C Programming - Deck 15

Pointers and Arrays

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Arrays and Pointers Relationship

- Array name is a constant pointer to the first element
- `arr` is equivalent to `&arr[0]`
- `arr[i]` is equivalent to `*(arr + i)`
- `&arr[i]` is equivalent to `arr + i`
- Array name cannot be reassigned (constant pointer)
- Pointers can be reassigned to point to different elements
- Pointer arithmetic works naturally with arrays
- Understanding this relationship is fundamental to C programming

Program 1: Array Name as Pointer

```c
#include <stdio.h>
int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  printf("arr = %p\n", (void*)arr);
  printf("&arr[0] = %p\n", (void*)&arr[0]);
  printf("*arr = %d\n", *arr);
  printf("arr[0] = %d\n", arr[0]);
  printf("*(arr+2) = %d\n", *(arr+2));
  printf("arr[2] = %d\n", arr[2]);
  return 0;
}
```

Output:

```
arr = 0x7ffeeb3c4a00
&arr[0] = 0x7ffeeb3c4a00
*arr = 10
arr[0] = 10
*(arr+2) = 30
arr[2] = 30
```

arr and &arr[0] are the same address

Program 2: Pointer vs Array Name

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};
  int *ptr = arr;
  printf("arr points to: %d\n", *arr);
  printf("ptr points to: %d\n", *ptr);
  ptr++;
  printf("After ptr++: %d\n", *ptr);
  printf("arr still: %d\n", *arr);
  return 0;
}
```

Output:

```
arr points to: 1
ptr points to: 1
After ptr++: 2
arr still: 1
```

ptr can be modified, arr cannot

Program 3: Array Traversal with Pointer

```c
#include <stdio.h>
int main() {
  int arr[5] = {5, 10, 15, 20, 25};
  int *ptr;
  printf("Using pointer:\n");
  for (ptr = arr; ptr < arr + 5; ptr++) {
    printf("%d ", *ptr);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Using pointer:
5 10 15 20 25
```

Traversing array with pointer increment

Program 4: Pointer Notation vs Array Notation

```c
#include <stdio.h>
int main() {
  int arr[4] = {100, 200, 300, 400};
  int *ptr = arr;
  int i;
  printf("Array notation:\n");
  for (i = 0; i < 4; i++) {
    printf("arr[%d] = %d\n", i, arr[i]);
  }
  printf("Pointer notation:\n");
  for (i = 0; i < 4; i++) {
    printf("*(ptr+%d) = %d\n", i, *(ptr+i));
  }
  return 0;
}
```

Output:

```
Array notation:
arr[0] = 100
arr[1] = 200
arr[2] = 300
arr[3] = 400
Pointer notation:
*(ptr+0) = 100
*(ptr+1) = 200
*(ptr+2) = 300
*(ptr+3) = 400
```

Program 5: Sum of Array Using Pointer

```c
#include <stdio.h>
int main() {
  int arr[6] = {5, 10, 15, 20, 25, 30};
  int *ptr = arr;
  int sum = 0;
  int i;
  for (i = 0; i < 6; i++) {
    sum += *(ptr + i);
  }
  printf("Sum = %d\n", sum);
  return 0;
}
```

Output:

```
Sum = 105
```

Computing sum using pointer arithmetic

Program 6: Reverse Array Using Pointers

```c
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  int *left = arr;
  int *right = arr + 4;
  int temp, i;
  while (left < right) {
    temp = *left;
    *left = *right;
    *right = temp;
    left++;
    right--;
  }
  for (i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
5 4 3 2 1
```

Two-pointer approach to reverse

Program 7: Find Maximum Using Pointer

```c
#include <stdio.h>
int main() {
  int arr[6] = {23, 67, 12, 89, 45, 34};
  int *ptr = arr;
  int max = *ptr;
  int i;
  for (i = 1; i < 6; i++) {
    if (*(ptr + i) > max) {
      max = *(ptr + i);
    }
  }
  printf("Maximum = %d\n", max);
  return 0;
}
```

Output:

```
Maximum = 89
```

Finding max with pointer arithmetic

Program 8: Copy Array Using Pointers

