1806ICT
Programming Fundamentals

C Arrays

Topics

- Arrays
  - Declaration
  - Initialization
  - Input/Output
  - Two-dimensional arrays
  - Dynamic Memory Allocation

Motivation for Arrays

- Suppose you want to compute
  - the average temperature for the seven days in a week
  - number of days with temperature below average
  - number of days with temperature above average

```
int main()  {
     double temp1, temp2,  temp3, temp4, temp5, temp6, temp7;
     double avgTemp;
     int countBelow = 0, countAbove = 0;

     scanf(“%lf %lf %lf %lf %lf %lf %lf”, &temp1, &temp2, &temp3, &temp4, &temp5, &temp6, &temp7);

     avgTemp = (temp1+temp2+temp3+temp4+temp5+temp6+temp7)/7;

     if (temp1 < avgTemp)
           countBelow++;
     else if (temp1 > avgTemp)                  This code is repeated for temp2, temp3, … temp7
           countAbove++;

     …
     …
     printf(“Average temperature = %f\n”, avgTemp);
     printf(“%d days with below average temperatures\n”, countBelow);
     printf(“%d days with above average temperatures\n”, countAbove);

     return 0;
}
```

Motivation for Arrays

- Clearly this method of storing and processing data is inefficient
  - leads to a lot of coding duplication (7 if-else statements)
  - very difficult to extend to, for example, temperature data for 365 days in a year

- Arrays help us organize large amounts of information

- An array is a group of contiguous memory locations used to store a series of related values
  - All values are of the same type

Topics

✓ Arrays

- Declaration
- Initialization
- Input/Output
- Two-dimensional arrays
- Dynamic Memory Allocation

Array Declaration

- The array variable must be created by defining the data type and number of elements

- For example, an array named `scores` could be declared as:

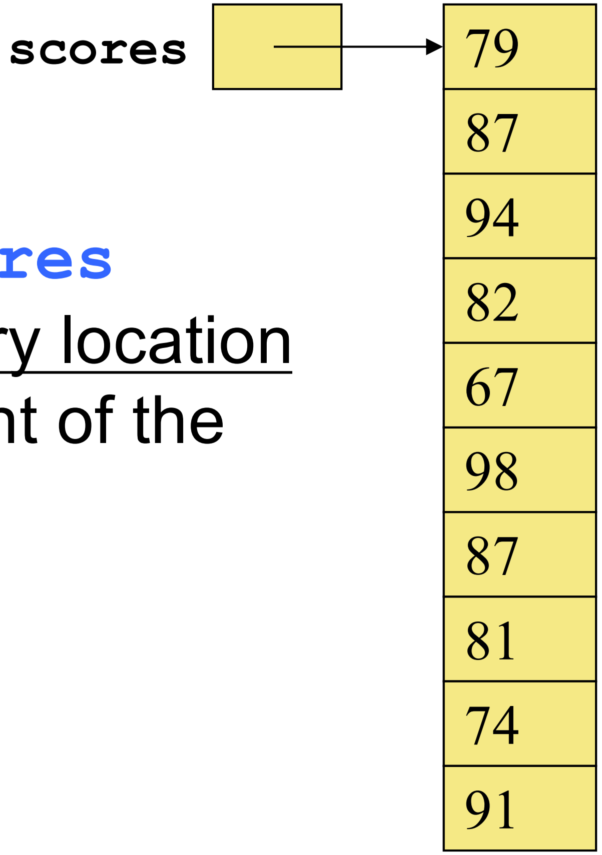
```
int scores[10];
```

  - The variable `scores` is an array of integer values
  - The value 10 in the declaration represents the number of elements in the array

