C Programming - Deck 14

Pointers Basics

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

What are Pointers?

- A pointer is a variable that stores the memory address of another variable
- Pointers allow direct memory access and manipulation
- Every variable in memory has an address
- Syntax: `datatype *pointer_name;`
- Example: `int *ptr;` declares a pointer to an integer

Pointer Operators

- **Address-of operator (&)**: Returns the memory address of a variable
- **Dereference operator (*)**: Accesses the value at the address stored in pointer
- Example: `int x = 10; int *p = &x;`
- `&x` gives the address of x
- `*p` gives the value stored at address p (which is x)
- p contains the address, `*p` contains the value

Program 1: Basic Pointer Declaration and Initialization

```c
#include <stdio.h>
int main() {
  int x = 42;
  int *ptr = &x;
  printf("Value of x: %d\n", x);
  printf("Address of x: %p\n", (void*)&x);
  printf("Value of ptr: %p\n", (void*)ptr);
  printf("Value at ptr: %d\n", *ptr);
  return 0;
}
```

Output:

```
Value of x: 42
Address of x: 0x7ffeeb3c4a1c
Value of ptr: 0x7ffeeb3c4a1c
Value at ptr: 42
```

Note: Actual addresses vary each run

Program 2: Changing Value Through Pointer

```c
#include <stdio.h>
int main() {
  int x = 10;
  int *ptr = &x;
  printf("Before: x = %d\n", x);
  *ptr = 20;
  printf("After: x = %d\n", x);
  printf("Value at ptr: %d\n", *ptr);
  return 0;
}
```

Output:

```
Before: x = 10
After: x = 20
Value at ptr: 20
```

Modifying *ptr modifies x

Program 3: Pointer to Different Data Types

```c
#include <stdio.h>
int main() {
  int i = 100;
  float f = 3.14;
  char c = 'A';
  int *pi = &i;
  float *pf = &f;
  char *pc = &c;
  printf("int: %d at %p\n", *pi, (void*)pi);
  printf("float: %.2f at %p\n", *pf, (void*)pf);
  printf("char: %c at %p\n", *pc, (void*)pc);
  return 0;
}
```

Output:

```
int: 100 at 0x7ffeeb3c4a1c
float: 3.14 at 0x7ffeeb3c4a18
char: A at 0x7ffeeb3c4a17
```

Each type has its own pointer type

Program 4: Pointer Size

```c
#include <stdio.h>
int main() {
  int *pi;
  float *pf;
  char *pc;
  double *pd;
  printf("Size of int pointer: %lu\n", sizeof(pi));
  printf("Size of float pointer: %lu\n", sizeof(pf));
  printf("Size of char pointer: %lu\n", sizeof(pc));
  printf("Size of double pointer: %lu\n", sizeof(pd));
  return 0;
}
```

Output:

```
Size of int pointer: 8
Size of float pointer: 8
Size of char pointer: 8
Size of double pointer: 8
```

All pointers same size (8 bytes on 64-bit)

Program 5: NULL Pointer

```c
#include <stdio.h>
int main() {
  int *ptr = NULL;
  printf("ptr value: %p\n", (void*)ptr);
  if (ptr == NULL) {
    printf("Pointer is NULL\n");
  }
  int x = 50;
  ptr = &x;
  if (ptr != NULL) {
    printf("Pointer now points to: %d\n", *ptr);
  }
  return 0;
}
```

Output:

```
ptr value: 0x0
Pointer is NULL
Pointer now points to: 50
```

Always check for NULL before dereferencing

Program 6: Swap Two Numbers Using Pointers

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20;
  int *p1 = &a, *p2 = &b;
  int temp;
  printf("Before: a=%d, b=%d\n", a, b);
  temp = *p1;
  *p1 = *p2;
  *p2 = temp;
  printf("After: a=%d, b=%d\n", a, b);
  return 0;
}
```

Output:

```
Before: a=10, b=20
After: a=20, b=10
```

Swapping values through pointers

Pointer Arithmetic

- Pointers support arithmetic operations: +, -, ++, –
- Adding 1 to a pointer moves it to the next element
- Increment depends on the data type size
- `int *p; p++;` moves p by `sizeof(int)` bytes
- `char *p; p++;` moves p by `sizeof(char)` bytes (1 byte)
- `p + n` moves pointer by n elements (not n bytes)
- Pointer subtraction gives number of elements between pointers

Program 7: Pointer Arithmetic - Increment

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

Output:

```
ptr points to: 10
After ptr++: 20
After ptr++: 30
ptr+2 points to: 50
```

Pointer moves by sizeof(int) each time

Program 8: Pointer Arithmetic - Address Difference

```c
#include <stdio.h>
int main() {
  int arr[5] = {1, 2, 3, 4, 5};
  int *p1 = &arr[0];
  int *p2 = &arr[4];
  printf("p1 address: %p\n", (void*)p1);
  printf("p2 address: %p\n", (void*)p2);
  printf("p2 - p1 = %ld elements\n",
    p2 - p1);
  printf("Byte difference: %ld\n",
    (char*)p2-(char*)p1);
  return 0;
}
```

Output:

```
p1 address: 0x7ffeeb3c4a00
p2 address: 0x7ffeeb3c4a10
p2 - p1 = 4 elements
Byte difference: 16
```

4 elements $\times$ 4 bytes/int $=$ 16 bytes

Program 9: Pointer Comparison

```c
#include <stdio.h>
int main() {
  int arr[5] = {10, 20, 30, 40, 50};
  int *p1 = &arr[1];
  int *p2 = &arr[3];
  if (p1 < p2) {
    printf("p1 comes before p2\n");
  }
  if (p1 == &arr[1]) {
    printf("p1 points to arr[1]\n");
  }
  printf("Distance: %ld elements\n", p2 - p1);
  return 0;
}
```

Output:

```
p1 comes before p2
p1 points to arr[1]
Distance: 2 elements
```

Pointers can be compared for ordering

Program 10: Traversing Array with Pointer

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

Output:

```
Using pointer arithmetic:
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

*(ptr + i) is equivalent to arr[i]

Program 11: Pointer Increment in Loop

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

Output:

```
Array elements:
5 10 15 20 25
```

Moving pointer through array with ++

Program 12: Pointer to Pointer

```c
#include <stdio.h>
int main() {
  int x = 100;
  int *ptr = &x;
  int **pptr = &ptr;
  printf("Value of x: %d\n", x);
  printf("*ptr: %d\n", *ptr);
  printf("**pptr: %d\n", **pptr);
  printf("Address in ptr: %p\n",
    (void*)ptr);
  printf("Address in pptr: %p\n",
    (void*)pptr);
  return 0;
}
```

Output:

```
Value of x: 100
*ptr: 100
**pptr: 100
Address in ptr: 0x7ffeeb3c4a1c
Address in pptr: 0x7ffeeb3c4a10
```

Pointer to pointer - double indirection

Program 13: Finding Sum Using Pointers

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

Output:

```
Sum of array: 150
```

Accumulating sum with pointer traversal

Program 14: Finding Maximum Using Pointers

```c
#include <stdio.h>
int main() {
  int arr[5] = {23, 67, 12, 89, 45};
  int *ptr = arr;
  int max = *ptr;
  int i;
  for (i = 1; i < 5; i++) {
    ptr++;
    if (*ptr > max) {
      max = *ptr;
    }
  }
  printf("Maximum element: %d\n", max);
  return 0;
}
```

Output:

```
Maximum element: 89
```

Finding max with pointer traversal

Program 15: 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--;
  }
  printf("Reversed: ");
  for (i = 0; i < 5; i++) printf("%d ", arr[i]);
  printf("\n");
  return 0;
}
```

Output:

```
Reversed: 5 4 3 2 1
```

Two pointers moving towards each other

Common Pointer Errors

- **Uninitialized pointers**: Using pointer before assigning address
- **Dangling pointers**: Pointer to memory that has been freed
- **Null pointer dereference**: Dereferencing NULL pointer causes crash
- **Wild pointers**: Pointer with garbage value
- **Memory leaks**: Allocated memory not freed (covered in later deck)
- **Buffer overflow**: Accessing beyond array bounds
- Always initialize pointers before use
- Check for NULL before dereferencing

Program 16: Uninitialized Pointer Error (DON'T DO THIS)

```c
#include <stdio.h>
int main() {
  int *ptr;
  printf("This will crash or show garbage\n");
  printf("Value: %d\n", *ptr);
  return 0;
}
```

Output (wrong):

```
Segmentation fault (crash)
OR
Value: 1234567 (garbage)
```

Correct version:

```c
#include <stdio.h>
int main() {
  int x = 42;
  int *ptr = &x;
  printf("Value: %d\n", *ptr);
  return 0;
}
```

Output (correct):

```
Value: 42
```

Always initialize pointers!

Program 17: Generic Pointer (void*)

```c
#include <stdio.h>
int main() {
  int x = 100;
  float f = 3.14;
  void *ptr;
  ptr = &x;
  printf("int value: %d\n", *(int*)ptr);
  ptr = &f;
  printf("float value: %.2f\n", *(float*)ptr);
  printf("void* can point to any type\n");
  return 0;
}
```

Output:

```
int value: 100
float value: 3.14
void* can point to any type
```

void* must be cast before dereferencing

Program 18: Constant Pointer vs Pointer to Constant

```c
#include <stdio.h>
int main() {
  int a = 10, b = 20;
  const int *ptr1 = &a;
  int *const ptr2 = &a;
  printf("ptr1 points to: %d\n", *ptr1);
  ptr1 = &b;
  printf("ptr1 now: %d\n", *ptr1);
  *ptr2 = 30;
  printf("ptr2 value: %d\n", *ptr2);
  return 0;
}
```

Output:

```
ptr1 points to: 10
ptr1 now: 20
ptr2 value: 30
```

ptr1: pointer can change, value can't
ptr2: pointer can't change, value can

Program 19: 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] points to: %d\n", i, *arr[i]);
  }
  return 0;
}
```

Output:

```
arr[0] points to: 10
arr[1] points to: 20
arr[2] points to: 30
```

Array where each element is a pointer

Program 20: Pointer Indirection Levels

```c
#include <stdio.h>
int main() {
  int x = 42;
  int *p1 = &x;
  int **p2 = &p1;
  int ***p3 = &p2;
  printf("x = %d\n", x);
  printf("*p1 = %d\n", *p1);
  printf("**p2 = %d\n", **p2);
  printf("***p3 = %d\n", ***p3);
  ***p3 = 100;
  printf("After change: x = %d\n", x);
  return 0;
}
```

Output:

```
x = 42
*p1 = 42
**p2 = 42
***p3 = 42
After change: x = 100
```

Multiple levels of indirection

Key Takeaways

- Pointers store memory addresses of variables
- Use & to get address, * to dereference (get value)
- Pointer arithmetic moves by element size, not bytes
- Always initialize pointers before use
- NULL pointers should be checked before dereferencing
- Pointer type must match the variable type it points to
- Can have pointers to pointers (multiple indirection)
- Pointers enable efficient array manipulation
- Common errors: uninitialized, NULL dereference, dangling pointers