 3
3. typedef struct
4. {
5. char name[20];
6. int age;
7. }student;
8. int main()
9. {
10. student stu1[NUM] = {"张三",20,"李四",19};
11. student stu2[NUM] = { {"张三"},{"李四",19 } };
12. for (int i = 0; i < NUM; i++)
13. {
14. printf("stu1[%d]：%s,%d\n",i,stu1[i].name,stu1[i].age);
15. printf("stu2[%d]：%s,%d\n", i, stu2[i].name, stu2[i].age);
16. }
17. getchar();
18. return 0;
19. }

程序8-3-2

程序的运行结果如下：

stu1[0]：张三,20

stu2[0]：张三,0

stu1[1]：李四,19

stu2[1]：李四,19

stu1[2]：,0

stu2[2]：,0

可以看到，我们声明了两个结构体数组，stu1和stu2，并且使用了不同的方式来初始化它们，第一种方式是按顺序初始化，即忽略stu[0]和stu[1]是两个不同的实例变量，而是利用了它们的在储存空间上紧邻的这一特性。而第二种初始化方式则用花括号来隔开了各个元素，同时我们在初始化stu2[0]的时候少初始化了一个元素，程序便自动为我们初始化为0。同时,stu1[2]和stu2[2]没有我们显示地初始化，所以程序自动将它们初始化为0。

说到数组，我们就不得不说指针，结构体指针和其它的指针拥有类似的操作方式。同时，当我们通过一个指针来操作一个实例变量的成员的时候，不必采用这种方式：(\*p).member，我们可以使用另外一个运算符：->，如p->member就等价于(\*p).member。取值运算符\*的优先级低于.，所以我们需要加上括号。

1. #include <stdio.h>
2. #define NUM 3
3. typedef struct
4. {
5. int age;
6. }type;
7. int main()
8. {
9. type data[NUM];
10. type \*p = data,\*q=&data[0];
11. p[0].age = 10;
12. printf("data[0].age的值：%d\n",p[0].age);
13. q->age = 11;
14. printf("data[0].age的值：%d\n", q->age);
15. getchar();
16. return 0;
17. }

程序8-3-3

程序的运行结果如下：

data[0].age的值：10

data[0].age的值：11

程序8-3-3中，我们声明了一个结构体数组，然后声明了两个结构体指针p和q，其中p指向整个数组，而q则指向数组中的第一个元素，第7章曾经探讨过，两种方式都是合理的。接下来我们分别通过p和q来访问了数组中的元素，p采用的是解除引用的方式（p[0]等价于\*(p+0)），而q则使用的是->运算符。从运行结果可以看出，两种方式都能正常发挥作用。

8.4 结构体与函数

前面的章节中介绍了这么多“xxx与函数”，所以读者肯定应该知道本节的内容有这些：结构体作为函数参数、结构体作为函数返回值、结构体数组作为函数参数、结构体数组作为函数返回值（实际上是结构体指针）。

首先我们来看一看最简单的，结构体作为函数参数，事实上，结构体作为函数参数和基本数据类型作为函数参数并没有本质区别，事实上，它甚至比指针作为函数参数还容易理解。形参是实参的拷贝，对形参的操作同样影响不了实参。

1. #include <stdio.h>
2. typedef struct
3. {
4. char name[20];
5. int age;
6. }student;
7. void fun(student);
8. int main()
9. {
10. student stu = {"余悦",19};
11. printf("stu的信息：姓名：%s，年龄：%d\n",stu.name,stu.age);
12. fun(stu);
13. printf("stu的信息：姓名：%s，年龄：%d\n", stu.name, stu.age);
14. getchar();
15. return 0;
16. }
17. void fun(student stu)
18. {
19. printf("stu的信息：姓名：%s，年龄：%d\n", stu.name, stu.age);
20. stu.age++;
21. }

程序8-4-1

程序运行结果如下：

stu的信息：姓名：余悦，年龄：19

stu的信息：姓名：余悦，年龄：19

stu的信息：姓名：余悦，年龄：19

首先，我们可以看出，fun中的stu的值main中的stu完全一样，同时我们在fun中改变了stu的成员的值之后，主函数中不受任何影响，这说明两个stu是独立的。

通过程序8-4-1，我们可以看出，结构体变量作为函数参数和基本数据类型作为函数参数没有本质区别。

接下来要说的是返回结构体的函数，也就是结构体作为函数返回值。我们曾说过，结构体的成员中可以有指针，而指针又可能与动态内存有关，这样，每个结构体在初始化的时候可能需要相当多的步骤，而如果我们用一个函数来完成这些初始化的工作，就能有效地减少重复的操作，减少代码量。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h> //实际上这里面已经包含了memory.h
4. #include <memory.h>
5. typedef struct
6. {
7. char \*name;
8. int age;
9. }student;
10. student creator();
11. void destroyer(student);
12. int main()
13. {
14. student t;
15. t = creator();//这里实际上是一个浅拷贝
16. printf("输入的信息\n");
17. printf("姓名：%s，年龄：%d\n",t.name,t.age);
18. destroyer(t); //同样地，需要一个函数来做收尾工作
19. rewind(stdin);
20. getchar();
21. return 0;
22. }
23. student creator()
24. {
25. student temp;
26. char buffer[256];
27. printf("创建一个新的学生信息\n");
28. printf("请输入姓名：");
29. scanf("%s",buffer);
30. printf("请输入%s的年龄：",buffer);
31. scanf("%d",&temp.age);
32. int lenth = strlen(buffer);
33. temp.name = malloc(lenth+1);
34. memcpy(temp.name, buffer, lenth + 1);
35. return temp;
36. }
37. void destroyer(student t)
38. {
39. free(t.name);
40. }

程序8-4-2

程序的某次运行结果如下：

创建一个新的学生信息

请输入姓名：**余悦**

请输入余悦的年龄：**19**

输入的信息

姓名：余悦，年龄：19

creator函数的主要作用是接受用户输入的各种信息，可以看到，提示语句加上输入语句需要占用很大一部分空间，如果我们不是用一个函数，而是采用每创建一个实例变量都写一次这些代码，那么我们的工作量会提高很多。

creator函数中，我们用了一个char数组buffer来暂时储存当前结构体变量的名字，这里我们依然没有对数据的安全性做检查，如果希望程序更加安全，可以将第29行的scanf替换成安全输入函数（scanf\_s或我们自己编写的安全函数）。在接受数据之后，我们根据实际情况来为temp.name指针分配内存，这样就比之前的一律采用大小为20的数组更加灵活。最后在接受所有数据之后，我们把临时变量temp当成函数的返回值，并且在主函数中用t来接受这个返回值，这里地方相当于是一个浅拷贝（因为temp肯定不会在其它地方被使用，它是一个局部变量）。

最后，当实例变量t不再有用时，我们还需要一个专门的函数destroyer来处理掉它，处理过程就是释放t.name指向的内存空间。程序8-4-2中destroyer函数只有一行，因此实际上没有必要单独作为一个函数，而如果需要收尾的东西特别多，那就应该使用函数了。

和普通数组相类似地，结构体数组也可以作为函数的参数（作为参数的实际上是一个结构体指针，也就是数组的首地址）。

1. #include <stdio.h>
2. typedef struct
3. {
4. char name[20];
5. int age;
6. }student;
7. void displayStudent(student \*, int);
8. int main()
9. {
10. student s[3] = { {"张三",19},{"李四",20},{"王五",21} };
11. displayStudent(s, 3);
12. printf("%s的年龄：%d\n", s[0].name, s[0].age);
13. getchar();
14. return 0;
15. }
16. void displayStudent(student \*s, int num)
17. {
18. for (int i = 0; i < num; i++)
19. {
20. printf("%s的年龄：%d\n",s[i].name,s[i].age);
21. }
22. printf("把%s的年龄+1\n",s[0].name);
23. s[0].age++;
24. }

程序8-4-3

程序的运行结果如下：

张三的年龄：19

李四的年龄：20

王五的年龄：21

把张三的年龄+1

张三的年龄：20

displayStudent函数接受一个student的指针和一个整数作为参数，student \*s也可以改写成student s[]。前面的章节中我们也探讨过，这两种写法是等价的，我们也说过不管哪种写法，s本身都是一个指针而不是一个数组。

由于我们传递的是数组s的地址，所以在displayStudent函数中我们不仅能够通过解除引用（下标也算是解除引用）来访问各个元素的成员，而且还能改变成员的值。这一点和普通的数组是相同的。

最后我们要说的就是返回结构体指针的函数，前面我们曾说过，返回结构体变量的函数有其利用价值，因为这样的函数可以将复杂的结构体初始化工作单独分离出来，从而减少代码量，减少出错的几率。而事实上，大多数时候，我们所需要处理的结构体变量不止一两个，而我们知道，结构体变量需要占用大量的内存，如果把这些变量全部放在栈空间中，显然是很不合适的，所以我们可以把它们放到堆空间里。而这样做也需要很多的操作，如申请堆内存，对申请到的内存进行初始化等等工作，所以我们可以把这些操作放到一个函数里面，在函数中完成所有的任务，然后返回已经处理好的结构体数组的指针。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. typedef struct
5. {
6. char \*name;
7. int age;
8. }student;
9. student \*createArray(int);
10. void destroyArray(student \*, int);
11. int main()
12. {
13. student \*data;
14. data = createArray(2);
15. printf("data数组中的信息：\n");
16. for (int i = 0; i < 2; i++)
17. {
18. printf("姓名：%s，年龄：%d\n",data[i].name,data[i].age);
19. }
20. destroyArray(data, 2);
21. rewind(stdin);
22. getchar();
23. return 0;
24. }
25. student \*createArray(int num)
26. {
27. if (num <= 0)
28. {
29. return NULL;
30. }
31. student \*temp;
32. temp = malloc(sizeof(student)\*num);
33. if (temp == NULL)

