第5章 构造数据类型——数组

第2章介绍了基本数据类型，如int、float、char等。它们就像一个个独立的“盒子”，每个盒子只能存放一个数据。但是现实世界中存在着更多的批量数据，例如，一个班中50名学生的成绩，一幅图像有1920×1080像素，一条生产线每天的产量等。

为了解决批量同类型数据的存储和处理问题，C语言提供了一种构造类型——数组。数组是程序中最基础、使用最广泛的数据结构之一，可以高效地组织和处理大量同类型的数据，广泛应用于数据分析、科学计算、音频处理、图像处理等多种应用领域中。本章将系统地介绍如何定义、初始化和处理一维数组、二维数组与多维数组、字符数组和字符串，学习批量数据的处理过程，并通过丰富的示例展示数组在解决实际问题中的能力。

5.1 数组概述

相同类型的若干数据元素按有序的形式组织起来形成数据集合，存储该数据集合时按照数据元素排列顺序安排变量内存空间，从而构成相同数据类型变量的有序集合，称为数组。数组的基本操作，如插入、删除、遍历和排序等，是实现计算机科学和软件开发中各种算法的基础。数组广泛应用于图像处理、统计分析等场景的数据存储和处理，是工业生产和人工智能领域中不可或缺的数据结构。

在C语言中，数组是一种构造数据类型。构造类型的数据是由整型、字符型、浮点型等基本数据类型按照一定规则组合而形成的数据类型。

一个数组中包含多个同类型的数据，每一个数据称为数组元素。每个数组元素等同于相应类型的变量，可以独立使用。数组元素使用下标(或索引)来区分，因此数组中的变量也称为下标变量。

数组元素的数据类型可以是C语言中的任何一种数据类型，既可以是基本数据类型，也可以是构造数据类型。按数组元素的类型不同，数组分为数值数组、字符数组、指针数组、结构数组等各种类别。本章介绍数值数组和字符数组。

5.2 一维数组

5.2.1 一维数组定义与存储

一维数组是C语言中最基本的线性数据结构，用于存储一组具有相同数据类型的

数据。

1. 一维数组定义

一维数组定义的一般形式为：

```
数据类型标识符 数组名[常量表达式];
```

其中：① “数据类型标识符”描述数组类型，也是数组中每一个元素的数据类型，可以是任一种基本数据类型或构造数据类型。

② “数组名”是用户定义的数组标识符，遵循 C 语言标识符的命名规则即可。

③ “常量表达式”位于方括号中，表示数组元素的个数，也称为数组的长度，为正整数。数组的长度可以是整型常量、整型常量表达式或符号常量，不能出现非整型表达式。

④ 数组名应是合法标识符，不能与其他标识符同名，避免产生歧义和代码错误。

例如：

```
float prices[5];
```

以上语句定义了一个存储价格的一维数组，其名为 prices，其中，float 表示数组 prices 中每一个元素的数据类型都是单精度实型，[5]表示数组有 5 个元素。

以下数组定义是正确的：

```
#define N 10
int b[N];                    //定义整型数组 b,有 10 个元素
float c[10+N];               //定义实型数组 c,有 20 个元素
char ch[26];                 //定义字符数组 ch,有 26 个元素
```

以下数组定义是错误的：

```
int arr(10);                 //要使用方括号
double b[2.5];               //数组长度必须为整型
char s[];                    //定义时需要设定数组长度
```

2. 一维数组元素的表示和存储

数组元素是组成数组的基本单元。数组元素通过数组名加上一个下标表达式的方法标识，下标表达式指出数组元素在数组中的位置。

一维数组元素的表示形式为：

```
数组名[下标表达式];
```

例如：

```
float score[5];
score[0]=91.5;
int i = 1;
score[i] = score[i-1]-5;
```

又如：

```
int i=0;
float prices[5];
prices[i++] = 91;
```

其中，“[]”是下标运算符；“下标表达式”必须是整型常量或整型表达式，从 0 开始计。

数组元素在内存中的存储形式是顺序结构。数组中所有元素是相同的数据类型，使得数组内存管理变得简单。在内存中，数组元素是按顺序连续存放的，数组第一个元素的内存地址称为数组首地址，后续元素紧随其后依次存放在内存的地址连续区域中。数组的大小通常在定义时就被确定了，不能动态地改变，在创建数组时，会根据数据类型和数组元素数量来计算所需的总字节数。在访问数组元素时，根据数组首地址和元素大小来计算具体元素的内存地址。

