Chapter 3 - Pointers in C

Introduction

Pointers are variables that store the memory address of other variables.

- They are one of the most powerful features of the C programming language. With pointers, you can directly manipulate data in memory, access hardware, and create complex data structures.
- However, with great power comes great responsibility. Pointers can be tricky to use, and if not used correctly, can cause crashes, memory leaks, and other difficult-to-debug errors.

Understanding pointers

- Before delving into pointers, it's important to understand how memory works in C.
- Every variable in C is stored in a specific memory location. When you declare a variable, the compiler assigns a memory address to it. Pointers allow you to access and manipulate the data stored at a given memory address directly.
- They are declared using the `*` symbol, and can be dereferenced using the `*` operator.
- Understanding how to use pointers is essential for advanced C programming, but requires careful attention to avoid common pitfalls and errors.

Advantage of using pointer

Advantages of using pointers in C:

- Pointers allow direct manipulation of data in memory.
- Pointers can be used to access hardware and memory-mapped I/O.
- Pointers can be used to create complex data structures, such as linked lists and trees.
- Pointers can reduce memory usage and improve program performance by passing pointers instead of large data structures to functions.

- Pointers can be used to create dynamic data structures that can be resized at runtime.

Accessing the address of a variable

To access the address of a variable in C, you can use the `&` operator. The following steps show how to access the address of a variable:

1. Declare a variable of any data type
2. Use the `&` operator followed by the variable name to get the address of the variable
3. Assign the address to a pointer variable using the `` symbol

For example, to get the address of an integer variable named `num`:

```
int num = 42;
int *ptr = &num;
```

The variable `ptr` is now a pointer to the memory location where the value `42` is stored.

Declaring and initializing pointers

To declare a pointer in C, use the `*` symbol, followed by the name of the pointer variable. The following syntax can be used to declare a pointer:

```
data_type *pointer_name;
```

For example, to declare a pointer to an integer variable:

```
int *ptr;
```

To initialize a pointer, you can assign it the memory address of a variable using the `&` operator. For example, to initialize a pointer named `ptr` to point to an integer variable named `num`:

```
int num = 42;
int *ptr = &num;
```

The variable `ptr` is now a pointer to the memory location where the value `42` is stored.

Here are two basic pointer examples for students:

Example 1: Pointer to an integer

```
#include <stdio.h>

int main() {
    int num = 42;
    int *ptr = &num;

    printf("The value of num is %d\\n", num);
    printf("The value of ptr is %p\\n", ptr);
    printf("The value pointed to by ptr is %d\\n", *ptr);

    return 0;
}
```

Output:

```
The value of num is 42
The value of ptr is 0x7fff5fbff74c
The value pointed to by ptr is 42
```

Example 2: Pointer to a character

```
#include <stdio.h>

int main() {
    char ch = 'A';
    char *ptr = &ch;

    printf("The value of ch is %c\\n", ch);
    printf("The value of ptr is %p\\n", ptr);
    printf("The value pointed to by ptr is %c\\n", *ptr);

    return 0;
}
```

Output:

```
The value of ch is A
The value of ptr is 0x7fff5fbff74f
The value pointed to by ptr is A
```

Dereference

- Dereferencing a pointer means accessing the value stored at the memory address pointed to by the pointer.
- This is done using the `*` operator.
- For example, if `ptr` is a pointer to an integer variable, you can dereference it to access the value of the variable like this: `int value = *ptr;`

Note that if you attempt to dereference a null pointer or a pointer that has not been initialized, your program will likely crash.

Accessing a variable through pointer

To access a variable through a pointer in C, you can use the `*` operator to dereference the pointer. This allows you to read or write to the variable being pointed to. Here are some examples:

```
// Example 1: Accessing an integer variable through a pointer
int num = 42;
int *ptr = &num;

// To read the value of num through the pointer, use the * operator
int value = *ptr;
printf("The value of num is %d\n", value);

// To write a new value to num through the pointer, use the * operator
*ptr = 100;
printf("The new value of num is %d\n", num);

// Example 2: Accessing a character variable through a pointer
char ch = 'A';
char *ptr = &ch;

// To read the value of ch through the pointer, use the * operator
char value = *ptr;
printf("The value of ch is %c\n", value);

// To write a new value to ch through the pointer, use the * operator
*ptr = 'B';
printf("The new value of ch is %c\n", ch);
```

In these examples, we declare a pointer variable `ptr` that points to an existing variable, `num` or `ch`. We then use the `*` operator to read or write to the value being pointed to. Note that changes made to the value through the pointer are reflected in the original variable.