1. {
2. return NULL;
3. }
4. char buffer[256];
5. for (int i = 0; i < num; i++)
6. {
7. printf("创建一个新的学生信息\n");
8. printf("请输入姓名：");
9. scanf("%s", buffer);
10. printf("请输入%s的年龄：", buffer);
11. scanf("%d", &temp[i].age);
12. int lenth = strlen(buffer);
13. temp[i].name = malloc(lenth + 1);
14. memcpy(temp[i].name, buffer, lenth + 1);
15. }
16. return temp;
17. }
18. void destroyArray(student \*t, int num)
19. {
20. for (int i = 0; i < num; i++)
21. {
22. free(t[i].name);//先释放成员指针指向的内存
23. }
24. free(t);//再释放储存成员的内存
25. }

程序8-4-4

程序的某次运行结果如下：

创建一个新的学生信息

请输入姓名：**张三**

请输入张三的年龄：**19**

创建一个新的学生信息

请输入姓名：**李四**

请输入李四的年龄：**20**

data数组中的信息：

姓名：张三，年龄：19

姓名：李四，年龄：20

可以看出，createArray函数中，我们执行的操作和程序8-4-2中的creator函数很类似。第39行的for循环可以说是完全按照creator函数改写的，事实上我们也完全可以用creator函数来替代这个for循环。

createArray函数所创建的数组存在于堆空间中，而数组中的每个元素都含有一个char型指针，这些指针又分别指向某片堆内存，这一点和通过二级指针申请二维数组很类似。同样地，在申请出来的数组不再被使用的时候，我们应该清理相关的内存，而我们也编写了一个相应的函数来完成这样的工作，在destroyArray函数中，我们先是通过一个循环来释放了每个元素的name指针指向的内存，然后释放了整个数组。

**试一试**

改写程序8-4-4，用creator函数来替换第39行for循环中的代码。

8.5 枚举类型

假设现在我们要编写一个考勤的程序，考勤程序访问30个同学的信息，显然我们应该用结构体来组织每个同学的信息。而每个结构体中，除了应该有姓名学号等信息外，还应该有一个变量用来表示该同学的状态（正常上课、迟到、请假等）。我们可以用字符串来储存这些状态，并且也不会花费太多空间，只是如果考勤程序需要经常执行找出所有正常上课的同学的操作，那么我们就会需要调用很多次strcmp函数来判断，这样就会使程序的运行效率大打折扣。所以不难看出，用整数来表示状态是更佳的选择，比较两个整数是否相等只需要很小的代价。

我们很容易想到的一种方法是，用0表示正常上课、1表示迟到、2表示请假，以此类推。这样有个很大的问题，那就是除了编写代码的人以外，谁都不知道0、1、2到底代表些什么，甚至就连作者自己也可能在一段时间后想不起这些东西，显然这样不利于代码的维护（也就是前面章节所提到的“幻数”）。显然更好的办法是使用#define指令来定义相关的宏，这也是我们在前面部分所推荐的方式，当时我们也说过，使用宏定义不是最佳方案，因为宏定义的常量会在预编译阶段被替换，也就是说，如果我们定义了一个常量NUM，它的值是5，然后我们在某个地方写了这样一行语句：int x=NUM;，那么预编译器会把NUM替换成5，最终编译器看到的就是int x=5;了。如果这个常量出了问题，编译器就会提示说常量5有问题，而不会说NUM有问题，在看到这个错误信息的时候，我们将会很疑惑，到底这个5是什么呢？找出错误所在又会花掉一点时间。

而我们前面说到过的最佳的方案，就是采用枚举，所谓枚举，就是一一列举。我们可以自定义一个类型，用这个类型来专门表示学生的状态。我们来看看枚举类型的基本用法。

1. #include <stdio.h>
2. typedef enum state
3. {
4. normal, //使用逗号而不是分号
5. leave,
6. late //结尾处没有任何符号
7. }state; //注意分号
8. typedef struct student
9. {
10. state s;
11. }student;
12. int main()
13. {
14. student stu;
15. stu.s = normal;
16. if (stu.s == normal)
17. {
18. printf("该同学正常出勤\n");
19. }
20. getchar();
21. return 0;
22. }

程序8-5-1

程序的运行结果如下：

该同学正常出勤

程序9-5-1中，我们首先定义了一种枚举类型enum state，并且在定义的同时用typefef为它起了一个“别名”，然后我们就可以在接下来的结构体中直接使用state声明枚举变量，如果没有typedef，第10行的state s;就只能改成enum state s了，这点和结构体很像。

需要注意的是，定以枚举类型和定义结构体一样，只是定义了类型，没有创建变量。state s相当于是把枚举类型state实例化了。和结构体不同的是，s作为state的实例变量不代表它同时拥有normal、leave等成员，而是它的取值只能是这几个中某一个，而在第15行，我们把stu.s赋值为normal。最后再将它与一个合法的值做比较。

计算机只能储存0和1，一切信息最终都是以数字的形式保存的，即使是枚举，也不例外。那么枚举类型和int型变量到底有哪些共通之处呢？

1. #include <stdio.h>
2. typedef enum
3. {
4. normal,
5. leave,
6. late
7. }state;
8. int main()
9. {
10. state a, b, c;
11. a = normal;
12. b = leave;
13. c = late;
14. printf("%d %d %d",a,b,c);
15. getchar();
16. return 0;
17. }

程序8-5-2

程序的运行结果如下：

0 1 2

可以看出，normal、leave、late3个枚举变量分别对应0、1、2这3个整数。事实上，枚举类型确实是以整数的形式储存的。而枚举类型中，第一个被声明的值对应0，而之后的值则递增。

枚举变量不仅以整数的形式储存，事实上它和int变量没有多大区别，甚至占用的内存的大小都是一致的。不过二者之间却不能直接相互赋值，我们通过一个程序来详细讲解这些问题。

1. #include <stdio.h>
2. typedef enum
3. {
4. normal,
5. leave,
6. late
7. }state;
8. int main()
9. {
10. state s=normal;
11. printf("枚举变量的大小：%d\n",sizeof(s));
12. int x = s; //直接赋值
13. printf("%d\n",x);
14. s = (state)2; //必须强制转换
15. printf("%d",s);
16. getchar();
17. return 0;
18. }

、、、

程序8-5-3

枚举变量的大小：4

0

2

枚举变量和int型变量占用同样的内存。同时，要把枚举变量的值赋值给int型变量可以直接赋值，而如果想要把整数赋值给枚举变量则需要强制转换。

需要注意的是，当把一个int型变量强制转换后赋值给一个枚举变量，并不会发生“绕回处理”。以程序9-5-3为例，将3转换后赋值给s，最终s的值不会是0，而是3，即使3不是state的合法取值。

1. #include <stdio.h>
2. typedef enum
3. {
4. normal,
5. leave,
6. late
7. }state;
8. int main()
9. {
10. state s;
11. s = (int)3;
12. if (s != normal)
13. {
14. printf("%d",s);
15. }
16. getchar();
17. return 0;
18. }

程序8-5-4

程序的运行结果如下：

3

可见，不仅s的值不是0，而且在进行!=运算的时候，它也被判定为一个不同于normal的值。

事实上，枚举变量的值不一定都是从0开始然后递增的，我们完全可以为枚举指定确定的值。只不过我们不推荐使用这种方式，因为它可能导致一些问题的发生。

1. #include <stdio.h>
2. typedef enum
3. {
4. first=10,
5. second=3,
6. third=2,
7. fourth,
8. fifth
9. }num;
10. int main()
11. {
12. printf("%d %d %d %d %d\n",first,second,third,fourth,fifth);
13. num t = second;
14. if (t == fourth)
15. {
16. printf("t的值为fourth\n");
17. }
18. getchar();
19. return 0;
20. }

程序8-5-5

程序的运行结果如下：

10 3 2 3 4

t的值为fourth

在枚举类型中，某个值ENUM对应的整数满足以下条件中的某一个：

1.若某个整数被赋值给ENUM，则ENUM对应这个整数。

2.若条件1不满足，则ENUM对应的整数为ENUM前一个值对应的整数加一。

3.若条件2不满足（ENUM是第一个值），则ENUM对应的整数为0。

根据这个条件，我们不难相同输出结果是这样。我们把3赋值给second，并且把2赋值给third，根据条件1，它们对应的值就分别是2和3。再根据条件2，我们可得fourth的值是3。这样，second和fourth就有了同样的值。所以之后我们将一个枚举变量t初始化为second，然后将它与fourth做比较，得到的结果居然是t==fourth是成立的。显然这种结果是对我们有害的，试想，如果在考勤软件中，某位同学的状态明明是正常上课，却因为枚举变量的问题而被判定成了迟到，那么造成的影响是非常不好的。

