Arrays and Pointers

Fundamentals of Programming

Lecture 3

Contents

- Arrays
- Pointers
- Array pointers

Arrays

Array

- An array is data associated with an identifier and similar data type

```
<data_type> <identifier>[SIZE];
```

- Data is individually stored as an element in the array
- The last element in a char array is NULL (\0)
- Arrays are either 1-dimensional or 2-dimensional

Array declaration examples

```
int a[5]; // 5 elements of int
float age[12]; // 12 elements of float
char message[20]; // 20 elements of char
```

Initialised arrays

- Array elements can be initialised with zeros

```
int a[5] = {0};
float b[12] = {0.0};
```

- Array elements can be initialised with pre-set values

```
int levels[5] = {1, 2, 3, 4, 5};
float scores[3] = {2.56, 17.42, 33.91};
double dist[3] = {32.5e6, 17.81e8, 34.4e7};
```

Elements in an array

```
int a[3] = {2, 7, 14};
```

Index	**0**	**1**	**2**
Element	a[0]	a[1]	a[2]
Data	2	7	14

Contents of arrays (1)

- If an array is **not** initialized, the data in the array elements will be dependent on what was left in the computer memory

```
#include <stdio.h>
int main()
{
   int a[3];
   int n;
   for (n=0; n<3; n++) {
      printf("a[%d] = %d\n", n, a[n]);
   }
   return 0;
}
```

Contents of arrays (2)

- Array is initialised with zero
- The data in the array elements will be cleared to zero
- Useful when the elements are needed to store running totals

```
#include <stdio.h>
int main()
{
  int a[3] = {0};
  int n;
  for (n=0; n<3; n++) {
    printf("a[%d] = %d\n", n, a[n]);
  }
  return 0;
}
```

Contents of arrays (3)

- Array is initialised with preset values
- The data in the array elements are unique
- Useful when the preset data is needed in the calculations

```
#include <stdio.h>
int main()
{
  int a[3] = {1, 2, 3};
  int n;
  for (n=0; n<3; n++) {
    printf("a[%d] = %d\n", n, a[n]);
  }
  return 0;
}
```

Array example 1

- The array x[] has 3 elements
- x[0] is the first element in the array
- x[1] is the second element
- x[2] is the third and last element
- Each element is assigned an integer value just like a regular variable

```
#include <stdio.h>
int main()
{
   int x[3];
   x[0] = 1;
   x[1] = 2;
   x[2] = 3;
   return 0;
}
```

Array example 2

- Array x[] is declared and initialised at the same time
- Variable n is the counter variable
- FOR loop is used to output all the data in array x[]

```
#include <stdio.h>
int main()
{
  int x[3] = {1, 2, 3};
  int n;
  for (n = 0; n < 3; n++) {
    printf("x[%d] = %d\n", n, x[n]);
  }
  return 0;
}
```

Array example 3

- Array x[] is cleared when declared
- Variable n is the counter variable
- First FOR loop is used to collect data from the user
- Second FOR loop is used to print the contents of the array x[]

```
#include <stdio.h>
int main()
{
  int x[3] = {0};
  int n;
  // input data
  for (n = 0; n < 3; n++) {
    printf("Enter x[%d] value: ", n);
    scanf("%d", &x[n]);
  }
  //output data
  for (n = 0; n < 3; n++) {
    printf("x[%d] = %d\n", n, x[n]);
  }
  return 0;
}
```

2-dimensional array

- In 2-dimensional arrays, the row and column dimensions have to be declared

```
<data_type> <identifier>[ROW][COL]
```

- The 2-dimensional arrays are suitable for complex data, such as coordinates, data tables, etc.

Elements in 2-D array

- The elements are arranged as a matrix

$$x = \begin{bmatrix} x_{00} & \cdots & x_{0j} \\ \vdots & \ddots & \vdots \\ x_{i0} & \cdots & x_{ij} \end{bmatrix}$$

where i is row, j is column

- For example, `int x[2][3]` means 2 rows and 3 columns of integers

2-D array example 1

- Array data $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
- Output:

```
Row0, Col0 = 1
Row0, Col1 = 2
Row1, Col0 = 3
Row1, Col1 = 4
```

```
#include <stdio.h>
int main()
{
  int a[2][2] = { {1,2}, {3,4} };
  printf("Row0, Col0 = %d\n", a[0][0]);
  printf("Row0, Col1 = %d\n", a[0][1]);
  printf("Row1, Col0 = %d\n", a[1][0]);
  printf("Row1, Col1 = %d\n", a[1][1]);
  return 0;
}
```

2-D array example 2

- Program prompts user for six integer values
- First nested FOR loop reads six values, stores the data in first row, then second row
- Second nested FOR loop is used to output all the contents in array x[][]

```
#include <stdio.h>
int main()
{
  int x[2][3];
  int i, j;
  // input data
  printf("Enter six values: ");
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      scanf("%d", &x[i][j]);
    }
  }
  // output data
  for (i = 0; i < 2; i++) {
    for (j = 0; j < 3; j++) {
      printf("x[%d][%d] = %d\n", i, j, x[i][j]);
    }
  }
  return 0;
}
```