Arrays

```
int scores[10];
```

A way to depict the **scores** array



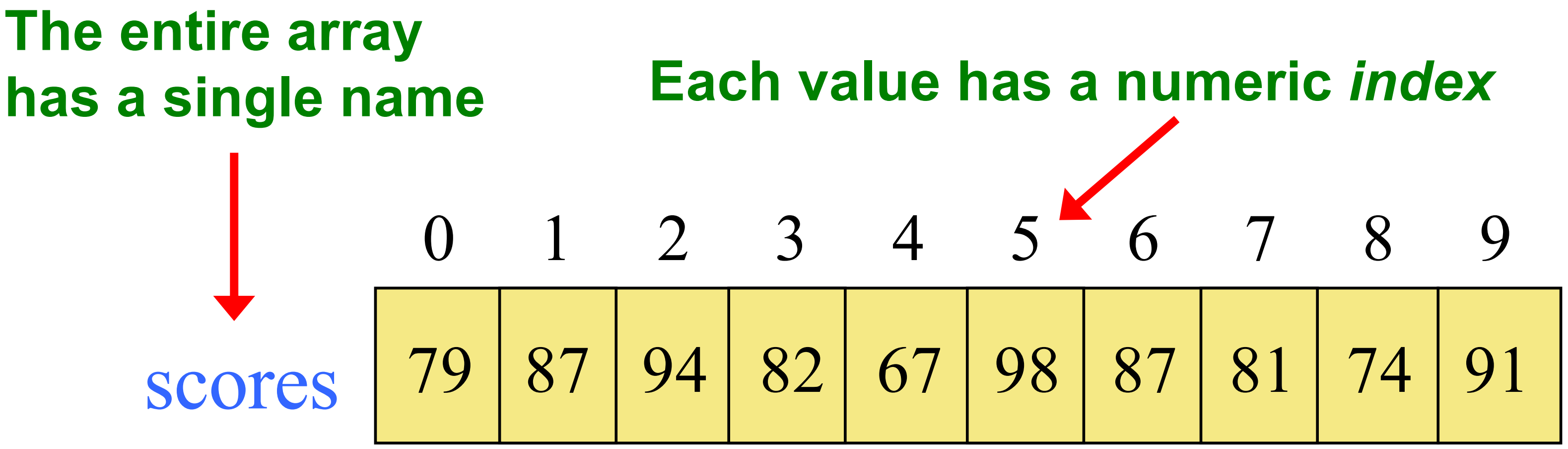
scores

79
87
94
82
67
98
87
81
74
91

The variable **scores** stores the memory location of the first element of the array

Arrays

An *array* is a list of values



The entire array has a single name

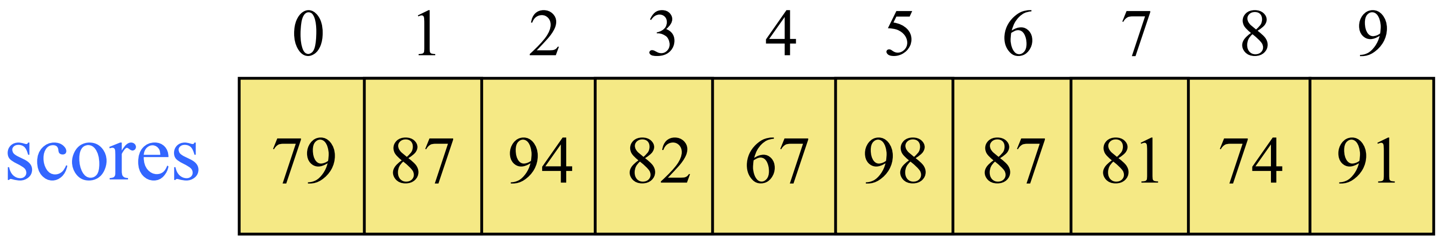
Each value has a numeric *index*

	0	1	2	3	4	5	6	7	8	9
scores	79	87	94	82	67	98	87	81	74	91

An array of size N is indexed from zero to N-1

This array holds 10 values that are indexed from 0 to 9

Arrays



	0	1	2	3	4	5	6	7	8	9
scores	79	87	94	82	67	98	87	81	74	91

- A particular value in an array is referenced using the array name followed by the index in square brackets
- For example, the expression

```
scores[2]
```

refers to the value `94` (the 3rd value in the array)

Topics

- ✓ Arrays
  - ❖ Declaration
  - – Initialization
  - – Input/Output
  - – Two-dimensional arrays
  - – Dynamic Memory Allocation

Initialization

- An *initializer list* can be used to instantiate and fill an array in one step
- Used when we know what values to put into the array
- The values are delimited by braces and separated by commas
- Examples

```
int units[7] = {147, 323, 89, 933, 540, 269, 97};

char letterGrades[5] = {'A', 'B', 'C', 'D', ’F'};
```

Initialization

- When the list of initializer is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero
  - `int age[100] = {0};`
  - Initializes all the elements of `age` to zero
- If an array is declared without a size and is initialized to a series of values, it is implicitly given the size of the number of initializers
  - `int age[] = {2, 3, 5, 7};`
  - `int age[4] = {2, 3, 5, 7};`

Equivalent declarations

Topics

- ✓ Arrays
  - ❖ Declaration
  - ❖ Initialization
  - – Input/Output
  - – Two-dimensional arrays
  - – Dynamic Memory Allocation

Input / Output of Arrays

- Library functions printf() and scanf() do not know about arrays

⇒ So we have to do I/O ourselves

Example: IORainfall

```c
#include <stdio.h>
#define  NMONTHS 12

/* Store and print rainfall */

int main()
{
  int data[NMONTHS];
  int month;

  for ( month=0; month < NMONTHS; month++ )
  {
     scanf("%d", &data[month] );
  }

  ...
```

What about our original problem?

