- 结构与联合
  - 结构概述
  - 结构的声明和引用
    - 结构类型的声明
    - 结构变量的定义
    - 结构变量的初始化
    - 点运算符
    - 嵌套的结构
    - 结构的大小
      - 内存对齐
  - 结构数组
    - 结构数组的定义
    - 结构数组的初始化
    - 结构数组作函数参数
  - 指向结构的指针
    - 结构指针的声明
    - 箭头运算符
    - 结构数组的指针表示
  - 结构与函数
    - 结构或结构指针作为函数参数
    - 结构或结构指针作函数返回值
  - 联合
  - 结构指针的应用
    - 静态与动态数据结构
    - 单链表的结构
    - 单链表的建立和输出
      - 后进先出链表或"栈"
      - 先进先出链表或"队列"
      - 递归建立先进先出链表
      - 通过二级指针参数返回头指针值
    - 单链表的基本操作
      - 查找结点
      - 逆序输出
      - 删除结点
      - 插入结点
      - 归并链表
    - 单链表排序
      - 交换结点数据域的排序方法

- 交换结点的单链表排序

结构与联合

结构概述

数组是一种**构造类型**,数组中所有元素必须是相同类型的变量,适合描述不同对象的同一属性.此外,C语言提供了可以由**不同数据类型**组合而成的结构类型.
结构就是**一种可以将不同类型数据组织成一个整体的结构类型**,这些数据称为结构的成员.先以一个简单的程序为例:

```
#include <stdio.h>
struct Planet {/* 声明结构类型 */
    char name[20];/* 行星名称 */
    float diameter;/* 行星直径 */
    int moons;/* 卫星数 */
};
int main() {
    struct Planet planet;/* 定义结构类型变量 */
    printf("请输入行星的名称、直径（单位：千米）和卫星数：\n");
    scanf("%s %f %d", planet.name, &planet.diameter, &planet.moons);
    printf("\n您输入的行星信息如下：\n");
    printf("名称：%s\n直径：%.2f 千米\n卫星数：%d\n", planet.name, planet.diameter, planet.moons);
    return 0;
}
```

程序使用了结构的三个重要内容:

1. 构造了描述行星的**结构类型struct Planet**,它有三个成员;
2. 定义了一个Planet**结构变量planet**,表示一颗行星,变量有Planet类型声明中指定的成员;
3. 用点运算符访问结构变量的成员.

结构的声明和引用

结构类型的声明

声明结构类型的一般形式为:

```
struct 结构类型名 {
    // 成员变量声明表
    数据类型 成员变量1;
    数据类型 成员变量2;
    // ...
};
```

struct是声明结构类型时必须使用的关键字,结构类型名是标志该结构类型的名字,用以区分其他数据结构,二者联合起来构成一个**类型说明符**.花括号中是该结构成员,结构成员的声明方式与变量和数组的定义方式相同,但不能初始化.成员的数据类型可以是除本结构类型以外的其他任何类型.结构类型声明以分号结束.

结构变量的定义

结构类型的声明仅仅描述了该结构的格式,系统不会为其分配实际的内存单元.为了在程序中使用结构,需要定义结构类型的变量,即**结构变量**.

1. **先声明结构类型,再定义结构变量**

```
struct Planet {/* 声明结构类型 */
    char name[20];/* 行星名称 */
    float diameter;/* 行星直径 */
    int moons;/* 卫星数 */
}
struct Planet inner,outer;
```

inner和outer是planet结构类型的两个变量,具有**存储单元**,可以被初始化.

2. **声明结构类型的同时定义结构变量**

```
struct Planet {/* 声明结构类型 */
    char name[20];/* 行星名称 */
    float diameter;/* 行星直径 */
    int moons;/* 卫星数 */
} inner,outer;
```

这种情况下,结构名Planet是可选的,如果以后的程序中不再使用该结构类型的声明,则可以不用写结构类型名.

3. 可以通过存储类型修饰符对结构变量的存储类型进行修饰. `static struct Planet  inner;` .inner为Planet类型的静态结构变量,此时其成员都具有静态变量性质.

4. 结构变量名不同于数组名,它表示的是一个结构体,它包含了结构体中定义的各个成员的值,并不直接代表结构变量的地址,而是代表了整个结构体对象.
   声明一个结构体变量时,编译器会为该变量分配内存空间,以存储结构体中每个成员的值.**结构体变量的地址是该内存空间的起始地址**,而不是结构变量名本身.
   要获取结构变量的地址,可以使用取**地址操作符**.例如,如果 `person` 是一个结构体变量,则 `&person` 将返回该结构体变量的地址.

