10 复合数据类型

除前面讨论的基本数据类型，与数组外，C语言还提供一些可自己定义的复合数据类型使用，包括枚举类型、结构体类型和联合体类型。

10.0 枚举类型

有时，一些变量的取值被限定在一个有限的范围内，比如一周只有7天，一年只有12个月。如果把这些量说明为整型、字符型或其他类型显然是不妥当的。为此，C语言中提供了一种称为“枚举”的类型。

枚举类型是通过枚举所以可取的值来定义一个有序的可数的值的集合。这个集中的元素是标识符，这些标识符实际上对应为一个整型常量。枚举类型是自定义数据类型，它的使用需要先定义类型，然后再用自定义的类型定义对应的变量。其类型声明的一般形式为：

```
enum 枚举类型名 {
    标识符[=整型常量],
    标识符[=整型常量],
    ...
    标识符[=整型常量],
};
```

其中，enum 是保留字，用来定义枚举类型，方括号内容为任选，如果不指定标识符对应的整型常量，那么按从 0 开始计数，后面每个自动加 1 。例如：

```
enum Days{Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

这个语句定义了一个枚举类型，名为 Days 。注意：Days 是自己定义的类型名，不是变量名。Days类型的取值范围就是一对大括号内所列举的那些值，即 Sun, Mon, Tue, Wed, Thu, Fri, Sat 。这些值形式上是标识符，但实际上对应的是整型的 0 到 6 。

在定义了枚举类型名后，就可以引用类型名定义枚举变量，一般形式为：

```
    enum 类型名 变量名表;
```

例如：

```
    enum Days start_day, end_day;
```

这个语句定义了两个枚举类型变量。当然，也可以在定义类型的同时定义变量，即：

```
    enum Days{Sun, Mon, Tue, Wed, Thu, Fri, Sat} start_day, end_day;
```

于是，可以引用这两个枚举类型变量。例如，对其赋值：

```
    start_day = Mon;
    end_day = Sat;
```

关于枚举类型，要注意：

- 任何枚举类型变量实际上是以整型的方式存储，这些符号常量的实际值也都是整数值。如果枚举定义时没有初始化，即省掉“=整型常数”时，则第一个“标识符”开始，他们的值分别为整数 0、1、2...
- 可以给某个或某几个“标识符”任意指定一个整数值。但当枚举中的某个标识符被初始化某个整数值后，其后的标识符按“依次加1”的规则确定其值。例如：

```
enum COLOR{A=200, B};
```

其中，B被自动赋值为201 。

- 枚举类型变量被当做整型变量看待，类型定义中的标识符被看成是整型变量。

10.1 结构体

有时，描述同一事物的不同数据属于不同类型的数据，这就不能简单是使用数据来存储数据，所以C语言中提供结构体（简称结构）类型来处理这种情况。结构体的一般形式为：

```
struct 结构体名称 {
      结构体成员1;
      结构体成员2;
      结构体成员3;
      ...
};
```

其中 struct 是关键字，用于声明结构体类型。同时结构体成员也可以是另一个结构体，但也不能递归定义结构类型。

与枚举类型类似，定义结构体变量（不同于结构体）的一般形式为：

```
struct 结构体名称 结构体变量名
```

有时为了程序的简洁或增强语句化，会使用 typedef 给

```
struct 结构体名称
```

取别名，来简化程序。

定义类型与定义变量也可以写在一起，例如定义图书馆中一本图书的简易基本信息：

```
    struct Book                 // 结构名，一般习惯首字母大写，但不必须
    {
        char title[128];        // 书名
        char author[40];        // 作者
        float price;            // 价格
        unsigned int date;      // 出版日期
        char publisher[40];     // 出版社
    } book;                     // 结构体变量和结构体也可以分开写，但要注意相对
于main函数的位置
```

对结构体内变量的初始换可以使用下面的方法：

```
struct Name {
    char a;
    int b;
    char c[10];
} name = {'s',67,"shiad"};      // 对应顺序不可以错
```

但这只是初始化，而不是赋值。对结构体变量进行赋值，只能通过同结构体类型变量或对其基础数据类型的成员进行赋值来达到目的。

结构体成员的访问通过成员访问运算符来完成，成员访问运算有两种，一种是通过

```
结构变量.成员名
```

的方式进行的；另一种是通过

```
结构指针->成员名
```

的方式进行。只是前者面向变量，后者面向指针。 比如下面的程序：

```c
#include <stdio.h>
struct point {
    int x;
    int y;
};
int main()
{
    struct point a, b, *p;
    a.x = 1; a.y = 2;
    printf("%d\t%d\n", a.x, a.y);
    p = &a;
    printf("%d\t%d\n", p->x, p->y);
    b = a;
    printf("%d\t%d\n", b.x, b.y);
    /* 运行结果为
1       2
1       2
1       2
    */
    return 0;
}
```

访问结构体成员需要使用点号(.)运算符，比如book.title就是引用book结构体的title成员，但其实也可以以此来初始化结构体（与数组类似）。