枚举本身非常简单，但它配合其它C语言知识却可以产生很好的效果。

前面第6章，程序6-4-11中，我们编写了一个简单的音乐播放器，该播放器可以播放用户指定路径的音乐，但是由于诸多限制，该播放器只能播放英文路径下的音乐（甚至音乐文件本身也不能带有中文），之所以会出现这种情况，是因为wchar\_t非常特殊，以至于在输入它的时候会有种种问题，即使前面6-4-11中我们已经实现了输入wchar\_t，但是输入中文的时候却会出现问题。要解决这个问题，最好的办法是先用char数组来接受用户的输入，然后再把这个char数组转换为wchar\_t的数组，而将char转换成wchar\_t需要调用一个Windows的API，这涉及到指针的相关知识，所以当时我们没有向读者展示这种操作。而现在我们有了指针的概念，可以很轻易的使用相关的API，接下来我们将结合指针、结构体、枚举等，编写一个更加强大的音乐播放器。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include <windows.h>
5. #include <conio.h>
6. #pragma comment(lib,"winmm.lib") //必须链接到静态库
7. #define MAXSIZE 9
8. #define FILELENTH 30
9. typedef enum state
10. {
11. play,
12. pause,
13. stop
14. }state;
15. typedef struct player
16. {
17. state playState;
18. char \*\*musicList;
19. int numOfSong;
20. wchar\_t \*cmdW;
21. char \*cmdC;
22. }player;
23. player \*createPlayer();//创建player结构体
24. void deletePlayer(player \*);//删除player结构体
25. void mainMenu(player \*);//显示主菜单
26. void addList(player \*);//向歌单中添加歌曲
27. void deleteList(player \*);//从歌单中删除歌曲
28. void playSong(player \*);//播放某首歌
29. void pauseSong(player \*);//暂停正在播放的歌曲
30. void resumeSong(player \*);//继续播放
31. void stopSong(player \*);//停止播放
32. void getStr(char \*, int lenth);//字符串输入函数
33. int main()
34. {
35. player \*pla = createPlayer();
36. mainMenu(pla);
37. deletePlayer(pla);
38. \_getch();
39. return 0;
40. }
41. player \*createPlayer()
42. {
43. player \*t = malloc(sizeof(player));
44. t->playState = stop;
45. t->musicList = malloc(MAXSIZE\*sizeof(char \*));
46. for (int i = 0; i < MAXSIZE; i++)
47. {
48. t->musicList[i] = malloc(FILELENTH\*sizeof(char));
49. }
50. t->numOfSong = 0;
51. t->cmdW = malloc(255\*sizeof(wchar\_t));
52. t->cmdC = malloc(255 \* sizeof(char));
53. return t;
54. }
55. void deletePlayer(player \*t)
56. {
57. if (t->playState != stop)
58. {
59. mciSendString(L"stop file", NULL, 0, NULL);
60. mciSendString(L"close file", NULL, 0, NULL);
61. }
62. for (int i = 0; i < MAXSIZE; i++)
63. {
64. free(t->musicList[i]);
65. }
66. free(t->musicList);
67. free(t->cmdW);
68. free(t->cmdC);
69. }
70. void mainMenu(player \*t)
71. {
72. char choice;
73. while (1)
74. {
75. system("cls");
76. printf("==================\n");
77. printf("1.添加歌曲\n");
78. printf("2.删除歌曲\n");
79. printf("3.播放歌曲\n");
80. if (t->playState == pause)
81. {
82. printf("4.继续播放\n");
83. }
84. else
85. {
86. printf("4.暂停播放\n");
87. }
88. printf("5.停止播放\n");
89. printf("6.退出播放器\n");
90. printf("==================\n");
91. do
92. {
93. choice = \_getch();
94. } while (choice<'1'||choice>'6');
95. system("cls");
96. switch (choice)
97. {
98. case '1':
99. {
100. addList(t);
101. break;
102. }
103. case '2':
104. {
105. deleteList(t);
106. break;
107. }
108. case '3':
109. {
110. playSong(t);
111. break;
112. }
113. case '4':
114. {
115. if (t->playState == pause)
116. {
117. resumeSong(t);
118. }
119. else
120. {
121. pauseSong(t);
122. }
123. break;
124. }
125. case '5':
126. {
127. stopSong(t);
128. break;
129. }
130. case '6':
131. {
132. printf("感谢使用\n");
133. return;
134. }
135. } //switch
136. }//while
137. }
138. void addList(player \*t)
139. {
140. if (t->numOfSong == MAXSIZE)
141. {
142. printf("歌单已满\n");
143. \_getch();
144. return;
145. }
146. printf("请输入歌曲路径：(%d个字符以内)\n",FILELENTH);
147. getStr(t->musicList[t->numOfSong], FILELENTH);
148. t->numOfSong++;
149. printf("添加完成\n");
150. \_getch();
151. }
152. void deleteList(player \*t)
153. {
154. if (t->numOfSong == 0)
155. {
156. printf("歌单已空\n");
157. \_getch();
158. return;
159. }
160. printf("要删除哪一首？\n");
161. printf("==================\n");
162. for (int i = 0; i < t->numOfSong; i++)
163. {
164. printf("%d.%s\n", i + 1, t->musicList[i]); //显示选项，选项为下标+1
165. }
166. printf("==================\n");
167. char choice;
168. do
169. {
170. choice = \_getch();
171. } while (choice<'1'||choice>t->numOfSong+1+'0');//下标是整数，选项是字符
172. int index = choice - 1 - '0';//计算出待删除的歌曲的下标
173. for (int i = index; i < t->numOfSong-1;i++)//i必须小于最大下标
174. {
175. strcpy(t->musicList[i], t->musicList[i + 1]);//依次移动数据
176. }
177. t->numOfSong--;
178. printf("删除成功\n");
179. \_getch();
180. }
181. void playSong(player \*t)
182. {
183. if (t->numOfSong == 0)
184. {
185. printf("歌单为空\n");
186. \_getch();
187. return;
188. }
189. printf("要播放哪一首？\n");
190. printf("==================\n");
191. for (int i = 0; i < t->numOfSong; i++)
192. {
193. printf("%d.%s\n", i + 1, t->musicList[i]); //显示选项，选项为下标+1
194. }
195. printf("==================\n");
196. char choice;
197. do
198. {
199. choice = \_getch();
200. } while (choice<'1' || choice>t->numOfSong + 1 + '0');
201. if (t->playState != stop)//如果有音乐正在播放
202. {
203. sprintf(t->cmdC,"stop file"); //首先停止
204. MultiByteToWideChar(CP\_ACP,0,t->cmdC,strlen(t->cmdC)+1,t->cmdW,255);
205. mciSendString(t->cmdW,NULL,0,NULL);
206. sprintf(t->cmdC, "close file");//然后关闭
207. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
208. mciSendString(t->cmdW, NULL, 0, NULL);
209. }
210. sprintf(t->cmdC,"open %s alias file",t->musicList[choice-1-'0']);//打开新的音乐
211. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
212. mciSendString(t->cmdW, NULL, 0, NULL);
213. sprintf(t->cmdC, "play file");
214. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
215. mciSendString(t->cmdW, NULL, 0, NULL);
216. t->playState = play;
217. printf("正在播放\n");
218. \_getch();
219. }
220. void pauseSong(player \*t)
221. {
222. if (t->playState != play)
223. {
224. printf("没有正在播放的音乐");
225. \_getch();
226. return;
227. }
228. sprintf(t->cmdC, "pause file");
229. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
230. mciSendString(t->cmdW, NULL, 0, NULL);
231. t->playState = pause;
232. }
233. void resumeSong(player \*t)
234. {
235. if (t->playState != pause)
236. {
237. printf("播放状态不为暂停，不能继续");
238. \_getch();
239. return;
240. }
241. sprintf(t->cmdC, "resume file");
242. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
243. mciSendString(t->cmdW, NULL, 0, NULL);
244. t->playState = play;
245. }
246. void stopSong(player \*t)
247. {
248. if (t->playState == stop)
249. {
250. printf("音乐已经停止");
251. \_getch();
252. return;
253. }
254. sprintf(t->cmdC, "stop file");//停止播放
255. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
256. mciSendString(t->cmdW, NULL, 0, NULL);
257. sprintf(t->cmdC, "close file");//关闭文件
258. MultiByteToWideChar(CP\_ACP, 0, t->cmdC, strlen(t->cmdC) + 1, t->cmdW, 255);
259. mciSendString(t->cmdW, NULL, 0, NULL);
260. t->playState = stop;
261. }
262. void getStr(char \*buffer, int lenth)
263. {
264. rewind(stdin);
265. char temp = 0;
266. if (lenth == 1)
267. {
268. buffer[0] = getchar();
269. }
270. else
271. {
272. int i;
273. for (i = 0; i < lenth - 1; i++)
274. {
275. temp = getchar();
276. if (temp == '\n')
277. {
278. break;
279. }
280. buffer[i] = temp;
281. }
282. buffer[i] = 0;
283. }
284. while (temp != '\n')
285. {
286. temp = getchar();
287. }
288. }

程8-5-6

这个播放器程序非常长，囊括了我们已经学过的一切知识，但它却并不难，接下来是详细的分析。

第1到第35行都是一些声明语句或者预编译指令，第6行的链接到指定的静态库是我们之前已经讲过的。而MAXSIZE和FILELENTH则是我们之后要用到的两个常量，MAXSIZE是歌单中最多能储存的音乐数，FILELENTH则是每首音乐的路径的最大长度。enum state枚举类型用来表示播放状态，分别是播放、暂停、停止。player结构体是程序的核心，它的作用是表示一个播放器，需要储存各种播放信息，state是播放状态、musicList是一个二级指针，它会被创建一个二维动态数组来储存歌单、numOfSong记录歌单的大小、cmdW和cmdC用来储存播放指令，与API的调用有关，cmdC能够直接处理中文，但API接口需要cmdW作为参数，所以我们需要将命令写入cmdC，然后将它转换，转换后的结果储存在cmdW中。

createPlayer函数用来“初始化”player结构体，deletePlayer则用来“销毁”它，这种做法我们之前就已经介绍过。而接下来的一系列的函数则是播放器的各个组件，负责播放音乐、维护歌单等。

主函数中，一共只有几行代码，我们首先初始化一个player变量，然后用pla指针来接受它的地址，接着把它传递给mainMenu函数，mainMenu中将会开始主循环，一旦用户选择退出程序，mainMenu就会返回，然后主函数中将会执行deletePlayer的操作。

createPlayer函数中，我们首先申请一个player的动态内存，然后对它进行各种初始化操作，然后返回它。我们将初始的播放状态设为stop，然后为musicList分配内存，由于musicList是一个二级指针，我们也想让它指向一个二维数组，所以这里的内存分配分两步，首先分配用来储存一级指针的空间，然后为这些一级指针分别分配内存，这是第7章详细讲解过的知识。然后我们把音乐数设定为0，并且分别为储存指令的cmdC和cmdW分配内存。至此，结构体的初始化完成。