结构变量的初始化

初值是由常量表达式组成的初值表,初值的**类型与顺序**要和结构成员一致.

```
struct Planet {/* 声明结构类型 */
    char name[20];/* 行星名称 */
    float diameter;/* 行星直径 */
    int moons;/* 卫星数 */
} x={"Jupiter",142987,79};
```

点运算符

结构变量不能作为一个整体进行输入输出,只能对结构变量名的成员进行输入输出.对结构变量中的成员的访问形式为:

结构变量名.成员名

其中．是**结构成员运算符**,其左操作数是结构变量,右操作数是结构成员,它具有最高优先级.
为了给结构变量中的成员赋值,除了前面 `struct Planet{char name[20]...} x={"Jupiter",...}` 的初始化方法,还可以使用下列方法:

- `scanf("%s",x.name);`
- `strcpy(x.name,"Jupiter");`
- `x.name[0] = 'J';//逐个字符赋值;`
- `struct Planet x={.name="Jupiter",...};`
  但是要注意, `x.name="Jupiter";` 这种赋值方法是错误的,因为数组名是地址常量,不能修改.在这种语句中,字符串字面量"Jupiter"存储在程序的常量区中.类似
  地, `struct Planet x; x={.name = "Jupiter"};` 也是错误的.

类似于 `char name[]="Jupiter";` 和 `char name[20];name="Jupiter";` ,前者是正确的赋值方式而后者则是错误的.但是 `char *p;p = "Jupiter";` 是正确的,因为p是指针变量.

嵌套的结构

含有结构成员的结构称为嵌套结构.

```
struct data{
    int month;
    int day;
    int year;
};
struct Planet{
    char name[];
    double diameter;
    int moons;
    struct data find_data;
};
```

data是日期结构类型Planet结构的最后一个成员find_data是data结构类型,所以Planet是一个嵌套的结构. `struct Planet mid;` 声明了一个Planet结构变量,如果成员本身是结构类型,则要用若干结构成员运算符逐级访问,引用到最低一级的成员.

```
scanf("%d %d %d",&mid.find_data.year,&mid.find_data.month,&mid.find_data.day);
```

结构的大小

结构所占存储空间的大小和**内存地址的对齐问题**有关,内存对齐是编译器为了便于CPU快速访问而采用的一项技术.

内存对齐

CPU读写内存数据时,并不能从任意地址开始,都是**从某个特定的地址开始,以字为一个块来操作**.对于32位x86系统,如果编译器可以保证int型数据都从4倍数地址开始存放,那么读写一个int型数据就只要一次内存操作,而short型数据从二倍数地址开始存放也只需要访问一次.否则可能要两次访问操作,因为数据可能横跨两个四字节内存块时.
正是由于CPU只能在**特定的地址处读取数据**,为了提高读取效率,就要求**数据在内存中的地址是某个整数k的倍数**,这就是内存对齐.k就称为该数据类型的对齐模数(对齐系数).变量在内存中的对齐可以加快读写系速度,提高程序性能.

在Win64平台上,内存对齐规则通常遵循CPU的要求.在x86-64架构上,内存对齐通常是按照数据类型的大小进行的,以确保数据能够以最有效的方式被处理器访问.内存对齐规则大致如下:

1. **基本类型的对齐规则**:
    - 基本数据类型T的对齐模数就是sizeof(T).
    - short类型数据通常按2字节对齐,int和float类型数据通常按4字节对齐.long,double和指针类型通常按8字节对齐.char型数据则可以从任何一个地址开始.