例如：

```
float x, prices[5];                    //数组定义，[]是数组长度的表示
x = prices[2];                         //使用数组元素，[]是下标运算符
```

数组 prices 在内存中的存储形式如图 5-1 所示。

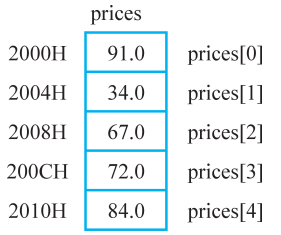

图 5-1　prices 数组在内存中的存储形式的元素地址及下标表示示意

数组中的每个元素是通过[ ]进行地址运算得到的。[]是下标运算符，为最高优先级，左结合，用于访问数组中的元素。下标运算的过程如下。

① 先计算元素的地址，计算方法为：

数组元素内存地址＝数组首元素地址＋元素下标值×数组类型所占内存空间大小

② 根据地址，从内存中获取相应类型的数据。

例如，假设 prices 为 2000H，则 prices[2]地址为 2000H＋2×4＝2008H，取出 2008H 开始的 4 个内存单元(float 型)中存放的内容(67.0)赋给 x。

数组元素使用时应注意：

① 数组名代表数组的首地址，是常量。在使用数组时，不能对数组整体操作，只能对数组元素单独使用。例如：

```
float prices[5];
prices = 91.0;                         //错误，prices 是常量，不能整体操作
prices[0] = 91.0;                      //正确，对数组元素进行操作
```

② 数组元素的下标从 0 开始，数组的最大下标是数组长度减 1。

编程时要特别注意，C 编译系统不做越界检查，如果引用的数组元素超出数组范围，就会破坏内存中其他变量的值，导致其他数据值被破坏、程序崩溃等难以预测的后果。

例如：

```
float prices[5];
prices[5]=50.0;                        //下标越界
```

应注意，定义数组时的方括号中给出的是数组的长度；而引用数组元素时的下标是该元素在数组中位置的标识，取值范围为 0～长度－1。前者只能是常量，后者可以是常量、变量或表达式。

【思政小课堂】

一维数组是一组类型相同的数据在内存中连续存放的结构，连续存储和地址计算可实现数据高效随机访问。

由于数据需要连续空间且长度固定易造成越界访问，存在内存管理安全问题。规范的内存管理像规则和制度，保障系统稳定；而越界访问则类似公民责任，违反规则会产生不可预见的风险。培养良好的编程习惯（边界检查、合理分配、及时释放）既是对技术的负责，也是对他人利益和公共资源的尊重，体现作为程序员的职业道德与社会责任。

3. 一维数组元素的基本处理方法

对数组中的所有元素进行统一处理（如输入、修改、计算、输出）时，最常见、最高效的方式就是使用循环，循环变量为数组元素的下标。例如：

```
float prices[5];
for(int i=0;i<5;i++)                          //循环变量 i 为元素下标
    scanf("%f", &prices[i]);                  //读入 prices[i]的值
```

【例 5-1】 计算斐波那契数列的前 30 项并存储，并实现 5 个数据一行的输出格式。

编程思路：

定义数组 fibonacci[30]用来存储斐波那契数列的前 30 项。

根据斐波那契数列的数学定义，可以通过 fibonacci[i]＝fibonacci[i－1]＋fibonacci[i－2]，建立数列各项之间递推关系，数组下标变化描述了前两项计算后续项过程。在计算过程中使用一层 for 循环逐步更新和存储斐波那契数列各项值。每次输出元素的值后，通过 if 判断当前数据个数 i＋1 是否为 5 的倍数，以确定是否换行，实现 5 个数据一行的输出格式。