而在deletePlayer函数中，我们首先判断播放器的状态是否为停止，如果不是则停止播放并关闭音乐文件，执行停止和关闭的操作调用了API，命令分别为”stop file”和”close file”，这类似于”play file”，总得说来，API的使用和前几章中是一样的，只是命令不同了（第一个参数不同了）。然后我们释放在createPlayer函数中申请出来的内存，首先是释放二级指针musicList的内存：先释放每个一级指针指向的内存，再释放储存这些一级指针的内存，这也是前面讲过的。然后再释放cmdC和cmdW。

mainMenu函数（75-143行）中，我们显示了主菜单，并接受用户的输入，这是典型的菜单选择程序，其中的技巧前面也多次提到，这里不再赘述。需要注意的就是显示第4个菜单的时候，我们根据播放器的状态不同而显示不同的选项，如果播放器状态为暂停(pause)，则显示“继续播放”，否则显示“暂停音乐”。在switch语句中，若选项为4，则也根据实际情况而选择调用不同的函数。

首先来看添加歌曲的函数，即addList（145-158行），在这个函数中，我们首先判断歌单是否已满（numOfMusic是否已经等于MAXSIZE），如果是则输出提示信息并返回，否则执行添加歌曲的操作。添加歌曲的时候，接受用户输入所用到的函数是我们之前编写好的getStr函数，使用这个函数能让程序变得更加安全。在接收输入之后，我们让numOfSong自增。然后输出提示信息并结束函数。

删除歌单的函数deleteList（160-188行）看起来复杂一些，同样地，我们首先判断歌单是否为空，若不为空则执行删除操作。这里我们把所有歌曲用菜单的形式显示了出来，有多少首歌就显示多少个选项，由于我们将MAXSIZE设定为9，所以最多显示9个选项，这9个选项用数字键就能全部表示，如果需要表示更多首歌，菜单选择部分就会变得更复杂。这里我们可以思考一下某一首歌，它在歌单中的下标与它在菜单中的选项的关系，很明显，对于下标为0的歌曲来说，它的选项是1，下标为1的歌曲的选项为1，以此类推。所以我们可以认为，选项-1等于下标。但是这样还不够，因为下标是整数，而选项是用字符来表示的，因此要把选项转换为下标，还需要减去’0’。因此才有了第180行的换算公式。在知道了要删除哪一首歌之后，我们就要进行删除操作了，那么到底该怎样删除呢？假设歌单里有7首歌，我们要删除第2首，直接释放对应的内存空间是肯定不合理的，这样之后就没法再使用相应的内存了。所以最好的办法是，把第3、4、5、6、7首歌全部往前移动一个位置，即用第3首歌来占用第2首歌的位置，第4首歌占用第3首歌的位置，以此类推。所以第181行我们用了一个循环来不断执行“移动”操作。接着用前面的例子，有7首歌，要删除第2首，我们可以想象，在执行strcpy函数的之后，只需要执行到strcpy(t->musicList[5], t->musicList[5 + 1])即可，所以不难理解i为什么必须小于最大下标。在执行完删除操作后，我们让numOfSong自减，然后输出提示信息。

playSong函数（190-229行），同样比较复杂。第192到209行的代码和deleteList类似，都是菜单选择。在接受完用户的输入以后，我们首先用一个if语句判断了当前的播放状态，如果为正在播放或者暂停状态，则需要停止并关闭之前的文件，在执行这些操作的时候，我们使用了API函数MultiByteToWideChar，第一个参数是一个宏，我们不必理会，第二个参数是待转换的字符串，这里当然应该是cmdC，第三个参数则是待转换的字节数，应该设定为命令字符串的长度+1，第四个参数则是接受转换结果的数组，这里应该是cmdW，最后一个参数则是cmdW的大小，我们如实填写即可。在API函数的调用完成后，我们就可以直接将cmdW作为mciSendString的参数了，需要注意的是，这里我们完全没必要写得这么复杂，事实上我们完全可以把212-214行改写成一行，即mciSendString(L"stop file", NULL, 0, NULL);，之所以写成这么3行，是为了让读者多感受几次这个转换的过程。在完成准备工作后，我们就可以播放音乐了，首先我们将音乐的路径通过sprintf写进cmdC，因为这时候音乐路径是未知的，所以我们必须采用sprintf+MultiByteToWideChar的方式来进行转换。然后接下来的操作就是打开文件+播放音乐，再把播放状态设为play，然后输出提示信息了。

pauseSong、resumeSong、stopSong里的操作都是类似的，首先判断是否满足条件（播放状态才能暂停、暂停状态才能继续、状态为关闭则不能再次关闭），然后执行对应的指令，最后再将播放状态设置为对应的值。

总的说来，这个播放器不算复杂，但是里面有比较多的编程思想，并且它结合了前面章节的各种知识，如果读者将这个程序理解到位，说明读者的C语言基础已经很不错了。

在这个程序中，mciSendString不是重点，因为它既不属于C语言标准库，作用也并不大，如果读者对它有兴趣，可以自行查阅它使用的各种播放命令。MultiByteToWideChar是一个比较有意义的函数，它提供从char向wchar\_t转换的功能，事实上，C语言标准库（stdlib.h）也提供mbstowcs函数来实现这个转换，只是这个函数对中文的处理能力很差，字符串中含有中文会导致转换不成功，所以我们更推荐使用MultiByteToWideChar。而如果要从wchar\_t转换到char，我们需要调用的API是WideCharToMultiByte，对应的库函数则是wcstombs。

8.6 联合体

和结构体类似地，C语言中还有一种叫联合体的东西，联合体的声明方式如下：

union 标识符

{

数据类型1 成员1;

数据类型2 成员2;

…

数据类型n 成员 n;

};

和结构体不同的是，联合体中，各个成员不是同时存在的，同一时间只存在一个成员。也就是说，如果有这样一个联合体：

union t

{

int x;

char y;

};

那么这个联合体的实例变量要么储存x，要么储存y。结构体为每个成员都分配了可用的空间，但联合体不同，它的空间只能供一个成员使用。

这么讲解读者不会有什么深刻的理解，我们不妨编写一个程序来演示一下。

1. #include <stdio.h>
2. typedef union test
3. {
4. int x;
5. char y;
6. }test;
7. int main()
8. {
9. test a;
10. a.x = 10;
11. printf("a.x=%d\n",a.x);
12. a.y = 'a';
13. printf("a.x=%d\n", a.x);
14. getchar();
15. return 0;
16. }

程序8-6-1

程序在某台计算机上的运行结果如下：

a.x=10

a.x=97

为什么要强调在某台计算机呢？因为第二次输出a.x的时候，输出结果不一定为97，虽然通常情况下都是97。输出的结果与计算机的操作系统有关，后面我们会具体说明。不过不管怎样，我们可以确定的是，当我们改变a.y的值的时候，a.x也受到了影响，这说明它们共用一片内存。

前面我们讲解过结构体的大小的计算方法，而联合体的大小的计算方法很简单，即最大的成员的大小。只不过需要注意的是，“最大成员的大小”依然应该是最大的类型的整数倍。

1. #include <stdio.h>
2. typedef union test
3. {
4. int x;
5. char y[5];
6. }test;
7. int main()
8. {
9. printf("test的大小：%d\n",sizeof(test));
10. getchar();
11. return 0;
12. }

程序8-6-2

程序的运行结果如下：

test的大小：8

可以看到，test结构体中，最大的成员是y，占用5个字节，而最大的基本类型是int，所以为了让大小是int的整数倍，整个结构体的大小也就变成8了。

接下来我们要探讨的就是联合体储存的本质。

1. #include <stdio.h>
2. typedef union test
3. {
4. int x;
5. char y;
6. }test;
7. int main()
8. {
9. test a;
10. printf("a的地址：%p\n",&a);
11. printf("a.x的地址：%p\n", &a.x);
12. printf("a.y的地址：%p\n", &a.y);
13. test \*p = &a;
14. \*(int \*)p = 10;
15. printf("a.x：%d\n",a.x);
16. \*(char \*)p = 'a';
17. printf("a.y：%c\n", a.y);
18. getchar();
19. return 0;
20. }

程序8-6-3

程序的某次运行结果如下：

a的地址：0018FA8C

a.x的地址：0018FA8C

a.y的地址：0018FA8C

a.x：10

a.y：a

可以看到，a、a.x、a.y的地址全是一样的。

程序的第13行，我们声明了一个test的指针，并且用a的地址来初始化了它。之后，我们把它强制转换成一个int型指针，并且对其进行解除引用的操作，然后我们输出a.x的值，发现结果正好与赋值给\*(int \*)p的值一样，这说明a.x的本质就是\*(int \*)p，而a.y则同理。

细心的读者可能就会发现问题所在了，程序9-6-1中，最终输出结果是97，而根据分析，不难发现a的大小是4个字节，而我们将a.x赋值为10之后，a对应的4个字节转换成二进制的数就是：00000000 00000000 00000000 00001010，也就是，前面24位全为0，而最后的8位为00001010（10对应的二进制数）。而当我们将a.y设置为’a’（97）之后，32位数就应该是：01100001 00000000 00000000 00001010，而这个数字对应的十进制数非常大（反正肯定不止97）。那么为什么最终输出的结果会是97呢？我们又为什么说输出结果与计算机本身有关呢？我们来做详细的解析。

原来，执行完a.y=10的操作之后，a变成了00001010 00000000 00000000 00000000而不是00000000 00000000 00000000 00001010，出现这种现象是因为小端模式（**Little-Endian**）。事实上，当一个数据占用的内存超过一个字节之后，小端模式就会发挥出作用。

分析一个数据在内存中的表示的步骤是：

1.将它转换成n\*8位的二进制数（n为其占用的字节数，int型是4个字节）。

2.把这个二进制数分为n份，每份8位。

3.将这n份数据的位置颠倒，最前的换到最后。

以10为例，10转换成4\*8位的二进制数并且分为等量的4份是00000000 00000000 00000000 00001010，然后把它们颠倒，就变成了00001010 00000000 00000000 00000000。

再以1024为例，1024对应的二进制数分成4份是00000000 00000000 00000100 00000000，位置颠倒之后是00000000 00000100 00000000 00000000。

对于字符’a’来说，由于它只占用一个字节，所以它转换成二进制是01100001，在内存中的表示也是01100001。所以a.y=’a’最终会让a变成01100001 00000000 00000000 00000000。