Pointer expressions

Pointer expressions are any expressions involving pointers, including arithmetic operations, comparison operations, and assignment operations. Here are some examples:

- Arithmetic operations:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;
int sum = (*ptr) + (*(ptr + 1)) + (*(ptr + 2)) + (*(ptr + 3)) + (*(ptr + 4));
printf("The sum of the array is %d\n", sum);
```

- Comparison operations:

```
int a = 42;
int b = 69;
int *ptr_a = &a;
int *ptr_b = &b;

if (ptr_a < ptr_b) {
    printf("ptr_a is less than ptr_b\\n");
} else {
    printf("ptr_a is greater than or equal to ptr_b\\n");
}
```

- Assignment operations:

```
int a = 42;
int b = 69;
int *ptr_a = &a;
int *ptr_b = &b;

ptr_a = ptr_b;
printf("The value of a is now %d\n", a);
```

These are just a few examples of pointer expressions. Pointers are a powerful tool in C programming, but require careful attention to use correctly.

Pointer increments and scale factor

- Pointer increments and scale factor can be used to access elements of an array.
- When a pointer is incremented, it points to the next element in the array, and the scale factor is used to determine the number of bytes between elements.
- For example, to print all the elements of an integer array using pointer increments and scale factor:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;

for (int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
}
printf("\n");
```

This will print the values `1 2 3 4 5` to the console.

Pointers and arrays

In C, arrays are implemented as a sequence of contiguous memory locations. Pointers and arrays are closely related, and in fact, the name of an array is a pointer to the first element of the array.

Here are two examples of using pointers and arrays in C:

```
// Example 1: Using a pointer to iterate over an array
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;

for (int i = 0; i < 5; i++) {
    printf("%d ", *(ptr + i));
}
printf("\n");

// Example 2: Passing an array to a function
void print_array(int *arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", *(arr + i));
    }
    printf("\n");
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    print_array(arr, 5);
```

```
    return 0;
}
```

In the first example, we use a pointer variable `ptr` to iterate over the array. We assign the address of the first element of the array to the pointer, and use pointer arithmetic to access the remaining elements.

In the second example, we define a function `print_array` that takes a pointer to an array and the size of the array as arguments. We use pointer arithmetic to iterate over the array and print out each element.

Note that in both examples, we use the `*` operator to dereference the pointer and access the value being pointed to.

Pointers and character strings

- Character strings are implemented as arrays of characters, with a null terminator character (`\0`) at the end.
- Because arrays are implemented as pointers to the first element, character strings and pointers are closely related. To declare a character string, you can use the `char` data type, followed by square brackets to indicate the size of the array.

For example:

```
char str[6] = "Hello";
```

- This declares a character string `str` with a length of 6 (including the null terminator character), and initializes it with the value `"Hello"`.
- To print the value of the string, you can use the `%s` format specifier in a `printf` statement:

```
printf("The value of str is: %s\n", str);
```

- This will print the value `"Hello"` to the console.
- Note that because arrays are implemented as pointers to the first element, you can also use pointer arithmetic to access individual characters in the string:

```
char *ptr = str;
printf("The first character of str is: %c\n", *ptr);
```

This will print the value `"H"` to the console.

Pointers and Functions

- Pointers can be used as arguments in functions, allowing functions to directly modify data in memory.
- This can be useful for functions that need to modify large data structures or arrays.
- When passing a pointer to a function, the function can use pointer arithmetic to access and modify the data being pointed to.

Here is a simple example of using a pointer in C:

```
#include <stdio.h>

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

int main() {
    int x = 42;
    int y = 69;

    printf("Before swap: x = %d, y = %d\\n", x, y);

    swap(&x, &y);

    printf("After swap: x = %d, y = %d\\n", x, y);

    return 0;
}
```

- In this example, we define a function `swap` that takes two pointer arguments `a` and `b`. The function uses pointer arithmetic to access the values being pointed to, and swaps them.
- In the `main` function, we declare two integer variables `x` and `y`, and print their values to the console. We then call the `swap` function, passing in the memory

addresses of `x` and `y`. After the function call, we print the updated values of `x` and `y`.

Note that changes made to the values through the pointers are reflected in the original variables. In this way, pointers allow functions to modify data in memory directly, without needing to pass large data structures as arguments.