Processing arrays

- Given the table below represents a student's score in a game, determine the average score
- The average result is a floating point value

Element	Data
score[0]	86
score[1]	72
score[2]	67
score[3]	94
score[4]	55

Solution 1

- Elements are added manually
- Total score is stored in variable average
- Variable average is divided by 5.0 because it is a float

```
#include <stdio.h>
int main()
{
  int score[5] = {86, 72, 67, 94, 55};
  float average = 0.0;
  average = score[0] + score[1]
    + score[2] + score[3] + score[4];
  average = average / 5.0;
  printf("Average = %.2f\n", average);
  return 0;
}
```

Solution 2

- Elements are added using a FOR loop
- Total score is stored in variable average
- Variable average is divided by 5.0 because it is a float

```
#include <stdio.h>
int main()
{
  int score[5] = {86, 72, 67, 94, 55};
  float average = 0.0;
  int n;
  for (n = 0; n < 5; n++) {
    average += score[n];
  }
  average = average / 5.0;
  printf("Average = %.2f\n", average);
  return 0;
}
```

Pointers

Standard streams

- In a C program, there are three standard streams
- `stdin` – used for input buffering (keyboard)
- `stdout` – used for output buffering (terminal screen)
- `stderr` – used for output buffering (terminal screen)
- Streams are used with pointers, file input and output operations

Pointer

- In the computer memory, the data is stored at a specific memory address location
- A pointer points to the memory address location, and in doing so, also has access to the data
- Pointers can be processed using arithmetic
- Thus, making processing memory data more predictable

Pointer Location	Computer memory	
	Address	**Data**
	0x0000	0x8312
→	0x0001	0x4A34
	0x0002	0x0137
	…	…
	0xFFFE	0xCB86
	0xFFFF	0xDE15

Pointer declaration

- Pointers are used in a C program to access elements in an array or structure
- The pointer makes it easier to process an array with large number of elements
- Pointer's data type must match the array's data type

```
<data_type> *<identifier>;
```

Accessing data via pointer

- Access to the data in a memory address location using the pointer requires the following syntax
- Asterisk (*) is known as the dereferencing operator

```
*<identifier>
```

Pointer declaration example

```
int *p;  // integer pointer
float *q;  // float pointer
double *dp;  // double pointer
char *charptr;  // char pointer
```

Pointer usage

- Before the pointer can be used, it must be initialised to point to the memory address location of an element in an array (usually the element 0)
- Address pointer is indicated by ampersand symbol (&)
- Example:

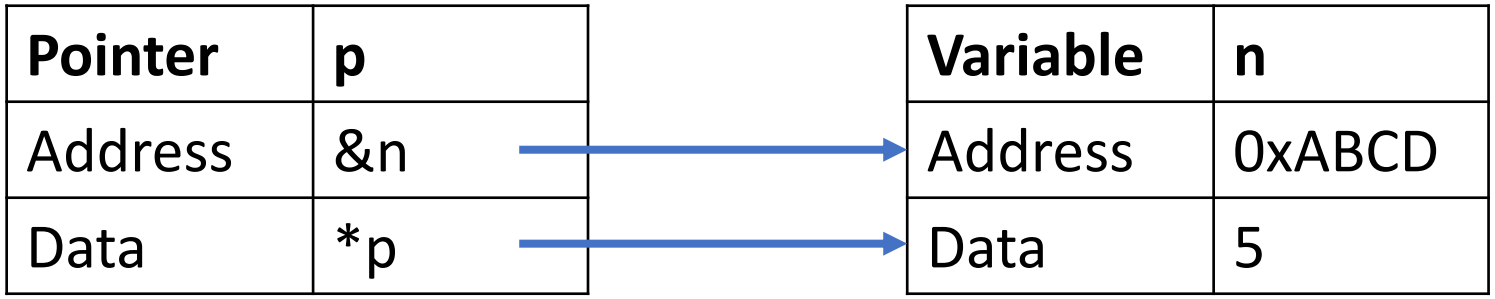
```
int n;  // declare variable n
int *p;  // declare pointer p
p = &n;  // set p to memory address of n
*p = 5;  // store data in n via p
```

NULL pointer

- A null pointer is initialised to NULL, in order to ensure the pointer does not point to any invalid object
- Commonly practised in functions
- Example:

```
int *p = NULL;
```

Pointer address and data



Pointer test program

- Pointer p is initialised with memory address location of variable n
- Data 5 is written into variable n via pointer p
- First two PRINTF() is used to show data and address stored in the pointer
- Last two PRINTF() is to verify data and address of variable n

```
#include <stdio.h>
int main()
{
   int n;
   int *p;
   p = &n;
   *p = 5;
   printf("Data *p = %d\n", *p);
   printf("Address p = %p\n", p);
   printf("Data n = %d\n", n);
   printf("Address n = %p\n", &n);
   return 0;
}
```