现在读者应该能理解程序9-6-1了，出现小端模式的原因则关系到系统底层的设置，由于这些原理超越了C语言基础教程的范围，且具有一定的难度，所以我们不做介绍，感兴趣的读者可以在自己有了一定的知识储备后继续学习。

我们可以写一个简单的程序来判断计算机是否为小端模式。原理也很简单，首先声明一个整形变量，然后将它赋值为97，最后再将它的地址强制转换成char \*，然后再解除引用，最后再和’a’比较，相等则是小端模式，否则不是。

1. #include <stdio.h>
2. int main()
3. {
4. int num = 97;
5. int \*p = &num;
6. if (\*(char \*)p == 97)
7. {
8. printf("这台计算机采用小端模式\n");
9. }
10. else
11. {
12. printf("这台计算机采用大端模式\n");
13. }
14. getchar();
15. return 0;
16. }

程序8-6-4

8.7 结构体与链表

链表（**linked list**）是一种数据结构，不属于C语言基础部分，但是比起其它数据结构来说，链表相对简单很多，并且应用也比较广泛，所以通常C语言基础教材中对链表都会有所介绍。

前面的章节中，我们介绍了数组和动态数组，我们知道动态数组比起传统数组的优势在于大小可以在运行阶段确定，而不用在编译阶段就确定，同时，传统数组一旦装满就不能再扩张空间，而动态数组则随时可以通过realloc和memcpy等函数来扩大或缩小空间，动态数组相对传统数组灵活很多。

而事实上，不管是哪一种数组，它们都有一个共同点，那就是所有元素一定是连续储存的，假设有一个数组num，那么num+0表示下标为0的元素的地址，而num+1则表示下标为1的元素的地址，以此类推。不难看出，数组的储存是线性的，所以数组也可以被认为是线性表（**line list**）。

而链表则采用一种比数组更加灵活的储存方式，假设第1个元素的地址是p，则第2个元素的地址可能是p+1、p+n，甚至是p-n，也就是说链表中的所有元素是可以不连续的。

那么链表到底长什么样呢？要解决这个问题，我们首先要思考下面这种写法是否合法：

struct data

{

int num;

struct data \*p;

};

我们的第一感觉通常都是不合法，但事实上，这样是合法的。结构体的成员中，可以包含结构体，但是不能包含它自己，这一点是没有疑问的。但是，我们在struct data中包含的是struct data的指针，而不是实例变量。我们知道，在同一个程序中，所有的指针都是一样大的，而指针之所以要有类型，是因为它的目标占用的空间不同。

而我们可以看出，在32位的程序中，struct data所占用的空间一定是8个字节，不管p是什么类型的指针。而struct data的大小是确定的，所以p的存在就合法了，它知道自己的目标的大小是8。

1. #include <stdio.h>
2. struct data
3. {
4. int num;
5. struct data \*address;
6. };
7. typedef struct data data;
8. int main()
9. {
10. data t;
11. t.address = &t;
12. t.num = 10;
13. printf("t的地址是：%p，t.address的值是：%p\n",&t,t.address);
14. printf("t.num的值是：%d\n",t.address->num);
15. getchar();
16. return 0;
17. }

程序8-7-1

程序的某次运行结果如下：

t的地址是：0018F94C，t.address的值是：0018F94C

t.num的值是：10

程序8-7-1很短，但是有很多需要注意的地方。首先我们可以看到第5行，声明address的时候，我们用的是struct data，而不是data，不仅仅是因为我们把typdef放在了下面。typedef发生在struct data的大小和成员都确定之后，而address的声明则是这个确定的过程，所以任何情况下，使用data替换struct data都是非法的。

在第11行，我们将t的地址赋值给了t.address，也就是说它的成员中保存了它自己的地址，可以看到&t和t.address的值确实相等，而这意味着我们甚至能通过t.address来访问t本身，所以我们在第14行，通过t.address来输出了t.num的值。

通过程序8-7-1，我们知道了结构体中含有指向该结构体的指针是合法的了，有了这个基础，我们就能开始研究链表了。

事实上，链表也分为很多种，包括单向链表、双向链表、循环链表等，其中又以单向链表最简单，所以我们主要介绍单向链表。

在数组中，储存数据的最小单元被称为元素，而在链表中，最小的单位叫做结点（**node**）。程序8-7-1中，struct data的实例变量t就相当于是一个结点。我们不难看出，一个结点中分为两部分，一部分用来储存数据，另一部分则用来储存指针，储存数据的部分被称为数据域，储存指针的部分则被称为指针域。

很多人都玩过RTS游戏(**Real Time Strategy，即时战略类游戏**)。这种游戏有一个共同的特点，那就是战斗是即时进行的。玩家自己都不知道这场战斗会造多少单位出来，更不用说程序了。每个单位都需要占用内存空间，既然不知道要制造多少部队，那么用数组来表示部队是不行的。同时动态数组也不可行，因为单位阵亡后就不再占用空间，我们就应该回收它占用的内存，在之前的播放器程序中我们已经见识过，要删除歌单中的某一首歌会让它之后的歌曲也跟着移动，这说明当我们需要经常删除和插入数据的时候，用数组不是很好的选择。

接下来我们便编写一个程序来模拟单位制造的过程，即展示链表的基本操作——链表的建立。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #define SIZE 5
4. typedef struct node
5. {
6. int ID;
7. struct node \*next;
8. }node;
9. int main()
10. {
11. node \*head = NULL, \*tail = NULL;
12. for (int i = 0; i < SIZE; i++)
13. {
14. node \*temp = malloc(sizeof(node));
15. temp->ID = i;
16. temp->next = NULL;
17. if (head == NULL)
18. {
19. head = temp;
20. tail = head;
21. }
22. else
23. {
24. tail->next = temp;
25. tail = temp;
26. }
27. }
28. for (node \*temp = head; temp != NULL; temp = temp->next)
29. {
30. printf("%d ",temp->ID);
31. }
32. getchar();
33. return 0;
34. }

程序8-7-1

程序运行结果如下：

0 1 2 3 4

struct node用来表示链表的结点，可以看到，struct node的数据域中只有一个int型数据，我们完全可以根据实际情况来做相应改变。

主函数中，我们声明了两个struct node的指针，head和tail。我们知道，这个程序的作用是模拟RTS游戏中创建单位的过程，所以肯定应该使用动态内存，而动态内存通常没有名字，使用malloc将它们分配出来后必须使用指针来“接受”，不然就无法访问。而我们的目的是把所有的结点组合成一个链表，所以我们必须要有一个指向链表的指针，否则无法访问链表中的数据，而head就是指向链表的指针。至于tail，是为了方便向链表中添加新的结点，即使没有它，有关链表的一切工作也是能完成的（只要有head），只不过效率会低很多。

第12行的for循环的目的就是建立一个链表。我们首先通过一个临时指针temp来接受malloc的返回值，然后通过temp来设置了通过malloc分配出来的结点的成员的值，首先是把ID设为i，然后把指针域的指针设为NULL。接着的工作就是把这个结点加入到链表中。

在向链表中添加结点的时候，我们首先判断head的状态，如果head为NULL，说明此时链表中还没有任何结点，这时候我们就应该把当前结点设为首结点，即让head指向它，此时链表中只有唯一的一个结点，它同时也是尾结点，所以我们需要让tail也指向它。而如果head不为NULL，说明链表中已有其它结点，这时候我们就没必要再改变head的值，直接从链表的尾部添加结点即可。添加的方法就是，让尾结点的next指向新的结点，然后再让tail指向新的结点，即让新的结点成为尾结点。

接下来我们使用第28行的循环来访问了所有的结点中的数据，首先从首结点开始访问，即首先访问head指向的结点，然后每访问一个结点之后，都访问该结点的下一个结点（next），如果该结点的下一个结点为NULL，则说明链表中的结点已经访问完毕。

事实上，程序8-7-1并不完整，因为它不包含收尾工作，即没有完成对所有结点的空间的释放工作，我们说这样的做法是不正确的，但是考虑到程序的重点是链表的建立，如果把销毁链表的工作也加上，会让程序更难理解，所以我们省去了这一部分的工作，而现在我们知道链表该如何建立了，接下来我们将通过一个程序演示链表的销毁。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #define SIZE 5
4. typedef struct node
5. {
6. int ID;
7. struct node \*next;
8. }node;
9. int main()
10. {
11. node \*head = NULL, \*tail = NULL;
12. …//此处省去程序8-7-1中第12到27行的代码
13. for (node \*temp = head; temp != NULL; )
14. {
15. node \*p = temp->next;
16. free(temp);
17. temp = p;
18. }
19. getchar();
20. return 0;
21. }

程序8-7-2

程序8-7-2和程序8-7-1有部分代码是类似的，为了突出重点，我们省去了这一部分的代码。第13到18行的作用就是销毁链表，可以看到，总得来说和访问所有结点是一样的（事实上，也确实是在访问每个结点，只不过这个访问的内容是销毁）。不同之处在于我们用了一个p指针来保存temp->next，而没有选择在for循环的条件中写temp=temp->next，这是因为temp指向的结点被free掉之后，里面的数据也就不存在了，自然也就找不到temp->next了，所以需要用p来临时地保存一下。

以程序8-7-1和程序8-7-2为例，其中的链表的储存方式可以用下图表示：

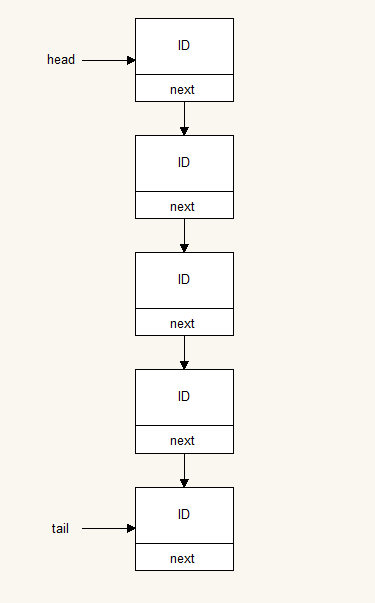


图8-7-1

其中，ID表示数据域，next表示指针域。可以看出，head保存了第1个结点的地址，即指向第1个结点，此后，每个结点的next都保存了下一个结点的地址（指向下一个结点），这样我们就能通过head找到所有的结点。tail指向最后一个结点，而最后一个结点中，next的值为NULL，因为没有下一个结点了。