2. **结构体对齐规则**:
   结构体对齐规则是指编译器在存储结构体类型的实例时如何对内存进行布局的规则。这些规则旨在确保结构体成员的访问是高效的，并且遵循硬件的内存访问要求。
   下面是结构体对齐规则的详细解释:
   i. **对齐基准**:
      - 结构体的对齐通常是按照结构体中的最大成员大小来确定的.编译器会将结构体的起始地址设置为其最大成员大小的倍数,以保证结构体中每个成员都能够按照其自身大小正确对齐.
      - 结构中第一个成员的地址等于结构变量的地址.
   ii. **成员对齐**:
      - 每个结构体成员都会按照其自身大小进行对齐.如果一个结构体成员是一个整型,通常会按照4字节对齐;如果是一个双精度浮点数,通常会按照8字节对齐.
      - 如果结构体成员的大小正好是基本对齐大小,则该成员会紧密地排列在结构体的布局中.
   iii. **填充字节**:
      - 为了保证结构体成员的对齐,编译器可能会在结构体成员之间插入**填充字节**(padding).这些填充字节通常是未使用的,但是它们确保了结构体成员的正确对齐.
      - 填充字节的数量取决于结构体成员之间的大小差异以及编译器的实现策略.
   iv. **结构体大小**:
      - 结构体的大小通常是其成员大小的总和,加上填充字节的大小,由于填充字节的存在,结构体的大小可能会大于其成员的总和.
      - 有时,编译器提供了选项来控制结构体的对齐方式,以最小化填充字节的数量,从而减小结构体的大小.

结构数组

以同类型的结构变量为元素的数组称为结构数组,结构数组适合描述通讯录,人员,登记表,清单以及编译程序处理的符号表等二维表格数据.

结构数组的定义

```
/* 描述30个学生情况的登记表 */
struct stud{
    char studentId[12];
    char name[9];
    char gender;
    struct date birthday;
    float score;
};
struct stud student[30];
```

数组student有30个元素,每个元素都是一个stud结构类型,可以描述一个学生的状况,整个数组可以描述30个学生的情况.
引用结构数组的元素同引用普通数组元素的形式一样: `student[i]` 是结构数组的第i个元素,等同于一个结构变量名的作用.

```
strcpy(student[0].studentId,"202210413");/* 不能写为student[0].studentId="202210413" */
student[0].gender='M';
student[0].birthday.day=16;
scanf("%f",&student[0].score);
```

结构数组的初始化

1. 使用数组初始化器

```
struct stud student[]{
    {"111111","AAA",'M',{9,9,9},0},
    {"222222","BBB",'F',{8,8,8},0},
    {"333333","CCC",'M',{7,7,7},0}
};
```

- student被声明为含有三个元素的结构数组,每个元素是一个stud结构类型.**整个数组**的初值由最外层的一对花括号界定,**每个元素**作为结构,其初值再用一个花括号括起来,由于**数组元素中的成员**birthday是date结构类型,其初值仍然要一个花括号括起来.
- 事实上,内层花括号只是为了增强初值列表可读性,可以使初值与元素的对应关系更准确,但是内层的{}是不必要的.

1. 使用索引初始化

```
struct stud student[]{
    [0]={"111111","AAA",'M',{9,9,9},0},
    [2]={"333333","CCC",'M',{7,7,7},0}
};
strcpy(student[1].studentId,"222222");
/* ...... */
```

结构数组作函数参数

将结构数组作为函数参数传递时,可以选择**传递数组的指针**或者**直接传递整个数组**.下面是一个使用结构数组作为函数参数的例子,它分别演示了这两种方法:

```c
#include <stdio.h>
#include <string.h>
#define MAX_SIZE 3
// 定义一个结构体,必须在主函数之外作为全局变量
struct Person {
    char name[50];
    int age;
};
int main() {
    struct Person people[MAX_SIZE] = {
        {"Alice", 25},
        {"Bob", 30},
        {"Charlie", 35}
    };
    // 调用函数，传递结构数组的指针作为参数
    printPeoplePtr(people, MAX_SIZE);
    // 调用函数，直接传递整个结构数组作为参数
    printPeopleArr(people, MAX_SIZE);
    return 0;
}
// 第一种方法：将结构数组的指针作为参数传递
void printPeoplePtr(struct Person *people, int size) {
    printf("People (passed by pointer):\n");
    for (int i = 0; i < size; ++i) {
        printf("Name: %s, Age: %d\n", people[i].name, people[i].age);
    }
}
// 第二种方法：直接将结构数组作为参数传递
void printPeopleArr(struct Person people[], int size) {
    printf("People (passed by array):\n");
    for (int i = 0; i < size; ++i) {
        printf("Name: %s, Age: %d\n", people[i].name, people[i].age);
    }
}
```