```c
struct Name {
    char a;
    int b;
    char c;
} name = {
    .b = 67,
    .a = 'a',
    .c = "shdai"
    };    //顺序无要求，也可以初始化部分变量
```

单链表

链表是一种抽象的数据结构，它在逻辑上是连续的，但在物理（内存）上一般都不连续。且一般都由若干个节点组成，每个结构体都是一个结构体变量，不同节点之间通过指针链接。链表结构是结构体很重要的应用方向，而单链表是最简单的链表结构。

单链表的每个节点包含两个部分，数据域与指针域，每个节点的指针域指向下一个节点的地址，（最后节点的指针域指向空即可）。只要记住单链表的第一个节点，即可找到所有结点的数据。一般单链表的第一个节点被称为头结点，不存储数据，只做指向下一节点的指针使用。

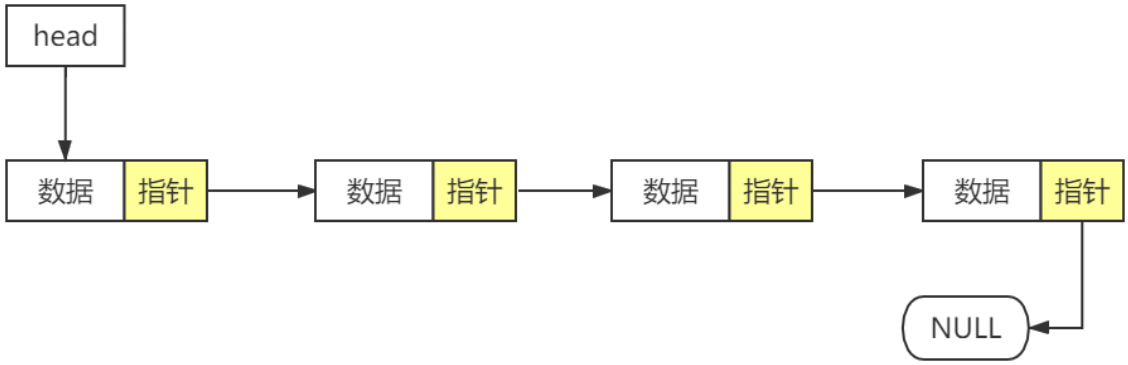

图 10-1 单链表示意图

由于节点数量是可变的，所以链表也是一种动态的数据结构，它的优势在于可以根据元素的多少来动态的添加或删除节点，从而充分利用内存。并且相比于数组，结构体只是描述变量之间的关联，不占多余的内存，结构体变量才占，(但结构体成员之间会进行内存对齐的操作，所以占用内存也确实会比单独多一点)，并且结构体可以定义在 main 函数外作为全局变量。但当结构体组成链表后，对内存的利用率就提高了很多。

具体的链表操作，即如何创建链表，如何添加节点等会在 数据结构与算法 中会详细探讨。

10.2 联合体

联合体（简称联合，也称共用体）是另一种自定义的复合数据类型，它很想结构体，与结构体的区别在于它能使几个不同类型的变量共同占用同一段内存空间（所占空间大小按最长的计算），不同的成员不会同时出现，这段内存地址总会被最后一次修改覆盖掉。声明、定义和结构体类似，只是使用 union 关键字代替 struct ，同时共用体的名称可以省略。成员访问的方法与结构体是一致的。

共用体的一般形式为：

```
union 共用体名称 {            //共用体名称可以省略
      共用体成员1;
      共用体成员2;
      共用体成员3;
      ...
};
```

共用体的初始化与结构体相比也有略微差别：

```
    union Test {int i; double pi; char str[6];};
    union Test a = {500};   //初始化共用体成员，但每次只能初始化一个，因为会被覆盖，第一次默认会初始化第一个成员
    //union Test a = b;              //用一个共用体初始化另一个共用体
    //union Test a = {.ch = 'c'};    //指定初始化成员
```

下面用实例来体验共用体的共用内存：

```c
#include <stdio.h>
#include <string.h>
union Test {
    int i;
    double pi;
    char str[6];
};

```

```c
int main()
{
    union Test test;        //创建共用体变量和结构体类似

    test.i = 520;
    test.pi = 3.14;
    strcpy(test.str, "Hello");

    printf("addr of test.i: %p\n",&test.i);
    printf("addr of test.pi: %p\n",&test.pi);
    printf("addr of test.str: %p\n\n",&test.str);

    printf("test.i: %d\n",test.i);
    printf("test.pi: %.2f\n",test.pi);
    printf("test.i: %s\n",test.str);
/*  共用体的地址是相同的，但如果打印其中的值，只有最后赋值的变量会打印出正确的值，因为变量的值会被覆盖
addr of test.i: 0060FF08
addr of test.pi: 0060FF08
addr of test.str: 0060FF08

test.i: 1819043144
test.pi: 3.13
test.i: Hello
*/
    return 0;
}

```