图中的链表被称为单向链表，通过任意一个结点，我们能找到它后面的所有结点，但不能找到它前面的结点。

现在我们实际上已经掌握了链表的建立、遍历（即访问所有结点）、销毁操作，这些操作相对简单，接下来我们将研究结点的插入和结点的删除操作。

首先来看看结点的插入，结点的插入分为3种情况，从表头插入、从表尾插入、在指定位置插入。事实上程序8-7-1中，链表的建立就已经包含从表尾插入结点的操作了。接下来我们通过一系列的程序来分别演示这些操作。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. #define SIZE 3
5. #define NAME 8
6. typedef struct
7. {
8. char name[NAME];
9. int ID;
10. }student;
11. typedef struct node
12. {
13. student data;
14. struct node \*next;
15. }node;
16. typedef struct
17. {
18. node \*head;
19. node \*tail;
20. }linkList;
21. linkList \*createLinkList();
22. void deleteLinkList(linkList \*);
23. void pushFront(linkList \*,student);
24. int main()
25. {
26. linkList \*l = createLinkList();
27. student s;
28. char name[SIZE][NAME] = {"张三","李四","王五"};
29. for (int i = 0; i < SIZE; i++)
30. {
31. s.ID = i + 1;
32. strcpy(s.name, name[i]);
33. pushFront(l, s);
34. }
35. for (node \*t = l->head; t != NULL; t = t->next)
36. {
37. printf("学号：%d 姓名：%s\n",t->data.ID,t->data.name);
38. }
39. deleteLinkList(l);
40. getchar();
41. return 0;
42. }
43. linkList \*createLinkList()
44. {
45. linkList \*l = malloc(sizeof(linkList));
46. l->head = NULL;
47. l->tail = NULL;
48. return l;
49. }
50. void deleteLinkList(linkList \*l)
51. {
52. node \*t,\*p;
53. for (t = l->head; t != NULL;)
54. {
55. p = t->next;
56. free(t);
57. t = p;
58. }
59. free(l);
60. }
61. void pushFront(linkList \*l,student data)
62. {
63. node \*t = malloc(sizeof(node));
64. t->data = data;
65. t->next = NULL;
66. if (l->head == NULL)
67. {
68. l->head = t;
69. l->tail = t;
70. }
71. else
72. {
73. t->next = l->head;
74. l->head = t;
75. }
76. }

程序8-7-3

程序的运行结果如下：

学号：3 姓名：王五

学号：2 姓名：李四

学号：1 姓名：张三

程序8-7-3比较长，包含的思想也比较深，我们来一一分析。第1到第5行是头文件的包含和宏定义。第6到20行则是各种结构体的声明，可以看到，我们一共声明了3个结构体，按层次分别是最基本的储存信息的结构体student、链表结点node、链表结构体linkList。之所以把结点的数据域单独提取出来，整合成一个结构体，是为了让node显得更直观，数据域和指针域分界清晰。而linkList结构体的作用则是将链表的head和tail集中到一起，这样有助于代码的可读性，也让代码变得更好维护。

createLinkList的作用是申请一个linkList结构体的空间，将其初始化并返回其地址。deleteLinkList的作用则是销毁链表中的所有结点以及链表本身。pushFront的作用则是从链表的头部插入结点。

主函数中，我们通过createLinkList函数创建了一个链表，并且用l来接受它的地址，此时链表l是一个空链表。然后我们通过一个循环往链表中插入了3个结点，由于我们选择的方式是从头部插入，所以最后插入的结点就是头结点，最先插入的结点则是尾结点，从链表遍历的结果看来，也确实是这样。

createLinkList函数比较简单，它的作用只是申请内存，然后把指针初始化为NULL，最后返回地址。

deleteLinkList中，销毁链表的方式我们在之前的程序程序8-7-2中已经介绍过了，p指针的存在非常关键。和程序8-7-2不同的是，在销毁所有结点之后，还要销毁链表本身。

而pushFront不困难，仔细观察会发现它和程序8-7-1中从尾部插入新的结点非常类似：申请一个新的结点、为结点赋值、按照一定规则插入新的结点。而规则也是很类似的，首先判断头指针是否为空，若为空则说明整个链表为空，这时候我们需要采用特殊的处理方式，即让待插入的结点同时成为头结点和尾结点；如果链表不为空，则我们不必理会尾结点，直接用待插入结点取代头结点，取代的方式为让待插入结点指向原来的头结点，然后再让head指向刚刚插入的结点，这就完成了从头部插入的操作，代码很短，理解起来也很简单。

我们再来看看从尾部插入结点，虽然程序8-7-1中已经展示了，不过显然不够完美（head和tail没有装在linkList里面）。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. /\*省去了各种定义\*/
5. int main()
6. {
7. linkList \*l = createLinkList();
8. student s;
9. char name[SIZE][NAME] = {"张三","李四","王五"};
10. for (int i = 0; i < SIZE; i++)
11. {
12. s.ID = i + 1;
13. strcpy(s.name, name[i]);
14. pushBack(l, s);
15. }
16. for (node \*t = l->head; t != NULL; t = t->next)
17. {
18. printf("学号：%d 姓名：%s\n",t->data.ID,t->data.name);
19. }
20. deleteLinkList(l);
21. getchar();
22. return 0;
23. }
24. /\*省去了createLinkList和deleteLinkList\*/
25. void pushBack(linkList \*l,student data)
26. {
27. node \*t = malloc(sizeof(node));
28. t->data = data;
29. t->next = NULL;
30. if (l->head == NULL)
31. {
32. l->head = t;
33. l->tail = t;
34. }
35. else
36. {
37. l->tail->next = t;
38. l->tail = t;
39. }
40. }

程序8-7-4

程序的运行结果如下：

学号：1 姓名：张三

学号：2 姓名：李四

学号：3 姓名：王五

为了使代码更简洁，重点更突出，我们省去了一些代码，这些代码和程序8-7-3是完全一致的。不同之处在于pushFront变成了pushBack，相关代码也有改变。不难理解，pushBack的工作原理是，首先判断链表是否为空，为空则让head和tail都指向待插入结点；如果不为空，则让尾结点的next指针指向待插入结点，然后让tail指向待插入结点，这样就完成了从尾部插入结点的操作。

最后来看在任意位置插入，这个操作相对复杂一些。要在指定位置插入，首先要指定一个位置，这个位置用一个整数来表示，我们规定：在x位置插入结点的意思是在第x个结点后面插入结点，特别地，当x为0时，表示从表头插入；当x大于等于结点数时，表示从表位插入。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. /\*省去了各种定义\*/
5. void insertAt(linkList \*, int,student);
6. int main()
7. {
8. linkList \*l = createLinkList();
9. student s;
10. char name[SIZE][NAME] = { "张三","李四","王五" };
11. for (int i = 0; i < SIZE; i++)
12. {
13. s.ID = i + 1;
14. strcpy(s.name, name[i]);
15. pushBack(l, s);
16. }
17. s.ID = 10;
18. strcpy(s.name, "余悦");
19. insertAt(l, 2, s);
20. for (node \*t = l->head; t != NULL; t = t->next)
21. {
22. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
23. }
24. deleteLinkList(l);
25. getchar();
26. return 0;
27. }
28. /\*省去了createLinkList、deleteLinkList和pushBack\*/
29. void insertAt(linkList \*l, int index, student data)
30. {
31. if (index < 0)
32. {
33. return;
34. }
35. node \*t = malloc(sizeof(node));
36. t->data = data;
37. t->next = NULL;
38. if (l->head == NULL)
39. {
40. l->head = t;
41. l->tail = t;
42. return;
43. }
44. if (index == 0)//相当于从头部插入
45. {
46. t->next = l->head;
47. l->head = t;
48. return;
49. }
50. node \*p=l->head,\*q=NULL;
51. for (int i = 0; i < index-1; i++)
52. {
53. p = p->next;
54. if (p == l->tail)
55. {
56. break;
57. }
58. }
59. if (p == l->tail)
60. {
61. p->next = t;
62. l->tail = t;
63. }
64. else
65. {
66. q = p->next;
67. p->next = t;
68. t->next = q;
69. }
70. }

程序8-7-5

程序的运行结果如下：

学号：1 姓名：张三

学号：2 姓名：李四

学号：10 姓名：余悦

学号：3 姓名：王五

和程序8-7-4一样，我们省去了一些代码。在主函数中，除了有用pushBack插入结点的循环，我们还多了几行代码用来测试insertAt函数。

可以看到，insertAt函数有3个参数，第1个是目标链表，第2个则是插入的位置，第3个是插入的结点的数据域。

insertAt比起pushFront和pushBack都要复杂得多，首先要对index的值加以判断，如果小于0则是不合法的数据，直接返回即可。而接下来还要判断链表本身，如果链表为空，则我们直接将结点加入链表即可，同时，为了让代码不至于缩进太多，我们在if语句里面使用了return，以此避免使用else。

如果链表不为空，我们还要对index加以判断，如果是0，则从表头插入并结束函数调用，结束调用也是为了不产生else。

如果链表不为空且需要插入的位置不是表头，那么我们就要开始使用一个循环来定位了。 我们声明了两个指针p和q，p指向的结点在插入后是待插入结点的前一个结点（也叫前驱结点），而q则指向待插入结点的后一个结点（后继结点），我们要把待插入结点放置在p和q之间。首先，不难理解循环次数是index-1，我们可以用特殊值来理解：假设index是1，那么p和q的位置已经是正确的了，不必再循环。如果index大于1，那么我们就要改变p的值，直到它指向正确的位置为止，移动该指针的方式就是让它每次都指向自己的下一个结点。同时，我们还要对p的值加以判断，因为index可能大于结点数（我们规定这是合法的），而当index大于结点数的时候，循环次数就应该是结点数-1而不是index-1，所以当p指向tail的时候，应该立即终止循环。