- Suppose you want to compute
  - the average temperature for the seven days in a week
  - number of days with temperature below average
  - number of days with temperature above average

```
int main()  {
     double temp1, temp2,  temp3, temp4, temp5, temp6, temp7;
     double avgTemp;
     int countBelow = 0, countAbove = 0;

     scanf(“%lf %lf %lf %lf %lf %lf %lf”, &temp1, &temp2, &temp3, &temp4, &temp5, &temp6, &temp7);

     avgTemp = (temp1+temp2+temp3+temp4+temp5+temp6+temp7)/7;

     if (temp1 < avgTemp)
           countBelow++;
     else if (temp1 > avgTemp)
           countAbove++;
```

This code is repeated for temp2, temp3, … temp7

```
     …
     …
     printf(“Average temperature = %f\n”, avgTemp);
     printf(“%d days with below average temperatures\n”, countBelow);
     printf(“%d days with above average temperatures\n”, countAbove);

     return 0;
}
```

Solving our original problem with arrays

```
#define NDAYS 7

int main()  {
   double temp[NDAYS];
   double sum = 0.0, avgTemp;
   int countBelow = 0, countAbove = 0;

   for (int i=0; i<NDAYS; i++)
   {
      scanf(“%lf”, &temp[i]);
      sum += temp[i];
   }

   avgTemp = sum/NDAYS;

   for (int i=0; i<NDAYS; i++)
   {
      if (temp[i] < avgTemp)
        countBelow++;
      else if (temp[i] > avgTemp)
        countAbove++;
   }

   printf(“Average temperature = %f\n”, avgTemp);
   printf(“%d days with below average temperatures\n”, countBelow);
   printf(“%d days with above average temperatures\n”, countAbove);

   return 0;
}
```

Reading into the array elements

Handling Indices

- Arrays have a fixed size
- An index used in an array reference must specify a valid element
  - The index value must be in the range 0 to N-1
- There is no built-in way of checking if the supplied index is within range
- We must check for valid indices ourselves

Example: DailyTemp

```
#include <stdio.h>
#define  NDAYS 7

/* checking for valid indices */
int main()
{
  double data[NDAYS] = {30.0, 32.1, 2    , 28.4, 30.4,
32.1, 33.4};
  int dayInput;
  scanf("%d", &dayInput );
  if (dayInput >= 0 && dayInput < NDAYS)
      printf("Temp on day %d = %f\n", data[dayInput]);
  else
      printf("Day must be between 0 and %d\n", NDAYS-1);

  return 0;
}
```

Checking for valid indices

Topics

✓ Arrays
    - ❖ Declaration
    - ❖ Initialization
    - ❖ Input/Output
    - Two-dimensional arrays
    - Dynamic Memory Allocation

Two-Dimensional Arrays

- Each element of an array is like a single item of a particular type
- But an array itself is an item of a particular type
  ➡ So, an array element could be another array
- An “array-of-arrays” is called “multi-dimensional” because you need to specify several ordinates to locate an actual element

Example: YearlyRainfall

columns month

rows

year

	0	1	2	3	4	5	6	7	8	9	10	11
0	30	40	75	95	130	220	210	185	135	80	40	45
1	25	25	80	75	115	270	200	165	85	5	10	0
2	35	45	90	80	100	205	135	140	170	75	60	95
3	30	40	70	70	90	180	180	210	145	35	85	80
4	30	35	30	90	150	230	305	295	60	95	80	30

Average Yearly Rainfall (in mm)

Problem: using the *Yearly Rainfall* table

- input month and year
- output average rainfall for that month and year

Example (cont): YearlyRainfall-1

```
#define  NYEARS   5
#define  NMONTHS  12

int lookup(int year, int month)
{
   int table[NYEARS][NMONTHS] =
      {
         {30,40,75,95,130,220,210,185,135,80,40,45},
         {25,25,80,75,115,270,200,165, 85, 5,10, 0},
         {35,45,90,80,100,205,135,140,170,75,60,95},
         {30,40,70,70, 90,180,180,210,145,35,85,80},
         {30,35,30,90,150,230,305,295, 60,95,80,30}
      };

   if ((0 <= year) && (year < NYEARS) &&
       (0 <= month) && (month < NMONTHS))
   {
      return table[year][month];
   }
   else
   {
      return -1;
   }
}
```

