CSE 06131223 ♦ CSE 06131224

Structured Programming

Lecture 22

Pointers in C (3)

Prepared by

Md. Mijanur Rahman, Prof. Dr.

Dept. of Computer Science and Engineering

Jatiya Kabi Kazi Nazrul Islam University, Bangladesh

www.mijanrahman.com

Contents

Pointers in C

- What is Pointer in C?
- Memory Organization of Pointers
- Accessing the Address of a Variable
- Declaration of Pointer Variables
- Initialization of Pointer Variables
- Accessing a Variable Through Its Pointer
- Pointer Expressions
- Pointers and Arrays, and Array of Pointers
- Pointers and Strings
- **Pointers and Functions**
- **Pointers and Structures**

Parameter Passing to Functions

- The parameters passed to function are called ***actual parameters***. The parameters received by function are called ***formal parameters***.
- There are two most popular ways to pass parameters.
- ***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.
- ***Pass by Reference*** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Parameter Passing to Functions

- **Pass by value:**
  - Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

```c
#include <stdio.h>
void fun(int x)
{
    x = 30;
}

int main(void)
{
    int x = 20;
    fun(x);
    printf("x = %d", x);
    return 0;
}
```

Output:

```
x = 20
```

Pointers as Function Arguments

- **Pass by reference:**
  - In C, we can use pointers to get the effect of pass-by reference. For example, consider the below program. The function **fun()** expects a pointer **ptr** to an integer (or an address of an integer). It modifies the value at the address **ptr**. The address operator **&** is used to get the address of a variable of any data type.

```c
# include <stdio.h>
void fun(int *ptr)
{
    *ptr = 30;
}

int main()
{
  int x = 20;
  fun(&x);
  printf("x = %d", x);

  return 0;
}
```

Output:

```
x = 30
```

Pointers as Function Arguments

- Hence, pointers as function arguments in programming languages like C and C++ allow us to pass a memory address instead of a value. This can be beneficial for several reasons:
  - **Passing by reference:** By passing a pointer to a function, we can modify the original variable's value within that function. This is particularly useful when we need to alter the value of a variable inside a function and have those changes reflected outside the function.
  - **Reduced memory overhead:** When we pass large data structures to functions, passing pointers instead of the entire data structure can reduce memory usage and improve performance. This is because we're passing just a memory address instead of copying the entire data.
  - **Efficiency:** Pointers allow us to directly manipulate memory, which can be more efficient than copying large data structures.
  - **Dynamic memory allocation:** Pointers are essential when dealing with dynamically allocated memory. Functions can use pointers to allocate and deallocate memory dynamically, allowing for more flexible memory management.

Pointers as Function Arguments

- Here's a simple example in C:

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    printf("Before swapping: x = %d, y = %d\n", x, y);
    swap(&x, &y); // Passing addresses of x and y
    printf("After swapping: x = %d, y = %d\n", x, y);
    return 0;
}
```

- In this example, swap() takes two integer pointers as arguments. Inside swap(), it dereferences these pointers to access and modify the values they point to, effectively swapping the values of x and y.

Terminal

```
Before swapping: x = 5, y = 10
After swapping: x = 10, y = 5
```

Function Returning Pointers

- Returning pointers from functions in languages like C and C++ can be powerful as it allows functions to allocate memory dynamically and return the address of that memory. Here's why returning pointers can be useful:
  - **Dynamic Memory Allocation:** Functions can allocate memory dynamically using functions like malloc() or new (in C++), and return a pointer to the allocated memory. This allows for flexible memory management, especially when the size of the data to be returned is not known at compile time.
  - **Returning Multiple Values:** Pointers enable functions to return multiple values by pointing to a data structure (e.g., an array or a struct) that holds these values. This is especially useful when you need to return more than one value from a function.
  - **Accessing Dynamically Allocated Data:** Functions can return pointers to dynamically allocated data structures (e.g., arrays, linked lists, trees), allowing the caller to access and manipulate this data.

Function Returning Pointers

- **Dynamic Memory Allocation and Accessing Dynamically Allocated Data**
- Here's a simple example in C:

- In this example, the createArray() function dynamically allocates memory for an array of integers, initializes its elements, and then returns a pointer to the first element of the array. The main() function calls createArray() to get a pointer to the dynamically allocated array, prints its elements, and then frees the allocated memory using free().

```c
#include <stdio.h>
#include <stdlib.h>
int *createArray(int size) {
    int *arr = (int *)malloc(size * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    for (int i = 0; i < size; i++) {
        printf("Enter Element-%d: ", i+1);
        scanf("%d", &arr[i]);
    }
    return arr;
}
int main() {
    int size = 5;
    int *arr = createArray(size);

    printf("Array elements: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

```
Terminal
Enter Element-1: 10
Enter Element-2: 20
Enter Element-3: 30
Enter Element-4: 40
Enter Element-5: 50
Array elements: 10 20 30 40 50
```

Function Returning Pointers

- **Functions that return multiple values.**
- **In C, functions inherently return only one value. However, we can simulate the concept of returning multiple values by using pointers.**
  - **Using Pointers:** Pass pointers to variables to the function, which will modify the values stored at those memory locations.

Function Returning Pointers

- Functions that return multiple values:
- **Using Pointers:** Pass pointers to variables to the function

```
Terminal
Enter Number-1: 20
Enter Number-2: 8
Sum: 28
Diff: 12
Product: 160
Div: 2.50
```

```c
#include <stdio.h>
void calculator(int a, int b, int *sum, int *sub, int *prod, float *dv){
    *sum = a + b;
    *sub = a - b;
    *prod = a * b;
    *dv = (float) a / b;
}

int main() {
    int num1, num2, sum, sub, prod;
    float dv;
    printf("Enter Number-1: ");
    scanf("%d", &num1);
    printf("Enter Number-2: ");
    scanf("%d", &num2);

    calculator(num1, num2, &sum, &sub, &prod, &dv);

    printf("Sum: %d\n", sum);
    printf("Diff: %d\n", sub);
    printf("Product: %d\n", prod);
    printf("Div: %.2f\n", dv);
    return 0;
}
```

Pointers and Structures

- Using structures and pointers together in C is common and powerful. Pointers allow us to dynamically allocate memory for structures, access and modify structure members efficiently, and pass structures to functions by reference.
- The usage of structures and pointers together:

```
struct Student {
    char name[50];
    int age;
    float gpa;
} *std;
```

- Accessing structure members:

```
std->name
std->age
std->gpa
```

Pointers and Structures

- Example of demonstrating how to use structures and pointers together in C:

```c
#include <stdio.h>
struct Book {
    char title[100];
    char author[100];
    int year;
};

void displayBook(struct Book *B) {
    printf("Title: %s\n", B->title);
    printf("Author: %s\n", B->author);
    printf("Year: %d\n", B->year);
}

int main() {
    struct Book myBook = {"Programming & Software Development", "M.M.
        Rahman", 2024};

    displayBook(&myBook);

    return 0;
}
```

Terminal

```
Title: Programming & Software Development
Author: M.M. Rahman
Year: 2024
```

THE END