Pointers and structures

- Pointers can also be used with structures in C. To declare a pointer to a structure.

Syntax:

```
struct struct_name *ptr;
```

For example, to declare a pointer to a structure named `person`:

```
struct person *ptr;
```

- To access members of a structure through a pointer, use the `->` operator. For example, to access the `name` member of a structure pointed to by `ptr`:

```
struct person {
    char name[50];
    int age;
};

struct person *ptr;

printf("The name is %s\n", ptr->name);
```

- This will print the value of the `name` member to the console.
- Pointer arithmetic can also be used with structures. For example, to access the second element of an array of structures:

```
struct person people[2] = {{"Itachi", 25}, {"Naruto", 30}};
struct person *ptr = people;

printf("The name of the second person is %s\n", (ptr + 1)->name);
```

- This will print the value `"Naruto"` to the console.
- Using pointers with structures can be a powerful way to manipulate complex data structures in C, but requires careful attention to avoid common errors and pitfalls.

Dynamic Memory Allocation and Linked List

The concept of dynamic memory allocation in C language enables the programmer to allocate memory at runtime. This is made possible by four functions in the stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

Static memory allocation	Dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

- The malloc() function allocates single block of requested memory.

- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

`ptr=(cast-type*)malloc(byte-size)`

Here's a simple example of using the `malloc` function to allocate memory for an integer variable:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr;

    // allocate memory for one integer variable
    ptr = (int*) malloc(sizeof(int));

    // check if allocation was successful
    if (ptr == NULL) {
        printf("Memory allocation failed\\\\n");
        return 1;
    }

    // assign a value to the variable
    *ptr = 42;

    // print out the value
    printf("The value of the variable is %d\\\\n", *ptr);

    // free the allocated memory
    free(ptr);

    return 0;
}
```

This program declares a pointer variable `ptr`, and uses the `malloc` function to allocate memory for an integer variable. It then checks if the allocation was successful, assigns a value to the variable, prints out the value, and frees the allocated memory using the `free` function. Note that failure to free allocated memory can lead to memory leaks and other errors.

Example : Store values in allocated memory.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
```

```
    int *ptr;
    int n;

    printf("Enter the number of integers you want to store: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int)); // allocate memory using malloc

    if(ptr == NULL) { // check if memory allocation was successful
        printf("Error: Memory allocation failed.\n");
        exit(1);
    }

    printf("Enter %d integers:\n", n);

    for(int i=0; i<n; i++) {
        scanf("%d", &ptr[i]); // read in integers and store them in allocated memory
    }

    printf("The integers you entered are:\n");

    for(int i=0; i<n; i++) {
        printf("%d ", ptr[i]); // print out the integers stored in allocated memory
    }

    free(ptr); // free the allocated memory

    return 0;
}
```

```
Enter the number of integers you want to store: 5
Enter 5 integers:
10
20
30
40
50
The integers you entered are:
10 20 30 40 50
```

calloc() function

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

`ptr=(cast-type*)calloc(number, byte-size)`

Let's see the example of calloc() function.

Example : Using calloc() function.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr;
    int n;

    printf("Enter the number of integers you want to store: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int)); // allocate memory using calloc

    if(ptr == NULL) { // check if memory allocation was successful
        printf("Error: Memory allocation failed.\\n");
        exit(1);
    }

    printf("Enter %d integers:\\n", n);

    for(int i=0; i<n; i++) {
        scanf("%d", &ptr[i]); // read in integers and store them in allocated memory
    }

    printf("The integers you entered are:\\n");

    for(int i=0; i<n; i++) {
        printf("%d ", ptr[i]); // print out the integers stored in allocated memory
    }

    free(ptr); // free the allocated memory

    return 0;
}
```

- This program declares a pointer variable `ptr`, and uses the `calloc` function to allocate memory for an integer variable. It then checks if the allocation was successful, assigns a value to the variable, prints out the value, and frees the allocated memory using the `free` function. Note that failure to free allocated memory can lead to memory leaks and other errors.

Example : Using realloc() function.

```
#include<stdio.h>
  #include<stdlib.h>
int main(){
 int n,i,*ptr,sum=0;
```

```
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

```
Enter number of elements: 5
Enter elements of array: 10
50
30
20
40
Sum=150
```

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

```
ptr=realloc(ptr,  new size)
```

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

```
free(ptr)
```