在这个例子中,我们定义了一个 `Person` 结构体,它有两个成员: `name` 和 `age` .然后我们在主函数中创建了一个Person结构类型的 `people` 数组,其中存储了三个 `Person` 结构体的实例.我们调用了两个函数: `printPeoplePtr()` 和 `printPeopleArr()` ,它们分别使用了结构数组的指针和整个结构数组作为参数.值得注意的是,结构数组名本身就是指针常量,而一般的结构变量名并不是的地址.

```
struct win
{
    int num;            /* 窗口号 */
    int x1, x2, y1, y2; /* 窗口坐标 */
};
int order[N];
int clickWin(struct win w[], int n)
{
    int x, y, top = -1;
    int i, c, k;
    scanf("%d %d", &x, &y); /* 输入鼠标单击处的坐标 */
    for (i = 0; i < n; i++)
    { /* 从最顶层窗口开始遍历，判断哪个窗口被单击 */
        c = order[i]; /* 纪录窗口索引 */
        if (w[c].x1 <= x && x <= w[c].x2 && w[c].y1 <= y && y <= w[c].y2)
        {                           /* 单击在窗口内 */
            top = w[c].num;         /* 记录单击的窗口号 */
            for (k = i; k > 0; k--) /* 移动被单击的窗口到最前端 */
                order[k] = order[k - 1];
            order[0] = c;
            break;
        }
    }
    return top; /* 返回窗口号 */
}
```

上述程序的核心在于窗口的索引与遍历,通过数组 `order[N]` 我们将描述窗口的结构数组与编号与顺序联系起来.order的下标表示窗口显示的顺序,order的元素表示结构数组元素中的窗口号成员.

指向结构的指针

结构指针的声明

结构指针变量的声明形式,除了在变量名前加上指针说明符*之外,其余同结构变量的声明一样.设data是已经定义的日期结构类型,若要使指针p指向一个data结构变量birthday,需要下列声明和赋值语句:
`struct data birthday,*p; p=&birthday;` .在指针声明时也可进行初始化, `struct data birthday,*p=&birthday;` .
birthday是一个结构变量,p是指向结构变量birthday的指针,与普通变量一样,**&birthday表示结构变量birthday所占据的内存的起始地址,是一个结构指针常量**.
*p是间接访问birthday,*p与birthday等价,可以有下列合法语句:

`(*p).year=2004;` , `scanf("%d",&(*p).month);` , `strcpy((*p).constellation,"sagittarius");` .
`(*p).year` 不能写为 `*p.year` 因为结构成员运算符"."优先级高于间接访问运算符"*".

箭头运算符

结构指针在程序中使用的很平凡,为了简化引用形式,C语言提供了另一个结构成员运算符 `->` ,其一般使用形式为: 结构指针->成员名 .
箭头运算符具有最高优先级,永远解释为一个整体. `->` 和 `.` 运算符的结合性是左结合, `p->find_day.day` 等价于 `(p->find_day).day` .由于find_day是data结构类型,所以表达式 `p->find_data` 的类型是data类型.

```
struct Planet inner,*p=&inner;
p->moons;/* 等价于(*p).moons */
p->find_data.day;/* 等价于(*p).find_data.day */
```

同时出现 `->` , `*` 和 `++` 时,注意 `->` 优先级最高, `*` 和 `++` 同属于第二优先级,结合性为右结合.

结构数组的指针表示

结构数组的元素除了用下标表示,还可以用数组名(指针常量)和指针变量表示.

```
struct stud{
    char studentId[12];
    char name[9];
    char gender;
    struct date birthday;
    float score;
};
struct stud student[]{
    {"111111","AAA','M',{9,9,9},0},
    {"222222","BBB','F',{8,8,8},0},
    {"333333","CCC','M',{7,7,7},0}
};
struct stud *p=student;
/* -------------------------------- */
p->gender;//相当于student[0].gender
(p+1)->studentId;//相当于student[1].studentId
student->birthday.day;//相当于student[0].birthday.day
(student+2)->name[0];//相当于student[2].name[0]
*(p->name+1);//相当于student[0].name[1]
```

结构数组作为函数参数时,实际上**传递的是结构数组的第0个元素的地址**,对应的形参被解释为指向数组元素的指针(结构指针),实参可以是结构数组名,也可以是指向结构数组中首元素的指针变量.
函数定义 `int clickWin(struct win w[], int n);` 可以改写为 `int clickWin(struct win *w, int n)` .
`if (w[c].x1 <= x && x <= w[c].x2 && w[c].y1 <= y && y <= w[c].y2)` 可以改写为 `if ((w+c)->x1 <= x && x <= (w+c)->x2 && (w+c)->y1 <= y && y <= (w+c)->y2)`

结构与函数

结构变量或结构指针都可以作为函数的参数,也可以作为**函数的返回值**.