Pointer example 1

- The pointer p is used to point towards the memory address of variable n
- User enters a single integer value, which is stored in the variable n via pointer p
- Value in variable is added with 1 via pointer p

```
#include <stdio.h>
int main()
{
  int n, result;
  int *p;
  p = &n;
  printf("Enter value: ");
  scanf("%d", p);
  result = *p + 1;
  printf("Result = %d\n", result);
  return 0;
}
```

Pointer example 2

- Two pointers are created, each initialised to point towards the memory address of variables n & m
- The user enters the values and data is stored in the variables via pointers p and q
- Addition is performed using pointers as well
  - result = *p + *q

```
#include <stdio.h>
int main()
{
   int n, m, result;
   int *p, *q;
   p = &n;
   q = &m;
   printf("Enter value 1: ");
   scanf("%d", p);
   printf("Enter value 2: ");
   scanf("%d", q);
   result = *p + *q;
   printf("Sum = %d\n", result);
   return 0;
}
```

Array pointers

Array pointer

- Pointer can be used to point to the memory address location of an element in an array (usually the element 0)
- Memory address is indicated by symbol &
- Example:

```
int a[5];  // declare array a[]
int *p;  // declare pointer p
p = &a[0];  // initialise p to element a[0]
```

Processing the array pointer

- The pointer p points to the current address &a[0], which is element zero of the array
- Incrementing the pointer (p+1) will point to the next element a[1], and so on

Pointer	**p**	**p+1**	**p+2**	**p+3**	**p+4**
Array element	`a[0]`	`a[1]`	`a[2]`	`a[3]`	`a[4]`
Array data	1	2	3	4	5

Accessing data via pointer

- Access to the data in a memory address location using the pointer requires the dereferencing operator as well
- Pointer needs to be incremented or decremented to point to the next element in the array
- Example:

```
int a[3] = {10, 23, 56};
int *p;
p = &a[0];
printf("%d ", *p);
```

Array pointer example 1

- Pointer p is initialised to element 0 in array a[]
- Every element data is accessed using *p, and output to the terminal screen using PRINTF()
- After end PRINTF(), pointer p is incremented (p++) to point to next element in a[]

```
#include <stdio.h>
int main()
{
  int a[3] = {10, 23, 56};
  int *p;
  p = &a[0];
  printf("%d\n", *p);
  p++;
  printf("%d\n", *p);
  p++;
  printf("%d\n", *p);
  return 0;
}
```

Array pointer example 2

- Pointer p is initialised to element 0 in array a[]
- FOR loop runs five rounds to output every element data using *p
- At the end of each round, pointer p must be increment (p++) to point to next element in a[]

```
#include <stdio.h>
int main()
{
  int a[5] = {1, 2, 3, 4, 5};
  int n;
  int *p;
  p = &a[0];
  for (n = 0; n < 5; n++) {
    printf("%d\n", *p);
    p++;
  }
  return 0;
}
```

Array pointer example 3

- Pointer p points to element score[0]
- The value in the pointer is added to the variable average
- At the end of every round, the pointer must be incremented

```
#include <stdio.h>
int main()
{
   int score[5] = {86, 72, 67, 94, 55};
   int n, *p;
   float average = 0.0;
   p = &score[0];
   for (n = 0; n < 5; n++) {
      average += *p;
      p++;
   }
   average = average / 5.0;
   printf("Average = %.2f\n", average);
   return 0;
}
```

Array pointer example 4

- User enters 3 integers with spaces
- Initial FOR loop used to read integers
- The p++ increments pointer without a separate statement
- Pointer p must be initialised before use in both FOR loops

```
Enter 3 integers: 1 2 3
a[0] = 1
a[1] = 2
a[2] = 3
```

```
#include <stdio.h>
int main()
{
  int a[3] = {0};
  int n, *p;
  p = &a[0];
  printf("Enter 3 integers: ");
  for (n=0; n<3; n++) {
    scanf("%d", p++);
  }

  p = &a[0];
  for (n=0; n<3; n++) {
    printf("a[%d] = %d\n", n, *p);
    p++;
  }
  return 0;
}
```

Summary

- Arrays are used to store data of similar type with a single identifier
- Access to arrays can be direct or indirect (using pointers)
- Pointer is a form of indirect addressing
- NULL pointer is useful when a pointer must be initialised to a predictable value

Further readings

- C: How to Program, 8th Edition, Paul Deitel & Harvey Deitel
  - Arrays: pp. 246–262
  - Pointers: pp. 307–310
- The C Programming Language, 2nd Edition, Kernighan & Ritchie
  - Arrays and pointers: pp. 83–102