C Programming and Data Structures

Computer Science and Technology-B

UNIT III Pointers and Structure

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Pointers:

What are Pointers?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

The general form of a pointer variable declaration is −

type \*var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

Here is how we can declare pointers.

int\* p;

Here, we have declared a pointer p of int type.

Assigning addresses to Pointers

Let's take an example.

int\* pc, c;

c = 5;

pc = &c;

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

Example Time: Swapping two numbers using Pointer

#include <stdio.h>

void swap(int \*a, int \*b);

int main()

{

int m = 10, n = 20;

printf("m = %d\n", m);

printf("n = %d\n\n", n);

swap(&m, &n); //passing address of m and n to the swap function

printf("After Swapping:\n\n");

printf("m = %d\n", m);

printf("n = %d", n);

return 0;

}

/\*

pointer 'a' and 'b' holds and

points to the address of 'm' and 'n'

\*/

void swap(int \*a, int \*b)

{

int temp;

temp = \*a;

\*a = \*b;

\*b = temp;

}

Output:

m = 10

n = 20

After Swapping:

m = 20

n = 10

Call by value:

In C everything is passed by value. That is, whatever you give as an argument to a function, it will be copied into the scope of that function. For instance, calling a function void foo(int) with foo(x) copies the value of x as the parameter of foo.  
Let’s explain it better with the well-known example of a function that swaps the content of two variables:

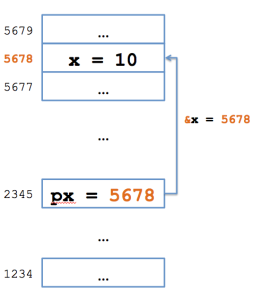
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | void swap(int x, int y) {      int tmp = x;      x = y;      y = tmp;  }    int main() {      int m = 6;      int n = 10;      printf("Before swapping (m,n) evaluate to: (%d,%d)\n",m,n);      swap(m,n);      printf("After swapping (m,n) evaluate to: (%d,%d)\n",m,n); // m and n still equal to their original values  } |

In the example above the two arguments m,n are copied from the main “into” the scope of the function swap. This means that the swapping that occurs within that function “lives” until swap returns. In fact, m,n within the main still preserve their original values.

However, function arguments can be of any type (i.e., not only primitive types like int above).  
In C a very useful type is a pointer to some other type. For instance, we could define the following:

|  |  |
| --- | --- |
| 1  2 | int x = 10;  int\* px = &x; |

The first one is the declaration of an integer variable x whereas the second statement declares a variable px of type int\*, that is “pointer to an integer”. Moreover, the pointer px is assigned to the address of the integer variable x via the reference operator (&).  
To get rid of the difference between a variable of a certain type and a pointer variable to that type, please refer to the following Figure:

[](https://gabrieletolomei.files.wordpress.com/2014/02/pointer.png)

Now, when passing a pointer to a function, you are still passing it by value. Indeed, the value of the pointer variable is copied into the function. In the example above, this means that a copy of px, namely a copy of the address of x (e.g., 5678), is passed to the function.  
It turns out that modifying that pointer inside the function will not change the pointer outside the function (i.e., outside the pointer will still contain the address of x). However, if you modify the object which the pointer points to (i.e., x) from within the function then the object itself results modified also outside the function.

C Call by Reference: Using pointers

In C programming, it is also possible to pass addresses as arguments to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

Example: Call by reference

#include <stdio.h>

void swap(int \*n1, int \*n2);

int main()

{

int num1 = 5, num2 = 10;

// address of num1 and num2 is passed

swap( &num1, &num2);

printf("num1 = %d\n", num1);

printf("num2 = %d", num2);

return 0;

}

void swap(int\* n1, int\* n2)

{

int temp;

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

}

When you run the program, the output will be:

num1 = 10

num2 = 5

The address of num1 and num2 are passed to the swap() function using swap(&num1, &num2);.

Pointers n1 and n2 accept these arguments in the function definition.

void swap(int\* n1, int\* n2) {

... ..

}

When \*n1 and \*n2 are changed inside the swap() function, num1 and num2 inside the main() function are also changed.

Inside the swap() function, \*n1 and \*n2 swapped. Hence, num1 and num2 are also swapped.

Notice that, swap() is not returning anything; its return type is void.

This technique is known as call by reference in C programming.