结构或结构指针作为函数参数

1. 结构成员做参数
   传各个结构成员,形参的类型与结构成员类型一致,用法和普通变量做函数参数没有区别.属于**值的传递**.
2. 结构变量做参数
   将实参结构的内容整体复制到形参,形参必须是同类型的结构变量,这种方式也属于**值传递方式**,**形参是实参的副本**,在被调用函数中形参结构的修改不影响调用函数中实参结构的值,适合于不想改变实参结构的场合.
3. 结构指针做参数
   将**实参结构变量的首地址**传给形参,属于传地址方式,在被调用函数中通过指针区访问调用函数中结构变量的值,这些值可以修改.

```
/* 判断单击是否在窗口中 */
int inWindow(struct win w,int a,int b)
{
    if (w.x1 <= a && a <= w.x2 && w.y1 <= b && b <= w.y2)
        return 1;
    return 0;
}
```

inWindow函数的形参是win结构,实参是机构数组的元素w[c],类型也是win结构.调用时,系统给形参分配sizeof(struct win)字节的存储空间,占用的存储空间取决于win结构的大小,并**将实参结构w[c]整体复制到形参x中,相当于对应成员一一赋值.

```
int inWindow(struct win *p,int a,int b)
{
    if (p->x1 <= a && a <= p->x2 && p->y1 <= b && b <= p->y2)
        return 1;
    return 0;
}
/* inWindow的调用形式 */
if(inWindow(w+c,x,y))
```

形参p是指向win结构的指针,实参w+c等价于&w[c].==形参p占用的空间是 `sizeof(struct win *)` ,和win结构的大小无关.

结构或结构指针作函数返回值

函数原型的一般形式为:

```
struct 结构名 函数名(形参列表);
```

```
struct win getWindow(){
    struct win a;
    static int num;
    scanf("%d %d %d %d", &a.x1, &a.y1, &a.x2, &a.y2); /* 输入窗口坐标 */
    a.num=++num;
    return a;
}
/* 调用函数时 */
w[i]=getWindow();
```

getWindow的返回值是win结构类型a,a被整体赋值给同类型结构变量w[i].C允许将两个同类型的结构变量直接赋值操作,但是结构变量定义时的初始化的形式不能用于赋值语句.

返回结构指针的函数的一般形式为:

```
struct 结构类型名 *函数名(形参列表);
```

```c
#include <stdio.h>
#include <stdlib.h>
// 定义一个结构体
typedef struct Point {
    int x;
    int y;
} POINTER;
// 返回一个Point结构体指针的函数
POINTER* createPoint(int x, int y) {
    // 分配内存来存储Point结构体
    POINTER* newPoint = (POINTER*)malloc(sizeof(POINTER));
    // 检查内存是否成功分配
    if (newPoint == NULL) {
        printf("内存分配失败\n");
        exit(1);
    }
    newPoint->x = x;newPoint->y = y;
    return newPoint;
}
int main() {
    // 调用函数创建一个Point结构体对象
    POINTER* p = createPoint(3, 5);
    printf("创建的Point结构体对象的值:(%d, %d)\n", p->x, p->y);
    // 释放动态分配的内存
    free(p);
    return 0;
}
```

联合

与结构类似,联合也是一种构造类型,是多个成员的集合.除了用关键字union取代struct之外,联合类型的声明,联合变量的声明,联合变量的定义和联合成员的引用在语法上与结构完全一样.联合的主要特点是**共享内存**,结构的成员占据各自不同的空间,而联合的所有成员共享一段内存,**所有成员的首地址都是一样的**,联合的大小由所占字节数最大的成员决定.

```c
union utag{
    int iVal;
    double dVal;
    char cVal;
} u;
```

utag是联合类型的标志(联合名),有时可以省略.联合变量u在不同时刻可以拥有int,double和char中任一个,编译程序按联合成员中最长的类型为联合变量分配存储空间,即sizeof(union utag)和sizeof(u)都等于8.**联合的所有成员都从低地址,即联合的起始地址开始存放**.