Example (cont): YearlyRainfall-2

```
int main()
{
   int   year;
   int   month;
   int   rainfall;

   printf("Enter year and month: ");
   scanf("%d %d", &year, &month);

   rainfall = lookup(year - 1, month - 1);

   if (rainfall < 0)
   {
      printf("Year must be between 1 and %d,\n", NYEARS);
      printf("and month must be between 1 and %d.\n", NMONTHS);
   }
   else
   {
      printf("Rainfall for year %d, month %d is %d mm.\n",
         year, month, rainfall);
   }
   return 0;
}
```

Topics

✓ Arrays

  ❖ Declaration

  ❖ Initialization

  ❖ Input/Output

  ❖ Passing arrays to functions

  ❖ Two-dimensional arrays

  ❖ Dynamic Memory Allocation

Dynamic Memory Allocation

- Often a program doesn’t know in advance how much memory it needs

- Dynamic memory allocation allows memory to be requested at runtime as required
  - Lifetime of each memory allocation is controlled by the programmer

Dynamic Memory Allocation

- Standard library functions for dynamic memory allocation and de-allocation are
  - `malloc()`, `calloc()` and `free()`
  - Need to include `<stdlib.h>`
  - Used to dynamically create memory space for arrays, structures, and unions
- Example

```
// allocate memory for array of 10 integers
int *ptr = (int *)malloc(10 * sizeof(int));
...          // do something with ptr
free(ptr);   // release memory allocated by malloc
```

Dynamic Memory Allocation

```
// allocate memory for array of 10 integers
int *ptr = (int *)malloc(10 * sizeof(int));
...            // do something with ptr
free(ptr);     // release memory allocated by malloc
```

- `void *malloc(size)`
  - `size` is the number of bytes of memory to allocate
  - return value is a pointer to the requested memory
  - the type `void *` specifies a generic pointer, and can represent a pointer of any type
    - Use explicit cast to the desired type

Dynamic Memory Allocation

- **`void *malloc(size)`**
  - Important to check the return value in case **`malloc()`** failed to allocate the memory
    - When this happens, **`malloc()`** returns a NULL pointer

```
// allocate memory for array of 10 integers
int *ptr = (int *)malloc(10 * sizeof(int));

if (ptr == NULL)
{
   printf("malloc failed\n");
   return 1;
}

...           // do something with ptr
free(ptr);    // release memory allocated by malloc
```

Dynamic Memory Allocation

- `void *calloc(n, size)`
  - behaves similar to `malloc()`, but `malloc()` returns uninitialized memory
  - `calloc()` returns memory initialized with zeros
  - `n` specifies number of elements in the requested array, `size` is the size of each element

```
// allocate memory for array of 10 integers, all zeros
int *ptr = (int *)calloc(10, sizeof(int));

if (ptr == NULL)
{
   printf("calloc failed\n");
   return 1;
}

...            // do something with ptr
free(ptr);     // release memory allocated by calloc
```

Dynamic Memory Allocation Example

```
/* Repeatedly allocate memory for an array, fill it with
 * random numbers, and print the array */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void fillArray(int *, int);
void printArray(int *, int);

int main()
{
   int *ptrArray;
   int n;
   srand(time(NULL));      // seed the random number generator

   while (1) {
       scanf("%d", &n);    // read in size of array
       ptrArray = calloc(n, sizeof(int)); // allocate memory
       if (ptrArray == NULL)
             break;
       fillArray(ptrArray, n);
       printArray(ptrArray, n);
       free(ptrArray);     // free the memory
   }
   return 0;
}
```

Dynamic Memory Allocation Example

```
/* fill array with random numbers between 0 to 9 */
void fillArray(int *numsPtr, int size)
{
   for (int i=0; i<size; i++)
      numsPtr[i] = rand()%10;
}

/* print array */
void printArray(int *numsPtr, int size)
{
   printf("array = [ ");

   for (int i=0; i<size; i++)
      printf("%d ", numsPtr[i]);

   printf("]\n");

}
```

Summary

- Arrays
  - Declaration
  - Initialization
  - Input/Output
  - Two-dimensional arrays
  - Dynamic Memory Allocation