代码实现：

```
#include <stdio.h>
#define N 30
int main() {
    int i;
    int fibonacci[N];                         //定义数组
    fibonacci[0] = 0;                         //斐波那契数列的前 2 项已知
    fibonacci[1] = 1;
    for (i = 2; i < N; i++)                   //fibonacci[2]~fibonacci[N-1]
        fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];     //递推计算
    for (i = 0; i < N; i++) {                 //输出所有项
        printf("%8d", fibonacci[i]);
        if ((i + 1) % 5 == 0)                 //判断是否已输出一行 5 个数据
            printf("\n");
    }
    return 0;
}
```

运行情况如下：

```
0       1       1       2       3
5       8       13      21      34
```

```
55          89          144         233         377
610         987         1597        2584        4181
6765        10946       17711       28657       46368
75025       121393      196418      317811      514229
```

思考与拓展：将上述程序中的 N 值修改为 50，运行程序时会出现输出负数的现象。思考其原因，并修改程序，确保在计算过程中保持数值的正确性和稳定性。

5.2.2 一维数组的初始化

数组初始化赋值是指在数组定义时给数组元素赋予初值。初始化数组有助于避免使用未初始化的变量带来不可预测行为，是编写健壮程序的重要一步。

一维数组初始化的一般形式为：

```
数据类型标识符 数组名[常量表达式]={值,值,……,值};
```

其中，{ }中的各数据值即为各数组元素的初值，各值之间用逗号间隔，并与数组元素的下标顺序一致。例如：

```
int a[10]={0,1,2,3,4,5,6,7,8,9};
```

相当于

```
a[0]=0;a[1]=1;…;a[9]=9;
```

这样多条语句逐一给元素赋值。

一维数组初始化有如下形式。

(1) 对全部数组元素赋初值。例如：

```
int a[5]={0,1,2,3,4};
```

a[0]的值为 0，a[1]的值为 1，……，a[4]的值为 4。

对全部数组元素赋初值时，可以不指定数组长度。例如：

```
int a[]={0,1,2,3,4};
```

系统将根据初值的个数确定 a 数组的长度为 5。

(2) 对部分数组元素赋初值。

当{ }中值的个数少于元素个数时，只给前面部分元素赋值，系统将后面没赋初值的元素自动赋值为 0。例如：

```
int a[5]={0,1,2};
```

a[0]的值为 0，a[1]的值为 1，a[2]的值为 2，其余元素的值为 0。

(3) 指定元素初始化。

在 C99 及后续标准中，可以使用指定的初始化器创建相应的初始值。例如：

```
int a[5] = {[0] = 1, [2] = 2};
```

a[0]的值为 1，a[2]的值为 2，其余元素的值为 0。

数组初始化时应注意：

(1) 数组未初始化时，数组元素的值是未定义的垃圾值，直接使用这些值进行运算，会导致程序出现不可预测的结果，这是常见的 C 语言编程陷阱。

(2) 当初值的个数多于数组元素的个数时，编译系统会报错。

【例 5-2】 已知近 5 天的钢材每吨价格分别为 5000、5200、5100、5200、5300 元，编写程序，按日期逆序打印出来。

编程思路：定义数组 prices[5]存储钢材价格，并进行初始化。一维数组元素有一个下标，用一层计数循环就可以控制一维数组元素下标变化，下标从最后一个元素到第一个元素递减，即可实现逆序打印。

代码实现：

```
#include <stdio.h>
#define SIZE 5                                          //数组长度
int main()
{   //定义数组并初始化
    float prices[SIZE] = {5000.0, 5200.0, 5100.0, 5200.0, 5300.0};
    printf("\n 钢材价格数据(单位:元/吨):\n");
    printf("以下是过去 5 天的钢材价格记录(逆序输出),反映了市场的波动情况:\n");

    for (int i = SIZE - 1; i >= 0; i--)          //逆序打印每一天的钢材价格
    {
        printf("第 %d 天的价格: %.2f 元/吨\n", i + 1, prices[i]);
    }
    return 0;
}
```