联合的初始化和结构初始化十分类似,不过C允许通过指定成员实现对联合变量的任意元素的进行初始化.即 `union Data data1 = {.i = 10};` ,若是结构 `struct Data data1 = {.i = 10};` 则默认结构中的其他元素为0或NULL.

```
#include <stdio.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main() {
    union Data data1 = {.i = 10};
    union Data data2; data2.f = 3.14;
    union Data data3; strcpy(data3.str, "Hello");
    union Data data4 = {10};//默认对对一个成员初始化
    return 0;
}
```

联合所有成员的地址和联合变量的地址相同,但他们**不是同一类型的指针**.

联合成员的引用也和结构成员的引用一致,例如, `union utag *p=&u;` ,则用联合变量和联合指针都可以引用u中的成员,例如, `u.dVal` 等价于 `p->dVal` .

联合成员不是并存的,任意时刻联合变量中只含有一个成员,该成员是最近一次存入的,称为**当前成员**.如果误将一个类型作为另一个类型解释,其结果与数据在内存中的二进制存储格式有关.例如:

```
u.iVal=0x3041;
printf("%c",u.cVal);
printf("%f",u.dVal);
```

利用联合的上述性质可以用下列程序交换高低字节:

```
union{
    short n;
    char a[2];
} x={0x2030};
int low,high;
low=x.a[0];
high=x.a[1];
```

联合的成员可以是结构和数组,数组的元素和结构的成员也可以是联合.

结构指针的应用

静态与动态数据结构

基本类型和构造类型的数据都属于静态数据,他们所占的存储空间大小是在**变量定义时确定的**,程序执行过程中不能改变,称之为**静态数据结构**.动态数组也是伪动态,因为在创建数组时,数组的规模必须有确定的值,之后**n的值发生变化,数组大小不会改变**.静态数据结构的特点是由**系统分配固定大小的连续存储空间**,以后在程序运行过程中,存储空间的位置和容量不会再改变,访问静态数据结构可以用**变量的名字也可以用指向变量的指针**.

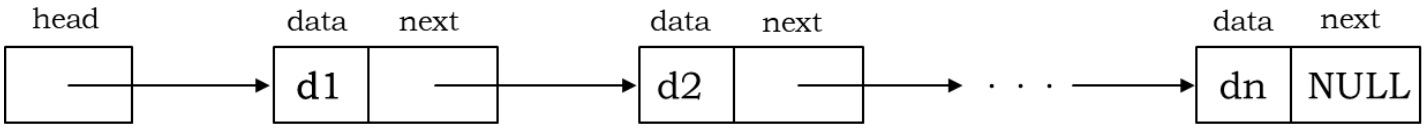
而动态数据结构可以满足任意规模数据的存储.动态数据结构是在程序运行过程中逐步建立起来的,其存储空间在程序执行过程中,由系统提供的**内存分配函数动态分配**,即动态数据结构所占用的存储空间的大小在程序执行过程中可以改变.程序运行过程中,数据量增加就向系统申请新的空间;数据量减少就将多余空间归还给系统.动态分配的空间没有名字,只有指针指向它,因此**对动态数据结构的访问只能通过指针进行**.
动态数据结构通常由称为"结点"的元素组成,每个结点是相同类型的一个结构变量,这些结点根据需要动态建立,他们的内存位置不连续,结点与结点之间**通过指针连接成一个数据整体**.每个结点由**数据域和指针域**组成,数据域用来存放所描述对象的相关信息,指针域用来存放描述对象的地址,是**指向结点结构的指针**,访问动态数据结构的元素只能通过指针进行.

动态数据类型的每个结点是相同类型的结构,结构中**含有指向结构自身的地址**,称这样的结构为自引用结构.一个结点中可以含有一个或多个指向结构自身的指针,分别指向不同的结点,因而,结点与结点之间可以连接成不同形式的动态数据结构.

单链表的结构

链表由**一系列包含数据域和指针域分结点组成**,结点的指针域如果只包含一个指向后一结点的指针,这种链表称为单向链表.

整个链表需要一个**头指针**,里面存放表中第一个结点的地址,即头指针指向链表的第一个元素,最后一个结点的指针不指向任何元素,用空指针NULL表示,该结点称为**尾节点(链尾)**.



上图中,head是头指针,data表示结点的数据域,next表示结点的指针域.利用头指针可以追踪到一个指定的元素.
例如引用第一个结点中的数据应写为: `head ->data` ;
引用第二个结点中的数据应写为: `head->next->data` ,其中头指针head指向第一个结点, `head->next` 是引用第一个结点的指针域,它指向第二个结点, `head->next->data` 就是第二个结点的数据域.利用链尾的NULL可以判断整个链是否结束.