```c
#include <stdio.h>
int main() {
  int src[5] = {10, 20, 30, 40, 50};
  int dest[5];
  int *ps = src;
  int *pd = dest;
  int i;
  for (i = 0; i < 5; i++) {
    *pd = *ps;
    ps++;
    pd++;
  }
  printf("Copied array: ");
  for (i = 0; i < 5; i++) {
    printf("%d ", dest[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Copied array: 10 20 30 40 50
```

Copying elements with two pointers

Array of Pointers

- Array where each element is a pointer
- Syntax: `int *arr[5];` - array of 5 integer pointers
- Each element can point to different variables
- Useful for managing multiple arrays or strings
- Different from pointer to array: `int (*ptr)[5];`
- Array of pointers: `int *arr[5]` - [] has higher precedence
- Pointer to array: `int (*ptr)[5]` - * is bound first by ()

Program 9: Array of Pointers

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20, c = 30;
  int *arr[3];
  arr[0] = &a;
  arr[1] = &b;
  arr[2] = &c;
  int i;
  for (i = 0; i < 3; i++) {
    printf("arr[%d] = %d\n", i, *arr[i]);
  }
  return 0;
}
```

Output:

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
```

Each array element points to a variable

Program 10: Modifying Through Array of Pointers

```c
#include <stdio.h>
int main() {
  int x = 5, y = 10, z = 15;
  int *ptrs[3] = {&x, &y, &z};
  int i;
  printf("Before:\n");
  for (i = 0; i < 3; i++) {
    printf("%d ", *ptrs[i]);
  }
  for (i = 0; i < 3; i++) {
    *ptrs[i] *= 2;
  }
  printf("\nAfter doubling:\n");
  printf("x=%d, y=%d, z=%d\n", x, y, z);
  return 0;
}
```

Output:

```
Before:
5 10 15
After doubling:
x=10, y=20, z=30
```

Modifying values through pointer array

Pointer to Array

- Single pointer that points to an entire array
- Syntax: `int (*ptr)[5];` - pointer to array of 5 ints
- Parentheses are essential: `(*ptr)`
- Without parentheses: `int *ptr[5]` is array of pointers
- Dereferencing gives the entire array
- `*ptr` gives first element address
- `**ptr` or `(*ptr)[0]` gives first element value
- Useful for 2D arrays and functions

Program 11: Pointer to Array

```c
#include <stdio.h>
int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  int (*ptr)[5] = &arr;
  int i;
  printf("Using pointer to array:\n");
  for (i = 0; i < 5; i++) {
    printf("(*ptr)[%d] = %d\n",
      i, (*ptr)[i]);
  }
  return 0;
}
```

Output:

```
Using pointer to array:
(*ptr)[0] = 10
(*ptr)[1] = 20
(*ptr)[2] = 30
(*ptr)[3] = 40
(*ptr)[4] = 50
```

ptr points to entire array

Program 12: Array of Pointers vs Pointer to Array

```c
#include <stdio.h>
int main() {
  int arr[3] = {1, 2, 3};
  int *ap[3];
  int (*pa)[3] = &arr;
  int i;
  for (i = 0; i < 3; i++) {
    ap[i] = &arr[i];
  }
  printf("Array of pointers:\n");
  for (i = 0; i < 3; i++) {
    printf("%d ", *ap[i]);
  }
  printf("\nPointer to array:\n");
  for (i = 0; i < 3; i++) {
    printf("%d ", (*pa)[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Array of pointers:
1 2 3
Pointer to array:
1 2 3
```

Different syntax, same result here

Program 13: Pointer Arithmetic with Arrays

```c
#include <stdio.h>
int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  int *p1 = arr;
  int *p2 = arr + 3;
  printf("*p1 = %d\n", *p1);
  printf("*p2 = %d\n", *p2);
  printf("p2 - p1 = %ld\n", p2 - p1);
  printf("*(p1 + 2) = %d\n", *(p1 + 2));
  printf("p2[-1] = %d\n", p2[-1]);
  return 0;
}
```

Output:

```
*p1 = 10
*p2 = 40
p2 - p1 = 3
*(p1 + 2) = 30
p2[-1] = 30
```

Negative indexing with pointers

Program 14: Search Element Using Pointer