运行情况如下：

```
钢材价格数据(单位:元/吨):
以下是过去 5 天的钢材价格记录(逆序输出),反映了市场的波动情况:
第 5 天的价格: 5300.00 元/吨
第 4 天的价格: 5200.00 元/吨
第 3 天的价格: 5100.00 元/吨
第 2 天的价格: 5200.00 元/吨
第 1 天的价格: 5000.00 元/吨
```

本例中使用数组存放和操作序列化的钢材价格数据；从数组的最后一个元素开始逆向遍历输出，是一种反向访问的逆序算法思维。

5.2.3 可变长数组

在 C99 标准之前，传统数组的大小在编译时必须确定，定义数组时，数组大小是固定的常量。C99 标准中允许在使用可变长度数组(Variable Length Arrays，VLA)，即数组大小可以在运行时动态确定，适合处理动态数据。例如：

```
int m;
scanf("%d", &m);
int arr[m];                                          //定义可变长数组
```

使用可变长数组时应注意，C99 及以上 C 标准中才能支持可变长数组，并非所有的现代编译器都支持可变长数组，如 Visual Studio 的 C 编译器 MSVC 只部分支持 C99 标准，不支

持可变长数组。

5.2.4 处理一维数组元素

一维数组中存放着同类型的一批数据，查找、统计、排序、最值处理是常见的数据处理任务，也是程序中数组的基本操作。

1. 数组元素总和、平均值计算和最值元素查找

通过遍历数组每一个元素实现累加求和及求均值，通过擂台法进行求最值。

【例 5-3】 编写一个 C 语言应用程序，实现对 9 天内钢材价格进行计算总和、求平均值，以及找出价格的最小值和最大值。

编程思路：

将价钢材价格以初始化赋值方式存储在 prices 数组中；通过循环遍历所有元素累加计算 9 天钢材价格的总和，总和除以 9 即可得到价格平均值；采用打擂台法，计算出价格的最小值和最大值。

打擂台法是常用的最值求解算法。首先指定某元素值为最大值或最小值的擂主，通常指定为数组第一个元素，本例中为 max＝min＝prices［0］。随后遍历数组，其他元素依次与擂主进行比较，若某元素大于最大值擂主，则该元素为最大值擂主 max＝a[i]，否则再判断该元素是否小于最小值擂主，如果小于则该元素为最小值擂主 min＝a[i]；数组遍历结束，max 和 min 中就是最大值和最小值。