链表中各个元素在内存中不一定连续,通过结点的指针将这些物理上不连续的元素按逻辑顺序连接起来,因此单链表是一种**链式存储的线性表**.

对于单链表,首先要用结构类型描述一个结点,链表的结构类型可以声明为:

```
struct intNode{
    int data;
    struct intNode *next;/* 指向下一结点的指针 */
};
```

在intNode结构中,next是指向intNode结构的指针,称为指向自身的指针,因此struct intNode结构是自引用结构类型.
如果di描述商品信息,可以先声明描述商品的goods结构,再声明描述链表结点的goodsNode结构,代码为:

```
struct goods{
    int code;
    char name[20];
    float price;
};
struct goodsNode{
    struct goods data;
    struct goodsNode *next;/* 指向下一节点的指针 */
};
```

单链表的建立和输出

链表的建立是指在程序执行过程中,在内存中声明一个链表,需要编码实现,包括定义头指针,给结点动态分配存储空间,输入各结点数据,通过指针将各节点连接起来,使头指针指向第一个结点,最后一个结点的指针域为NULL.

建立链表的关键是**建立结点并建立结点的连接关系**,依据新结点插入位置的不同,建立单链表有下列两种发生.

后进先出链表或"栈"

每次生成的新结点总是插入当前列表的表头作为首结点,使链表从链头到链尾的结点排列顺序和数的输入顺序相反,即最先建立的结点是尾结点,最后建立的结点是首结点,这种方法也叫作**头插法**,依次将元素放到最前面.

以输入一批整数,建成一个后进先出的单链表为例,建立链表的步骤为:

1. 为新结点分配存储空间,并使p指向该结点.
   ```
   p=(struct intNode *)malloc(sizeof(struct intNode));
   ```
2. 输入数据,并存入新结点的数据域.
   ```
   scanf("%d",&p->data);
   ```
3. 用头插法将新结点加入链中,使新结点的指针域指向当前的第一个结点,头指针指向新结点.
   ```
   p->next=head;
   head=p;
   ```
   p始终指向新结点,head始终指向第一个结点,链中每个元素的next依次指向上一次建立的结点,链尾是最先建立的结点.

链表建立后,标志链表位置的是头指针,头指针起着链表名的作用,所以建立链表的函数应该返回头指针,即返回第一个节点的首地址.

先进先出链表或"队列"

建立链表时,将每次生成的新结点插入当前链表的尾部作为尾结点,使链表中从链头到链尾的结点排列顺序和输入顺序相同,最先建立的结点是首结点,最后建立的是尾结点.这种方法称为**头插法**,依次将元素放到最后面.

```
struct intNode *head,*tail,*p;
    int x;
    head=NULL;/* 链表开始为空 */
    tail=NULL;/* 尾指针初值为空 */
    scanf("%d",&x);
    while(x){
        p=(struct intNode *)malloc(sizeof(struct intNode));/* 生成新结点 */
        p->data=x;
        if(head=NULL)
            head=p;/* 首结点插入空表 */
        else
            tail->next=p;/* 其余结点插入尾部 */
        tail=p;/* 新结点作为尾结点 */
        scanf("%d",&x);
    }
    if(tail!=NULL)/* 对于非空表,将尾结点指针域置空 */
        tail->next=NULL;
    return head;
```

递归建立先进先出链表

单链表可以看成是递归定义的:一个结点后面再跟着其指针指向的一个子链表或者空表,可以通过递归实现单链表的各种操作.

```
struct intNode *createList2()
{
    struct intNode *head,*p;
    int x;
    head=NULL;
    scanf("%d",&x);
    if(x){
        p=(struct intNode*)malloc(sizeof(struct intNode));
        p->data=x;
        head=p;
        p->next=createList2()
    }
    return head;
}
```

递归结束后,会层层返回每一层头指针的值.

通过二级指针参数返回头指针值

可以用指针作为函数参数间接返回.要返回一个指针值,则参数必须是二级指针.

```
void createList4(struct intNode **head_p)
{
    struct intNode *p;
    int x;
    *head_p=NULL;
    if(x){
        p=(struct intNode*)malloc(sizeof(struct intNode));
        p->data=x;
        *head_p=p;
        createList4(&p->next);
    }
}
```

参数head_p是一个指向intN结构指针的指针,其作用是将函数创建的链表首结点地址返回给调用者.