当p就位之后，我们还需要对p进行判断，如果它指向尾结点，那么我们就把待插入结点按照从表尾插入的方式添加到链表中；而如果p指向的结点不是尾结点，那么我们需要将q设为q->next，然后让p-> next指向待插入结点，再t->next指向q即可。

图8-7-2和图8-7-3分别展示了程序8-7-5中，插入前和插入后的状态

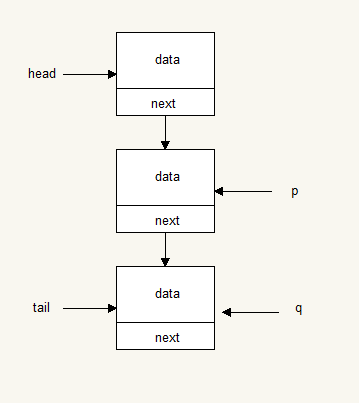


图8-7-2

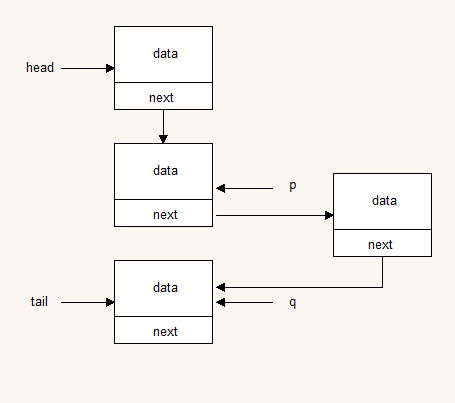


图8-7-3

说完了插入操作，我们再来看更复杂的删除操作，和插入一样的，删除操作也分为从表头删除、从表尾删除、从指定位置删除，我们按顺序来分别介绍这三种操作。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. /\*省去部分代码\*/
5. void popFront(linkList \*);
6. int main()
7. {
8. linkList \*l = createLinkList();
9. student s;
10. char name[SIZE][NAME] = { "张三","李四","王五" };
11. for (int i = 0; i < SIZE; i++)
12. {
13. s.ID = i + 1;
14. strcpy(s.name, name[i]);
15. pushBack(l, s);
16. }
17. popFront(l);
18. for (node \*t = l->head; t != NULL; t = t->next)
19. {
20. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
21. }
22. deleteLinkList(l);
23. getchar();
24. return 0;
25. }
26. /\*省去部分代码\*/
27. void popFront(linkList \*l)
28. {
29. if (l->head == NULL)
30. {
31. return;
32. }
33. node \*t=l->head->next;
34. free(l->head);
35. l->head = t;
36. if (t == NULL)
37. {
38. l->tail = NULL;
39. }
40. }

程序8-7-6

程序的运行结果如下：

学号：2 姓名：李四

学号：3 姓名：王五

程序8-7-6中我们同样省去了大部分重复的代码，这个程序的核心在于popFront函数，它的作用是从表头删除一个结点。

可以看出，和从表头插入结点一样，从表头删除结点也是比较简单的。不同之处在于删除操作多了更多判断的条件。首先要判断链表本身是否为空，为空则直接返回，如果不为空则删除第一个结点并移动head指针，这一点和销毁链表时的操作很像。最后，在完成结点的删除操作后，我们还需要判断链表是否为空（即我们是否删除了链表中的最后一个结点），如果我们删除了链表中唯一的一个结点，那么我们还需要稍微设置一下，即把tail设为NULL，表示链表已空。而运行结果也验证了，链表中的第一个结点被删除了。

从表头删除结点是最简单的删除方式，从表尾删除则要复杂一些，因为我们需要在删除结点后重新定位tail，tail的新值就是尾结点的前驱结点的地址，而已知尾结点通常是无法找到尾结点的前驱结点的，所以我们还需要通过一个循环来寻找，这会让事情变得更复杂。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. /\*省去部分代码\*/
5. void popBack(linkList \*);
6. int main()
7. {
8. linkList \*l = createLinkList();
9. student s;
10. char name[SIZE][NAME] = { "张三","李四","王五" };
11. for (int i = 0; i < SIZE; i++)
12. {
13. s.ID = i + 1;
14. strcpy(s.name, name[i]);
15. pushBack(l, s);
16. }
17. popBack(l);
18. for (node \*t = l->head; t != NULL; t = t->next)
19. {
20. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
21. }
22. deleteLinkList(l);
23. getchar();
24. return 0;
25. }
26. /\*省去部分代码\*/
27. void popBack(linkList \*l)
28. {
29. if (l->head == NULL)
30. {
31. return;
32. }
33. if (l->head == l->tail)
34. {
35. free(l->head);
36. l->head = NULL;
37. l->tail = NULL;
38. return;
39. }
40. node \*prev = l->head;
41. while (prev->next != l->tail)
42. {
43. prev = prev->next;
44. }
45. free(l->tail);
46. prev->next = NULL;
47. l->tail = prev;
48. }

程序8-7-7

程序的运行结果如下：

学号：1 姓名：张三

学号：2 姓名：李四

在popBack函数中，我们首先判断链表是否为空，然后判断链表中是否只有一个结点，这是对两种特殊情况的处理方式，这样对特殊情况特殊处理的例子前面也已经见过不少了。真正有特色的是第44到51行，我们声明了一个临时变量prev来储存尾结点的前驱结点的地址，我们起初将它初始化为head，然后不断地移动它，直到它的后继结点是尾结点为止（标志是prev->next==l->tail）。当prev就位之后，我们就可以删除尾结点了，此时prev->next依然指向原来的位置，而事实上prev已经是尾结点了，所以我们需要把prev->next设置为NULL，然后把tail的值更新为prev。

我们说过，链表分为很多种，如果需要经常执行删除尾结点的操作，那么采用程序8-7-7中的这种单向链表是不合适的，因为寻找尾结点的前驱结点需要花费很多时间，使用双向链表是更好的选择。

最后再来看看在指定的位置删除，这个操作和在指定的位置插入很类似。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. /\*省略部分代码\*/
5. void deleteAt(linkList \*,int);
6. int main()
7. {
8. linkList \*l = createLinkList();
9. student s;
10. char name[SIZE][NAME] = { "张三","李四","王五" };
11. for (int i = 0; i < SIZE; i++)
12. {
13. s.ID = i + 1;
14. strcpy(s.name, name[i]);
15. pushBack(l, s);
16. }
17. deleteAt(l,2);
18. for (node \*t = l->head; t != NULL; t = t->next)
19. {
20. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
21. }
22. deleteLinkList(l);
23. getchar();
24. return 0;
25. }
26. /\*省略部分代码\*/
27. void deleteAt(linkList \*l,int index)
28. {
29. if ((l->head == NULL) || (index <= 0))
30. {
31. return;
32. }
33. if (index == 1)
34. {
35. node \*t = l->head->next;
36. free(l->head);
37. l->head = t;
38. if (t == NULL)
39. {
40. l->tail = NULL;
41. }
42. return;
43. }
44. node \*t = l->head,\*p;
45. for (int i = 0; i < index-2; i++)
46. {
47. t = t->next;
48. if (t ->next== l->tail)
49. {
50. break;
51. }
52. }
53. p = t->next;
54. if (p == l->tail)
55. {
56. free(p);
57. t->next = NULL;
58. l->tail = t;
59. }
60. else if(t==l->tail)
61. {
62. free(p);
63. l->head=NULL;
64. l->tail=NULL;
65. }
66. else
67. {
68. t->next = p->next;
69. free(p);
70. }
71. }

程序8-7-8

程序运行结果如下：

学号：1 姓名：张三

学号：3 姓名：王五

在指定位置删除结点是链表操作中最复杂的操作之一。deleteAt函数有两个参数，一个是目标链表，另一个就是删除的位置，我们规定删除x位置的结点表示删除第x个结点，当x大于结点数时，将x视为结点数来处理。

deleteAt函数中，我们首先判断链表是否为空，然后判断index的值是否合法，链表为空或index值不合法都将终止函数。然后我们判断index是否为1，如果是则按从表头删除结点来处理（参考程序8-7-6）。

接下来就是通过循环来寻找目标结点和目标结点的前驱结点，这里用p表示目标结点的前驱结点，用t表示目标结点，循环次数是index-2，我们同样可以用特殊值法来理解：删除第3个结点，t需要移动1次。在循环的时候，还需要判断t是否已经指向最后一个结点（我们认为index大于实际结点数也是合法的），如果是则需要立即终止循环。

之后，我们再设置p的值即可（将p设为t->next）。最后我们还需要进行判断，如果p指向最后一个结点，那么我们要按删除尾结点的方式来操作。要注意的是，我们还对t的值进行了判断，如果t指向首结点，并且p不指向尾结点，这说明链表中只有一个结点，并且index大于1，这时候我们清空链表即可。

如果这些条件都不满足，那么就按一般的方式来处理，即让待删除结点的后继结点成为待删除结点的前驱结点的后继结点，然后删除之。

关于单向链表的操作就介绍这些。事实上，这些操作总结起来也并不复杂，它们分为两类：插入操作和删除操作，而插入操作和删除操作的位置又分别有3种情况：在头部、在尾部、在指定位置，进行这些操作之前还要做对应的检查工作。

我们所引用到的各种例子都只是为了演示链表操作的基本原理。在实际应用中，我们用到的方法不一定非常适合，而通常都需要根据实际情况做对应的改进。程序8-7-8中，在任意位置删除结点的时候，我们用了一个循环来寻找目标结点，而每次循环都要执行一次if语句，这会消耗一定的时间。如果需要经常执行这样的操作，那么所有的if语句累积起来就会是一个相当大的数字。所以我们应该稍加修改，以节省更多的资源，最容易想到的方法就是在linkList结构体中增加一个int型成员用来记录结点的数目，这样我们就很清楚循环的次数是多少了，而不再是判断两个指针是否相等，有兴趣的读者可以自己实现这种改进。