程序流程如图 5-2 所示。

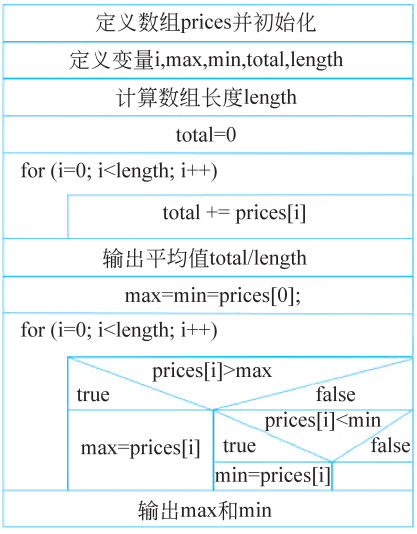

图 5-2 数组求和与求解最值流程

代码实现：

```
#include <stdio.h>
int main()
{
    //初始化钢材价格(单位:元/吨)
    float prices[] = {5000.0, 5200.0, 5100.0, 5300.0, 5400.0, 4950.0, 6700.0,
                      8900.0, 9000.0}, max, min;
    int length = sizeof(prices) / sizeof(prices[0]), i;     //计算数组长度
    //计算价格的总和、平均值
    float total = 0.0;
    for (i = 0; i < length; i++)
        total += prices[i];
    //输出结果
    printf("总价格: %.2f 元/吨\n", total);
    printf("平均价格: %.2f 元/吨\n", total / length);
    //擂台法,计算价格的最小值和最大值
    max = min = prices[0];                          //指定当前擂主为 prices[0]
    for (i = 1; i < length; i++) {                  //与后续的每个元素一一比较
        if (prices[i] > max)                        //后续元素更大时
```

```
            max = prices[i];                    //更新最大值
        else if (prices[i] < min)               //后续元素更小时
            min = prices[i];                    //更新最小值
    }
    //输出结果
    printf("最低价格: %.2f 元/吨\n", min);
    printf("最高价格: %.2f 元/吨\n", max);
    return 0;
}
```

运行情况如下：

```
总价格: 55550.00 元/吨
平均价格: 6172.22 元/吨
最低价格: 4950.00 元/吨
最高价格: 9000.00 元/吨
```

思考与拓展：打擂台法实际上是对比优化的典型应用。在寻找最大值或最小值的过程中，通过逐步比较来找到最值。这种遍历所有数据、逐步比较寻优的过程体现了朴素的贪心算法思想。通常寻优时，除了记录最值，还需要记录最值的位置。请尝试修改例 5-3 中的程序，在查找最值的同时，记录出现最值的时间点，即对数组每个数据的下标进行处理。

2. 顺序查找数组指定元素

一维数组中每个元素的下标是元素前后顺序来排列的，可以直接使用下标顺序访问数组，查找数组中的指定元素。

【例 5-4】 已知近 10 日的钢材价格，根据用户输入的日期，查找特定日期的钢材价格。用户输入 1 代表第 1 天，输入 2 代表第 2 天，以此类推。如果输入有效，即在 1 到 9 之间，程序将输出对应日期的价格，否则无效，会提示未找到。

编程思路：

使用例 5-3 中的 prices 数组，使用 sizeof 计算数组长度。数组下标是从 0 开始的，将用户输入的第几天转换为数组下标，即 int array_index=index_to_find−1，设定日期有效性检测标志变量并检查转换后的下标是否在有效范围内，即 array_index >= 0 && array_index < length。如果有效，使用下标直接定位，从 prices 数组中获取对应的价格，否则给出提示。

代码实现：

```
#include <stdio.h>
int main()
{
    float prices[] = {5000.0, 5200.0, 5100.0, 5300.0, 5400.0, 4950.0, 6700.0,
                      8900.0, 9000.0};
    int length = sizeof(prices) / sizeof(prices[0]);
    //查找特定时间点的钢材价格
    int index_to_find;                          //存放用户输入的日期
    printf("输入要查找的日期(从 1 开始):");
    scanf("%d", &index_to_find);
    int array_index = index_to_find - 1;     //将用户输入的日期转换为数组下标(减 1)
    if (array_index >= 0 && array_index < length) {     //有效日期
        float price = prices[array_index];  //找到对应价格，存入 price 中
```

```
        printf("第 %d 天钢材价格: %.2f 元/吨\n", index_to_find, price);
    }
    else                                        //无效日期,提示
        printf("输入无效,超出日期范围。\n");
    return 0;
}
```

运行情况如下：

```
输入要查找的日期(从 1 开始):5↙
第 5 天钢材价格: 5400.00 元/吨
```

再次运行：

```
输入要查找的日期(从 1 开始):15↙
输入无效,超出日期范围。
```

思考与拓展：本例根据用户输入的日期，直接在数组当中定位到对应的价格。如果用户输入价格，要查找该价格对应的日期，就需要对数组进行遍历了。对数组进行遍历可以从前到后正向遍历，也可以从后往前逆向遍历，只需要修改下标的变化方向即可。