```c
#include <stdio.h>
int main() {
  int arr[6] = {12, 45, 23, 67, 34, 89};
  int *ptr = arr;
  int key = 67;
  int found = 0;
  int i;
  for (i = 0; i < 6; i++) {
    if (*(ptr + i) == key) {
      printf("Found %d at index %d\n",
        key, i);
      found = 1;
      break;
    }
  }
  if (!found) {
    printf("%d not found\n", key);
  }
  return 0;
}
```

Output:

```
Found 67 at index 3
```

Linear search using pointer

Program 15: Count Even Numbers Using Pointer

```c
#include <stdio.h>
int main() {
  int arr[8] = {12, 15, 18, 21, 24, 27, 30, 33};
  int *ptr = arr;
  int count = 0;
  int i;
  for (i = 0; i < 8; i++) {
    if (*(ptr + i) % 2 == 0) {
      count++;
    }
  }
  printf("Even numbers count: %d\n", count);
  return 0;
}
```

Output:

```
Even numbers count: 4
```

Counting with condition check

Program 16: Multiply Array Elements by Scalar

```c
#include <stdio.h>
int main() {
  int arr[5] = {2, 4, 6, 8, 10};
  int *ptr = arr;
  int scalar = 3;
  int i;
  printf("Before: ");
  for (i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
  }
  for (i = 0; i < 5; i++) {
    *(ptr + i) *= scalar;
  }
  printf("\nAfter multiply by %d: ", scalar);
  for (i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Before: 2 4 6 8 10
After multiply by 3: 6 12 18 24 30
```

Scalar multiplication in-place

Program 17: Compare Two Arrays Using Pointers

```c
#include <stdio.h>
int main() {
  int arr1[5] = {1, 2, 3, 4, 5};
  int arr2[5] = {1, 2, 3, 4, 5};
  int *p1 = arr1;
  int *p2 = arr2;
  int equal = 1;
  int i;
  for (i = 0; i < 5; i++) {
    if (*(p1 + i) != *(p2 + i)) {
      equal = 0;
      break;
    }
  }
  if (equal) {
    printf("Arrays are equal\n");
  } else {
    printf("Arrays are not equal\n");
  }
  return 0;
}
```

Output:

```
Arrays are equal
```

Element-wise comparison

Program 18: Rotate Array Left Using Pointers

```c
#include <stdio.h>
int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  int *ptr = arr;
  int temp = *ptr;
  int i;
  for (i = 0; i < 4; i++) {
    *(ptr + i) = *(ptr + i + 1);
  }
  *(ptr + 4) = temp;
  printf("After left rotation: ");
  for (i = 0; i < 5; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
After left rotation: 20 30 40 50 10
```

Shifting elements left by one

Program 19: Merge Two Arrays Using Pointers

```c
#include <stdio.h>
int main() {
  int arr1[3] = {1, 2, 3};
  int arr2[3] = {4, 5, 6};
  int merged[6];
  int *p1 = arr1;
  int *p2 = arr2;
  int *pm = merged;
  int i;
  for (i = 0; i < 3; i++) {
    *(pm + i) = *(p1 + i);
  }
  for (i = 0; i < 3; i++) {
    *(pm + 3 + i) = *(p2 + i);
  }
  printf("Merged: ");
  for (i = 0; i < 6; i++) {
    printf("%d ", merged[i]);
  }
  printf("\n");
  return 0;
}
```

Output:

```
Merged: 1 2 3 4 5 6
```

Combining two arrays into one

Program 20: Print Array in Reverse Using Pointer

```c
#include <stdio.h>
int main() {
  int arr[6] = {10, 20, 30, 40, 50, 60};
  int *ptr = arr + 5;
  printf("Reverse order: ");
  while (ptr >= arr) {
    printf("%d ", *ptr);
    ptr--;
  }
  printf("\n");
  return 0;
}
```

Output:

```
Reverse order: 60 50 40 30 20 10
```

Traversing backward with pointer

Key Takeaways

- Array name is a constant pointer to first element
- `arr[i]` and `*(arr + i)` are equivalent
- Pointers can traverse arrays efficiently
- Array of pointers: `int *arr[n]` - each element is a pointer
- Pointer to array: `int (*ptr)[n]` - points to entire array
- Pointer arithmetic simplifies array operations
- Can use negative indexing with pointers: `ptr[-1]`
- Understanding arrays and pointers is crucial for C mastery