总得说来，链表的类型有很多种，它们分别适用于各种不同的情况。我们曾介绍过，除了单向链表外还有双向链表和循环链表，双向链表和循环链表都有着比较广泛的应用，我们分别演示一下这两种链表的基本操作。

先来看看双向链表，双向链表的指针域中有两个指针，分别指向结点的前驱结点和后继结点。我们说，在单向链表中，通过一个结点可以找到它后面的所有结点，但不能找到它前面的结点，而双向链表则不同，已知一个结点可以找到链表中的所有结点。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. #define SIZE 3
5. #define NAME 8
6. typedef struct
7. {
8. char name[NAME];
9. int ID;
10. }student;
11. typedef struct node
12. {
13. student data;
14. struct node \*prev;
15. struct node \*next;
16. }node;
17. typedef struct
18. {
19. node \*head;
20. node \*tail;
21. }linkList;
22. linkList \*createLinkList();
23. void deleteLinkList(linkList \*);
24. void pushBack(linkList \*, student);
25. int main()
26. {
27. linkList \*l = createLinkList();
28. student s;
29. char name[SIZE][NAME] = { "张三","李四","王五" };
30. for (int i = 0; i < SIZE; i++)
31. {
32. s.ID = i + 1;
33. strcpy(s.name, name[i]);
34. pushBack(l, s);
35. }
36. for (node \*t = l->head; t != NULL; t = t->next)
37. {
38. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
39. }
40. deleteLinkList(l);
41. getchar();
42. return 0;
43. }
44. linkList \*createLinkList()
45. {
46. linkList \*l = malloc(sizeof(linkList));
47. l->head = NULL;
48. l->tail = NULL;
49. return l;
50. }
51. void deleteLinkList(linkList \*l)
52. {
53. node \*t, \*p;
54. for (t = l->head; t != NULL;)
55. {
56. p = t->next;
57. free(t);
58. t = p;
59. }
60. free(l);
61. }
62. void pushBack(linkList \*l, student data)
63. {
64. node \*t = malloc(sizeof(node));
65. t->data = data;
66. t->prev = NULL;
67. t->next = NULL;
68. if (l->head == NULL)
69. {
70. l->head = t;
71. l->tail = t;
72. }
73. else
74. {
75. l->tail->next = t;
76. t->prev = l->tail;
77. l->tail = t;
78. }
79. }

程序8-7-9

程序的运行结果如下：

学号：1 姓名：张三

学号：2 姓名：李四

学号：3 姓名：王五

这个程序看起来和程序8-7-3没有什么区别，这是因为它只涉及了双向链表的简单操作。不难看出，我们在定义struct node的时候，往里面增加了一个prev的指针，它的作用就是指向当前结点的前驱结点，多了这个指针会多很多麻烦。在单向链表中，如果要在链表尾部插入一个结点，那么我们只需让表尾结点的前驱结点的next指向它即可，不必对它进行什么操作，而双向链表中则不同，我们除了要让tail->next指向目标结点，还要让t->prev指向它的前驱结点，这才有了第75到77行的代码，只有这样做，才能保证链表中有两条“通路”。

对于双向链表来说，链表的各种操作中最简单的就是从表尾插入结点了，在任意位置插入和删除结点要复杂很多，因为需要同时设置3个结点，每个结点又有2个指针，这些指针都需要分别进行设置。但是不管怎样，双向链表和单向链表的各种操作的原理是一致的，系统地介绍双向链表会占用大量的篇幅，所以我们只介绍了双向链表的基本概念，有兴趣的读者可以自行研究或参考数据结构的相关书籍。事实上我们也曾说过，链表本身不属于C语言基础，它是数据结构所研究的内容，深入学习也有一定的难度，所以我们点到为止，不再加大学习负担。

接下来再看看循环链表，循环链表中，尾结点的next指针不再是NULL，而是指向head，也就是说，尾结点的后继结点是头结点。循环链表的好处是通过任何一个结点都能访问到链表中的所有结点，即使链表本身是单向链表。

1. #include <stdio.h>
2. #include <string.h>
3. #include <stdlib.h>
4. #include <conio.h>
5. #define SIZE 3
6. #define NAME 8
7. typedef struct
8. {
9. char name[NAME];
10. int ID;
11. }student;
12. typedef struct node
13. {
14. student data;
15. struct node \*next;
16. }node;
17. typedef struct
18. {
19. node \*head;
20. node \*tail;
21. }linkList;
22. linkList \*createLinkList();
23. void deleteLinkList(linkList \*);
24. void pushBack(linkList \*, student);
25. int main()
26. {
27. linkList \*l = createLinkList();
28. student s;
29. char name[SIZE][NAME] = { "张三","李四","王五" };
30. for (int i = 0; i < SIZE; i++)
31. {
32. s.ID = i + 1;
33. strcpy(s.name, name[i]);
34. pushBack(l, s);
35. }
36. for (node \*t = l->head; ; )
37. {
38. printf("学号：%d 姓名：%s\n", t->data.ID, t->data.name);
39. t = t->next;
40. if (t == l->head)
41. {
42. break;
43. }
44. }
45. deleteLinkList(l);
46. getchar();
47. return 0;
48. }
49. linkList \*createLinkList()
50. {
51. linkList \*l = malloc(sizeof(linkList));
52. l->head = NULL;
53. l->tail = NULL;
54. return l;
55. }
56. void deleteLinkList(linkList \*l)
57. {
58. node \*t, \*p;
59. l->tail->next = NULL;
60. for (t = l->head; t != NULL;)
61. {
62. p = t->next;
63. free(t);
64. t = p;
65. }
66. free(l);
67. }
68. void pushBack(linkList \*l, student data)
69. {
70. node \*t = malloc(sizeof(node));
71. t->data = data;
72. t->next = NULL;
73. if (l->head == NULL)
74. {
75. l->head = t;
76. l->tail = t;
77. t->next = l->head;
78. }
79. else
80. {
81. l->tail->next = t;
82. l->tail = t;
83. t->next = l->head;
84. }
85. }

程序8-7-10

程序的运行结果如下：

学号：1 姓名：张三

学号：2 姓名：李四

学号：3 姓名：王五

程序8-7-10比起程序8-7-9还要复杂一些，但这不代表循环链表本身比双向链表复杂。循环链表也分为单向循环链表和双向循环链表，我们介绍的是单向循环链表。程序8-7-10中，有几个需要注意的地方。

从尾部插入结点的时候，除了要将尾结点设置正确（即将tail设置正确），还需要设置尾结点的next，使其指向头结点，所以有了第73到84行。我们可以看出，当链表为空的时候，我们就将目标结点作为首结点加入链表，同时让目标结点的next指向它自身，这也算是循环；而如果链表中已经有了其它结点，那么我们就将目标结点当成尾结点加入链表，然后再让目标结点的next指向首结点。

在执行遍历操作的时候，也需要注意，循环结束条件不再是t!=NULL，而是t!=head，可是这样还有一个问题，那就是如果我们把t!=head设为循环继续的条件，那么循环根本就不会执行。所以我们只好把循环继续的条件放到了循环体中，需要注意的是，程序8-7-10中的第36到44行代码并不是通用的遍历代码，一般情况下，我们应该对链表是否为空加以判断，只是在这个地方我们能确定链表不为空，所以也就没有判断。

最后就是销毁链表的操作，最简单的做法就是把循环链表改造成一个单向链表（即让尾结点的next从指向head变成指向NULL），然后再把链表当成单向链表来处理。这是最合理最简洁的方式。

关于循环链表，我们也仅做简单介绍，深入讲解没有太多价值，因为实际应用中，链表总要和教材上有所出入，以此来达到增加效率的目的，读者同样可以参考数据结构的相关书籍来进一步学习循环链表。事实上，如果读者对单向链表的各种操作已经有了很深的理解，那么解决双向链表、循环链表甚至是双向循环链表都是不在话下的，毕竟它们有着相似的原理。所以，在学习的时候一定要注意区分，什么是精髓，什么是陪衬，如果将目标放在程序设计的思想上，那么就很容易举一反三；如果拘泥于某个程序或某段代码，就会发现C语言很难学习，这样不仅不利于以后学习其它编程语言，而且面对C语言的变化也会很吃力。

结构体和指针一样，也是C语言最重要的内容之一，结构体带来了一定的封装以及抽象的思想。对于计算机来讲，是否使用结构体没有什么区别，但是对于人类来讲，使不使用结构体的区别在于是否使用接近人类的思维模式来进行程序设计，可以说，程序设计的思想越接近人类，开发的速度越快，学习也就越简单，这也就是为什么面向对象是程序设计语言的主流。而C语言作为面向过程的语言，更多地用于底层开发而不是常用的应用层开发，同时，C语言也是最难精通的语言之一。

练习题

1. 编写一个course结构体来表示课程，该结构体有2个成员，分别是name、date，其中name表示该课程的名字，date则表示该课程是星期几的课，其中date的取值从Sunday到Saturday，是一个枚举类型，要求实例化course结构体，并接受用户输入的输入来为实例变量赋值。

2. 任何数据结构的基本操作都是：增、删、查、改，对于单向链表来说，我们已经介绍了相对复杂的增和删，查找相对简单很多。编写程序，实现链表的查找功能，程序自己建立一个链表，然后提示用户输入要查找的结点的数据域，最后通过一个函数来实现查找功能，查找成功则返回目标结点的指针，否则返回NULL，并输出提示信息提示用户是否查找成功。

3.有时候，我们需要对链表中的各个结点进行排序，排序的原理和数组基本类似。编写程序，实现链表的排序，程序任意建立一个链表，并以结点的某个成员为依据进行排序（如根据成绩来对student进行排序），排序方式不限，选择排序、冒泡排序、快速排序均可。