【例 5-4(改进版)】 已知近 10 日的钢材价格，根据用户输入的价格，查找该钢材价格对应的日期。

编程思路：

使用例 5-3 中的 prices 数组，使用 sizeof 计算数组长度。

将用户输入的价格和数组元素的值逐一比较：相等则输出对应的天数(下标加 1)；不相等则继续比较下一个元素。

为了记录是否找到该价格，可以定义标志变量 isfound，相等时把它置为 1。

代码实现：

```
#include <stdio.h>
#include <math.h>
int main()
{
    float prices[] = {5000.0, 5200.0, 5100.0, 5300.0, 5400.0, 4950.0, 6700.0,
                      8900.0, 9000.0}, price;
    int length = sizeof(prices) / sizeof(prices[0]);
    int isfound = 0;                            //标志变量,是否找到,初值为 0
    printf("输入要查找的价格:");
    scanf("%f", &price);
    for(int i=0; i<length; i++)                 //从前到后,正向遍历
        if (fabs(prices[i]-price) < 1e-6)       //浮点数相等
        {
            printf("第 %d 天钢材价格: %.2f 元/吨\n", i+1, price);     //输出
            isfound = 1;                        //找到,标志变量置 1
        }
    if(isfound==0)                              //无找到
        printf("未找到对应价格。\n");
    return 0;
}
```

运行情况如下：

```
输入要查找的价格:5300↙
第 4 天钢材价格: 5300.00 元/吨
```

再次运行：

```
输入要查找的价格:5500↙
未找到对应价格。
```

思考：如何修改程序实现从后向前、逆序查找，以及找到第一个结果时即结束查找？

3. 数组元素排序

对数组元素进行排序是数据处理中的常见操作。选择排序和冒泡排序是两种思想简单易理解的排序算法。

【例 5-5】 编写程序，实现对 9 天内钢材价格按升序排列，打印输出对应的时间点和价格。

编程思路：

为实现题目要求，需要设计一个排序算法。选择排序是一种经典的排序算法，基本思想是重复选择未排序部分中的最小元素，并将其放到已排序部分的末尾，直至所有元素有序。

以 6 个数：9、2、5、1、6、4 为例，介绍选择排序过程。

初始数据：9，2，5，1，6，4

第一趟：选择基准为第一个元素 9，依次与后面的元素进行比较。

初始数组：[9]，2，5，1，6，4

第一次比较 9 与 2，9 ＞ 2，交换两数。	数组状态：2，9，5，1，6，4
第二次比较 2 与 5，2 ＜ 5，不交换。	数组状态：2，9，5，1，6，4
第三次比较 2 与 1，2 ＞ 1，交换两数。	数组状态：1，9，5，2，6，4
第四次比较 1 与 6，1 ＜ 6，不交换。	数组状态：1，9，5，2，6，4
第五次比较 1 与 4，1 ＜ 4，不交换。	数组状态：1，9，5，2，6，4

经过第一趟比较，得到最小值 1，数组状态为：[1]，9，5，2，6，4

第二趟：选择基准为第二个元素 9，依次与后面的元素进行比较。

初始数组：1，[9]，5，2，6，4

第一次比较 9 与 5，9 ＞ 5，交换两数。	数组状态：1，5，9，2，6，4
第二次比较 5 与 2，5 ＞ 2，交换两数。	数组状态：1，2，9，5，6，4
第三次比较 2 与 6，2 ＜ 6，不交换。	数组状态：1，2，9，5，6，4
第四次比较 2 与 4，2 ＜ 4，不交换。	数组状态：1，2，9，5，6，4

经过第二趟比较，得到第二小的值 2，数组状态为：1，[2]，9，5，6，4

第三趟：选择基准为第三个元素 9，依次与后面的元素进行比较。

初始数组：1，2，[9]，5，6，4

第一次比较 9 与 5，9 ＞ 5，交换两数。	数组状态：1，2，5，9，6，4
第二次比较 5 与 6，5 ＜ 6，不交换。	数组状态：1，2，5，9，6，4
第三次比较 5 与 4，5 ＞ 4，交换两数。	数组状态：1，2，4，9，6，5

经过第三趟比较，得到第三小的值 4，排序状态为：1，2，[4]，9，6，5

第四趟：选择基准为第四个元素 9，依次与后面的元素进行比较。