单链表的基本操作

查找结点

```
struct intNode *searchNode(struct intNode *head,int x)
{
    struct intNode *p=head;
while(*p!=NULL&&p->data!=x)
    p=p->next;
return p;
}
```

逆序输出

```
void reversePrintList(struct intNode *head)
{
    struct intNode *p=head;
    if(p!=NULL){
        reversePrintList(p->next);
        printf("%d\t",p->data);
    }
}
```

删除结点

设被删节点是当前节点,则他的前一个结点称为直接前驱结点,后一个结点称为直接后继结点.删除结点就是使前驱结点直接指向后继结点,同时释放被删结点的存储空间.删除结点要用到两个临时指针,一个是遍历指针p,另一个是用于记住遍历过程中p的前驱结点的指针last,以便将拆成两段的结点连接起来.

1. 遍历链表,用遍历指针p从首结点开始遍历,找到要删除掉的结点: `while(p->data!=x&&p->next!=NULL){last=p;p=p->next;};`
2. 从链表中删去p指向的结点,需要区分被删的是链头还是非链头.如果被删的是链头,则修改头指针head,使它指向被删结点的后继: `head=p->next;` .否则修改被删结点的前驱(被last所指)的指针域,指向后继: `last->next=p->next;`
3. 释放p所指的存储区域: `free(p);`

插入结点

在链表中插入指针new指向的新结点时,先遍历找到插入点位置.

1. 插入位置位于链头: `new->next=head;head=new;` .
2. 插入位置位于链尾: `p->next=new;new->next=NULL;`
3. 插入位置位于两个结点之间时,同样需要两个临时指针,一个是遍历指针,一个是指针last,始终指向当前的前驱结点: `last->next=new;new->next=p;`
4. 具体程序见D:\C_code\C\Struct\Node\node_7\node_7(插入结点).c

归并链表

```
/* 将两个升序单链表归并成一个升序单链表 */
struct intNode *mergeList(struct intNode *head1,struct intNode *head2)
{
    struct intNode *head,*tail;/* head是新链表的头指针,tail是尾指针 */
    if(head1==NULL) return head2;
    if(head2==NULL) return head1;
    if(head1->data<head2->data){/* head指向两个链表中的最小元素 */
        head=head1;
        head1=head1->next;
    }
    else{
        head=head2;
        head2=head2->next;
    }
    tail=head;/* 尾指针始终指向新链表的 */
/* 将两个升序链表结点数据值一次对比,添加到新链表,同时改变原链表结点和头指针 */
    while(head1!=NULL&&head2!=NULL){
        if(head1->data<head2->data){
            tail->next=head1;
            head1=head1->next;
        }
        else{
            tail1->next=head2;
            head2=head2->next;
        }
        tail=tail->next;
    }
    if(head1!=NULL) tail->next=head1;
    else if(head2!=NULL) tail->next=head2;
    return head;
}
```

上述算法巧妙地将两个有序链表归并成了一个有序链表.也揭示了链表本质上就是将内存空间通过指针依次连接起来.

单链表排序

交换结点数据域的排序方法

```
struct intNode*p,*tail;
int tmp;
for(tail=NULL;head!=tail;tail=p){//基于冒泡排序,让tail所指的结点不断前移
    for(p=head;p->next!=tail;p=p->next)
        if(p->data>p->next->data){
            tmp=p->data;
            p->data=p->next->data;
            p->next->data=tmp;
        }
}
```

交换结点的单链表排序

交换节点主要改变结点的指针域,使结点的连接顺序发生改变.假设要交换p所指的结点和它的后继结点,指针last指向p的前驱结点.代码为:

```
last->next=p->next;
p->next=p->next->next;
last->next->next=p;
p=last->next;
```

在具体不带头结点的链表中,一般要新增头结点.新增头结点的好处有几个:

1. **简化插入操作**:在排序过程中,可能需要在链表头部插入新的节点.有了头结点,可以减少对头部插入的特殊情况的处理,并统一插入操作的逻辑.
2. **保持对链表头部的引用**:在排序过程中,节点的位置可能会改变,但需要保持对链表头部的引用.新增头结点可以确保排序后的链表始终有一个指向第一个节点的引用,而不必担心头部节点是否发生变化.