C Dynamic Memory Allocation

Dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

C malloc()

The name "malloc" stands for memory allocation.

The malloc() function reserves a block of memory of the specified number of bytes. And, it returns a [pointer](https://www.programiz.com/c-programming/c-pointers) of void which can be casted into pointers of any form.

Syntax of malloc()

ptr = (castType\*) malloc(size);

Example

ptr = (float\*) malloc(100 \* sizeof(float));

The above statement allocates 400 bytes of memory. It's because the size of float is 4 bytes. And, the pointer ptr holds the address of the first byte in the allocated memory.

The expression results in a NULL pointer if the memory cannot be allocated.

C calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.

Syntax of calloc()

ptr = (castType\*)calloc(n, size);

Example:

ptr = (float\*) calloc(25, sizeof(float));

The above statement allocates contiguous space in memory for 25 elements of type float.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

Example 1: malloc() and free()

// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>

#include <stdlib.h>

int main()

{

int n, i, \*ptr, sum = 0;

printf("Enter number of elements: ");

scanf("%d", &n);

ptr = (int\*) malloc(n \* sizeof(int));

// if memory cannot be allocated

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");

for(i = 0; i < n; ++i)

{

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

// deallocating the memory

free(ptr);

return 0;

}

Here, we have dynamically allocated the memory for n number of int.

Example 2: calloc() and free()

// Program to calculate the sum of n numbers entered by the user

#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));

if(ptr == NULL)

{

printf("Error! memory not allocated.");

exit(0);

}

printf("Enter elements: ");

for(i = 0; i < n; ++i)

{

scanf("%d", ptr + i);

sum += \*(ptr + i);

}

printf("Sum = %d", sum);

free(ptr);

return 0;

}

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Syntax of realloc()

ptr = realloc(ptr, x);

Here, ptr is reallocated with a new size x.

Example 3: realloc()

#include <stdio.h>

#include <stdlib.h>

int main()

{

int \*ptr, i , n1, n2;

printf("Enter size: ");

scanf("%d", &n1);

ptr = (int\*) malloc(n1 \* sizeof(int));

printf("Addresses of previously allocated memory: ");

for(i = 0; i < n1; ++i)

printf("%u\n",ptr + i);

printf("\nEnter the new size: ");

scanf("%d", &n2);

// rellocating the memory

ptr = realloc(ptr, n2 \* sizeof(int));

printf("Addresses of newly allocated memory: ");

for(i = 0; i < n2; ++i)

printf("%u\n", ptr + i);

free(ptr);

return 0;

}

When you run the program, the output will be:

Enter size: 2

Addresses of previously allocated memory:26855472

26855476

Enter the new size: 4

Addresses of newly allocated memory:26855472

26855476

26855480

26855484

Structure

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

How to define structures?

Before you can create structure variables, you need to define its data type. To define a struct, the struct keyword is used.

Syntax of struct

struct structureName

{

dataType member1;

dataType member2;

...

};

Here is an example:

struct Person

{

char name[50];

int citNo;

float salary;

};

Here, a derived type struct Person is defined. Now, you can create variables of this type.

Create struct variables

When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Here's how we create structure variables:

struct Person

{

char name[50];

int citNo;

float salary;

};

int main()

{

struct Person person1, person2, p[20];

return 0;

}

Another way of creating a struct variable is:

struct Person

{

char name[50];

int citNo;

float salary;

} person1, person2, p[20];

In both cases, two variables person1, person2, and an array variable p having 20 elements of type struct Person are created.

Access members of a structure

There are two types of operators used for accessing members of a structure.

. - Member operator

Suppose, you want to access the salary of person2. Here's how you can do it.

person2.salary

Example: Add two distances

// Program to add two distances (feet-inch)

#include <stdio.h>

struct Distance

{

int feet;

float inch;

} dist1, dist2, sum;

int main()

{

printf("1st distance\n");

printf("Enter feet: ");

scanf("%d", &dist1.feet);

printf("Enter inch: ");

scanf("%f", &dist1.inch);

printf("2nd distance\n");

printf("Enter feet: ");

scanf("%d", &dist2.feet);

printf("Enter inch: ");

scanf("%f", &dist2.inch);

// adding feet

sum.feet = dist1.feet + dist2.feet;

// adding inches

sum.inch = dist1.inch + dist2.inch;